



UNIVERSITY OF SOUTH AUSTRALIA

**Modelling, Analysis and Prototyping of the ODP Trader
using Coloured Petri Nets and Java**

ANDREW A. TOKMAKOFF

B. Eng. Computer Systems Engineering (Hons), 1993

A thesis submitted in total fulfilment
of the requirements for the degree of

Doctor of Philosophy

*Telecommunications Systems Engineering Centre,
Institute for Telecommunications Research,
School of Computer and Information Science and
School of Physics and Electronic Systems Engineering,
Faculty of Information Technology,
University of South Australia.*

March, 1998

Table of Contents

Table of Contents	ii
List of Figures	xii
List of Tables	xv
Abbreviations	xvi
Summary	xviii
Statement of Authorship	xx
Acknowledgments	xxi
Chapter 1	
Introduction	1
1.1 Motivation	1
1.2 Research Aims	2
1.3 Scope	3
1.4 Organisation of the Thesis	3
Chapter 2	
Community Networking, Electronic Commerce and Information Infrastructure	6
2.1 Introduction	6
2.2 Smart Cities	7
2.2.1 Smart Valley Inc.	7
2.2.2 Singapore IT2000	7
2.2.3 The Multimedia Super Corridor	8
2.2.4 The Multi Function Polis	8
2.3 Community Networking	9
2.4 Local loop Transmission Infrastructure	9
2.4.1 Analogue Modems	10

2.4.2 ADSL	11
2.4.3 Cable Modems	11
2.4.4 ISDN	12
2.4.5 Fibre to the Curb/Home	12
2.5 Electronic Commerce	12
2.5.1 Smartcards	13
2.5.2 Entrepreneurial Activity	13
2.6 Motivation for Trading Infrastructure	14
2.7 Summary	14

Chapter 3

Trading in Open Object-based Distributed Systems	15
3.1 Open Object-based Distributed Systems	15
3.2 The Reference Model for Open Distributed Processing	15
3.2.1 Object Model	17
3.2.2 Object Bindings	17
3.2.3 RM-ODP Viewpoints	18
3.2.3.1 Enterprise	18
3.2.3.2 Information	19
3.2.3.3 Computational	19
3.2.3.4 Engineering	19
3.2.3.5 Technology	19
3.3 The Object Management Architecture	20
3.3.1 OMG	20
3.3.2 Common Object Request Broker Architecture	20
3.3.2.1 The Object Request Broker	21
3.3.2.2 Interface Definition Language	21
3.3.2.3 Application Domains	22
3.3.2.4 Inter-Orb Protocol	22
3.4 The Trading Function	22
3.4.1 Basic Trading Entities	22
3.4.2 Service Types	23
3.4.3 Service Offers	24
3.4.4 Trading Interfaces	24
3.4.4.1 Functional Interfaces	24
3.4.4.2 Abstract Interfaces	25
3.4.5 Iterator Interfaces	26
3.4.5.1 Offer Iterator	26
3.4.5.2 Offer Id Iterator	26
3.4.6 Interworking Traders	26
3.4.6.1 Links	27
3.4.7 Offer Space	28
3.4.8 Trading Policies	28
3.4.8.1 Trader Policy	28
3.4.8.2 Import Policy	29

3.4.8.3 Link Policy	30
3.4.8.4 Resolving Policies	30
3.4.8.5 Example of Query Propagation	31
3.5 OMG's Object Trading Service	32
3.5.1 OMG's Object Trading Service	32
3.6 ANSAware Trader	33
3.7 Discussion	33
Chapter 4	
Formal Models of the Trader	34
4.1 RM-ODP Architectural Semantics	34
4.2 SDL '92 Trader Specification	35
4.2.1 Draft Trading Standard Annexes	35
4.2.2 Fischer, Prinz and Vogel	36
4.2.3 Discussion	36
4.3 Lotos Trader Specifications	36
4.4 Z Trader Specification	37
4.5 Object-Z Trader Specification	37
4.6 Message Sequence Chart Trader Specification	38
4.7 Comparison of FDTs	39
4.8 Summary	39
Chapter 5	
Coloured Petri Nets and Design/CPN	41
5.1 Coloured Petri Nets	41
5.2 CPN Analysis Techniques	44
5.2.1 Occurrence Graphs	44
5.2.2 Occurrence Graphs with Equivalences	44
5.2.3 Occurrence Graphs with Symmetries	45
5.2.4 Stubborn Sets	45
5.2.5 Invariants	46
5.3 Design/CPN	46
5.3.1 Editing	46
5.3.1.1 Global Declaration Node	47
5.3.1.2 CPN Structure	47
5.3.1.3 Hierarchies	47
5.3.2 Simulation	48
5.3.3 Analysis	49
5.4 A Top-level Trader Interaction CPN	49
5.4.1 Global Declaration Node	49
5.4.2 Pages in the model	53

5.4.3	Initial Marking	53
5.4.4	Dynamic behaviour	54
5.4.4.1	Exporter–Trader Interaction	54
5.4.4.2	Importer–Trader Interaction	55
5.4.4.3	Importer–Exporter Interaction	56
5.4.5	Analysis of the Top–Level Trader Model	56
5.4.5.1	Simulation	56
5.4.5.2	Occurrence Graph Analysis	58
5.4.5.3	Occurrence Graphs with Equivalences	61
5.5	Discussion of the Model	66
5.5.1	Concurrency	66
5.5.2	Variables on Arcs	66
5.5.3	Non–determinism	67
5.6	Summary	67
Chapter 6		
Modelling the Trading Environment with Coloured Petri Nets ...		68
Part I – Modelling Issues		68
6.1	Scope	68
6.2	Petri Nets and Objects	69
6.3	Modelling Assumptions	71
6.3.1	Static Link and Trader attributes	71
6.3.2	RPC Messaging	71
6.3.3	Type Repository object	71
6.4	Modelling Decisions	71
6.4.1	The Communications Medium	72
6.4.2	Message Format	73
6.4.3	Multiple Object Instantiation	74
6.4.4	Threading within Objects	74
6.4.5	Link pre–fetching	75
6.5	Improved Interworking Protocol	75
6.5.1	Redundant Forwarding of Queries	76
6.5.1.1	Example of Redundant forwarding	76
6.5.1.2	Proposed Solution	77
6.5.2	Actions to perform when a Duplicate Query is detected	77
6.5.3	Non–deterministic Traversal of Linked Traders	78
6.5.3.1	Example of Non–deterministic Traversal	78
6.5.3.2	Proposed solution	79
6.5.4	Summary	80
6.6	Objects in the System	80
6.6.1	Trader	80
6.6.2	Exporter	80
6.6.3	Importer	81

6.6.4 OfferSpace	81
6.6.5 LinkSpace	81
6.7 Part I Summary	82
Part II – CPN Model of the Trading Environment	83
6.8 The Model Hierarchy	83
6.8.1 Global Declaration Node	86
6.8.2 Colour Sets	86
6.8.3 Functions	86
6.8.3.1 Projection Functions	86
6.8.4 Variables	87
6.9 Trading Environment	87
6.10 Auxiliary objects	91
6.10.1 OfferSpace	91
6.10.1.1 get_offers() Method	93
6.10.1.2 add_offer() Method	93
6.10.1.3 Summary	94
6.10.2 LinkSpace	94
6.10.3 Importer	96
6.10.4 Exporter	98
6.11 Trader	99
6.11.1 Trading Interfaces	99
6.11.2 Inter–Object Interfaces	101
6.11.2.1 OfferSpace Interface	102
6.11.2.2 LinkSpace Interface	103
6.11.3 Functional Interfaces	103
6.11.3.1 Lookup Interface	105
6.11.3.2 Register Interface	106
6.11.3.3 Proxy, Link and Admin Interfaces	106
6.11.4 Abstract Interface	107
6.11.5 Functional Operations	107
6.11.6 Query Operation	109
6.11.6.1 Process Request	111
6.11.6.2 Import from Links	115
6.11.7 Export Operation	118
6.12 Refining the Model	119
6.12.1 Stages of Modelling	119
6.12.2 Object–based Modelling	120
6.12.3 Modifying Colour Set Definitions	121
6.13 Comments on High–level Petri Nets and Design/CPN for Modelling OOBDS	121
6.13.1 Petri Nets	121
6.13.2 Design/CPN	122
6.13.2.1 Positive Points	122
6.13.2.2 Negative Points	122
6.14 Summary	122

Chapter 7

Analysis of the Trader	123
7.1 Methodology for detecting errors in the model	123
7.1.1 Examples of modelling errors	123
7.1.2 Simulation	124
7.1.3 Occurrence Graph Analysis	124
7.1.4 Occurrence Graphs with Equivalence Classes	126
7.2 Permutations of merge_policy_options()	126
7.3 Initialisation of the Trading Environment	128
7.4 Approach to Analysis of the Trading Environment	129
7.5 Analysis of a Trader Configured as a Standalone	130
7.5.1 Single Non-threaded Trader Query	130
7.5.1.1 Initialisation	130
7.5.1.2 OG Analysis	131
7.5.2 Single Trader with Concurrent Request Servicing	133
7.5.3 Single Trader with Importer-Exporter interaction	135
7.5.4 Summary	136
7.6 Verification of Multiple Independent Traders	138
7.6.1 Multiple Autonomous Objects	138
7.6.1.1 Initialisation	138
7.6.1.2 OG Analysis	138
7.6.2 Multiple Autonomous Threaded Objects	139
7.7 Verification of Interworking Traders	140
7.7.1 Recursive Definition of Interworking Traders	140
7.7.2 Reducing the size of the Problem	141
7.7.3 Interworking Query Stopping Conditions	141
7.7.3.1 hop_count=0 (Scenario 1)	142
7.7.3.2 Total available links=0 (Scenario 2)	144
7.7.3.3 Unified Policy follow behaviour=local_only (Scenario 3)	145
7.7.3.4 Merged Policy=if_no_local and local OfferSpace search is successful (Scenario 4)	146
7.7.3.5 Valid links to follow=0 (Scenario 5)	148
7.7.3.6 Revisit Trader with smaller hop_count (Scenario 6)	149
7.8 Deterministic Traversal of the Offer Space	150
7.9 Combined Trading Topology (All Stopping Conditions)	152
7.9.1 Stopping condition 1	153
7.9.2 Stopping condition 2	154
7.9.3 Stopping condition 3	154
7.9.4 Stopping condition 4	154
7.9.5 Stopping condition 5	155
7.9.6 Stopping condition 6	156
7.9.7 Conclusion	156
7.9.8 OG Analysis of Combined Scenario	158

7.10 Using High-level Petri Nets and Design/CPN for Analysis of OOBDSs	158
7.10.1 High-level Petri Nets	158
7.10.2 Design/CPN	159
7.11 Summary	159

Chapter 8

Prototyping the Trader	161
8.1 Motivation	161
8.2 Aims and Scope of the Prototype	161
8.3 Requirements	162
8.4 Design Decisions	162
8.4.1 Implementation Language	162
8.4.1.1 Object-oriented	163
8.4.1.2 Distributed	163
8.4.1.3 Portable	163
8.4.1.4 Multi-threaded	164
8.4.1.5 Easy to program	164
8.4.1.6 Performance	164
8.4.2 Support for Distribution	164
8.4.2.1 Sockets	165
8.4.2.2 CORBA	165
8.4.2.3 RMI	165
8.4.3 Trader Front-end	166
8.4.3.1 Standalone Application	166
8.4.3.2 Downloadable Client in a Browser	167
8.5 Prototype Trading Domain	167
8.6 Using the CPN Model for Implementation	167
8.6.1 Implementation based upon early CPN Models	167
8.6.1.1 Message passing	167
8.6.1.2 Multi-threading	168
8.6.1.3 Offer Space and Link Space objects	168
8.6.1.4 Limitations	168
8.6.2 Re-implementing using RMI and Trader Standard 96	169
8.6.2.1 RMI as a Communications Medium	169
8.6.3 CPN to Java mapping	169
8.6.3.1 Data structures and Logic	170
8.6.3.2 Concurrency	170
8.6.3.3 Offer Space and Link Space objects	170
8.7 Implementation Source code	171
8.7.1 Class Hierarchy	171
8.7.2 Interfaces	171
8.7.3 Data structure classes	171
8.7.4 Applets	172
8.7.5 Sample Services	173

8.8	Running the Trading Environment	174
8.8.1	Compilation	174
8.8.2	Starting the Objects in the System	174
8.9	Testing of the Trading Environment	175
8.9.1	OfferSpace	176
8.9.2	Link Space	176
8.9.3	Trader	176
8.9.3.1	Standalone	176
8.9.3.2	Interworking Traders	176
8.10	Limitations	178
8.11	Summary	179

Chapter 9

Conclusions	180
9.1 Contributions of the Dissertation	180
9.2 Future Work	181
9.2.1 Extension of the Trader model	181
9.2.2 Stubborn Set Analysis of the Trader	181
9.2.3 Extended Trading Test-bed	182
9.2.4 Investigate effective Trader topologies	182
9.2.5 Investigate automatic mapping from CPN models to Java	182
9.2.6 Extension of Interworking Analysis	182
References	183

Appendix A

CPN model of the Trader	195
A.1 Global Declaration Node (GDN#1)	195
A.2 OEOS Functions	207
A.2.1 EquivBE()	207
A.2.2 EquivMark()	211
A.3 Occurrence Graph Functions	214

Appendix B

Java Source code: Trader Implementation	215
B.1 Interfaces	215
B.1.1 Lookup.java	215
B.1.2 Register.java	215
B.1.3 Trader.java	216
B.1.4 LinkSpace.java	216
B.1.5 OfferSpace.java	217
B.2 Abstract Data Types	218

B.2.1	impPolicyRecord.java	218
B.2.2	linkRecord.java	219
B.2.3	offerRecord.java	221
B.2.4	reqIdRecord.java	223
B.2.5	linkStore.java	224
B.2.6	offerStore.java	225
B.2.7	traderPolicyRecord.java	226
B.3	Object implementations	227
B.3.1	Trader_impl.java	227
B.3.2	OfferSpace_impl.java	235
B.3.3	LinkSpace_impl.java	241
B.4	Applets	245
B.4.1	OfferSpaceApplet.java	245
B.4.2	LinkSpaceApplet.java	247
B.4.3	TradGuiApplet.java	248
B.5	Sample Service Types	253
B.5.1	servProps.java	253
B.5.2	serviceRecord.java	253
B.5.3	pizzaServRec.java	255
B.5.4	usedCarServRec.java	257

Appendix C

	Prototype Trader Trace Output	260
C.1	OfferSpace	260
C.2	LinkSpace	263
C.3	Traders	264
C.3.1	Trader 1	264
C.3.2	Trader 2	265
C.3.3	Trader 3	265
C.3.4	Trader 4	266
C.4	Importer Applet	267

Appendix D

	Some thoughts on Trading Applications and Topologies	268
D.1	Layers of Trading	268
D.1.1	Load Balancing in Distributed Operating Systems	269
D.1.2	Telecommunications Information Networking Architecture	269
D.1.3	Open Distributed Applications	270
D.1.4	Trading in Mobile Environments	271
D.1.4.1	Re-binding of Services	271
D.1.4.2	Changing Quality of Service	271
D.1.4.3	Process Migration for load balancing	271
D.2	Trader Organisation Topologies	272

D.2.1	Server Farms	272
D.2.2	Planes of Linked Traders	273
D.2.3	Examples	274
D.2.3.1	Local level Trading	274
D.2.3.2	National level Trading	274
D.2.3.3	International level Trading	275
D.3	Summary	275
 Appendix E		
	Publications	276
E.1	Conferences and Workshops	276
E.2	Journals	279
 Appendix F		
	CD containing Source code and CPN Trader model	280
F.1	Design/CPN Trader Model	280
F.2	Trader Source Code	280
F.3	Directory Listing	280

List of Figures

Fig. 3.1: The RM–ODP Object Model	17
Fig. 3.2: Explicit Binding between objects	18
Fig. 3.3: Implicit Binding between objects	18
Fig. 3.4: Common Object Request Broker Architecture	21
Fig. 3.5: Sequence of Interactions between Exporter, Trader, and Importer	23
Fig. 3.6: The Trading Offer Space	28
Fig. 3.7: Example of unifying policies	31
Fig. 5.1: A simple CPN with one transition and three places	43
Fig. 5.2: Simple CPN after transition Go occurs	43
Fig. 5.3: Example of the diamond sub–structure	46
Fig. 5.4: An example of CPN Hierarchy	48
Fig. 5.5: Top–Level Trader Interaction Model	50
Fig. 5.6: Global Declaration Node of the Top–Level CPN	52
Fig. 5.7: Hierarchy page of the Top–Level CPN	53
Fig. 5.8: Initial Marking of the Top–Level CPN	54
Fig. 5.9: Simulation report from the Top–Level CPN	57
Fig. 5.10: Marking of node 5060 from OGA analysis of the Top–Level CPN . . .	59
Fig. 5.11: Marking of node 5061 from OGA analysis of the Top–Level CPN . . .	59
Fig. 5.12: Top–Level Model Occurrence Graph	60
Fig. 5.13: A MapOut function used by the OEOS tool	61
Fig. 5.14: The Equivalent Marking function used by the OEOS tool	62
Fig. 5.15: The Equivalent Binding Element function used by the OEOS tool . . .	63
Fig. 5.16: Command used to set Equivalence Functions for the OEOS tool	64
Fig. 5.17: Top–Level Model Occurrence Graph with Equivalences	65
Fig. 5.18: Marking of node 2685 from OEOS analysis of the Top–Level CPN . .	66
Fig. 6.1: The “Objects inside Petri Nets” concept	70
Fig. 6.2: The “Petri nets inside objects” concept	70

Fig. 6.3: 2 Objects, 4 methods and 1 common place	73
Fig. 6.4: 2 Objects, 4 methods and a mesh of inter-connecting arcs	73
Fig. 6.5: Trader blocks while it processes Initial Request	75
Fig. 6.6: Redundant Query propagation	76
Fig. 6.7: Non-deterministic Query propagation	79
Fig. 6.8: Hierarchy page (Hierarchy#10010)	83
Fig. 6.9: Trading Environment (Trad_Env#2)	88
Fig. 6.10: A CPN of the OfferSpace object (Offer_Space#4)	91
Fig. 6.11: A CPN of the LinkSpace object (Link_Space#5)	94
Fig. 6.12: Three linked Traders	95
Fig. 6.13: Importer with two Queries	96
Fig. 6.14: Exporter object with one Offer	99
Fig. 6.15: Trading Interfaces (Trad_Int#3)	100
Fig. 6.16: Inter-Object Interfaces (Inter_Obj#7)	102
Fig. 6.17: Offer Space Interface (OSI#10)	103
Fig. 6.18: Link Space Interface (LSI#11)	104
Fig. 6.19: Functional Interfaces (Functional#8)	104
Fig. 6.20: Lookup Interface (LUI#12)	105
Fig. 6.21: Register Interface (REGI#19)	106
Fig. 6.22: Functional Operations (Funct_Opns#9)	108
Fig. 6.23: Query Method (Query#13)	110
Fig. 6.24: Process Request (Process_Req#14)	113
Fig. 6.25: Import from Links (Link_Import#15)	116
Fig. 6.26: Export Operation (Export#19)	119
Fig. 7.1: Calculating the OG using Breadth then Depth	125
Fig. 7.2: Analysis goals	129
Fig. 7.3: Trader servicing a single Query	130
Fig. 7.4: The Trader services multiple Queries concurrently	133
Fig. 7.5: Reduced marking of node 376 from OEOS analysis of Importer-Exporter interaction	136
Fig. 7.6: Reduced marking of node 366 from OEOS analysis of Importer-Exporter interaction	137
Fig. 7.7: 2 Traders and 2 independent Queries	138
Fig. 7.8: 2 Traders and 4 independent Queries	139
Fig. 7.9: Query stops when hop_count=0	142
Fig. 7.10: Reduced marking of node 2628 from OG analysis of scenario 1	143

Fig. 7.11: Query stops when there are no links to follow	144
Fig. 7.12: Reduced marking of node 758 from OG analysis of scenario 2	145
Fig. 7.13: Query stops when unified_policy=local_only	146
Fig. 7.14: Query stops with a local match and merged follow_policy=if_no_local	147
Fig. 7.15: Reduced marking of node 296 from OG analysis of scenario 3	147
Fig. 7.16: Reduced marking of node 844 from OG analysis of scenario 4	148
Fig. 7.17: Query stops when there are no valid links to follow	148
Fig. 7.18: Query stops when it re-visits a Trader	149
Fig. 7.19: Reduced marking of node 758 from OG analysis of scenario 5	150
Fig. 7.20: T1 gets duplicates with different hop_count values	151
Fig. 7.21: Non-deterministic Query propagation	151
Fig. 7.22: Combined Scenario	152
Fig. 7.23: Query trace 1	153
Fig. 7.24: Query trace 2	154
Fig. 7.25: Query trace 3	155
Fig. 7.26: Query trace 4	155
Fig. 7.27: Query trace 5	156
Fig. 7.28: Query trace 6	157
Fig. 7.29: Concurrent independent traces	157
Fig. 8.1: Class hierarchy from javadoc	172
Fig. 8.2: Block Diagram of class relationships	173
Fig. 8.3: Four linked Traders	177
Fig. C.1: Four linked Traders	260
Fig. D.1: Three Layers of Trading	268
Fig. D.2: Object-Based Desktop Applications Interworking	270
Fig. D.3: Server Farm using a Trading path	272
Fig. D.4: City-level Traders	273
Fig. D.5: National-level Traders	274
Fig. D.6: International-level Traders	274

List of Tables

Table 1 – OGA Generation statistics from the Top–Level CPN	58
Table 2 – OEOS Generation statistics from the Top–Level CPN	64
Table 3 – Combinations of hop_count and follow_option for links=0	131
Table 4 – Combinations of hop_count and follow_option for links<>0	132
Table 5 – Simulation Steps for Single Trader Concurrent Queries	134
Table 6 – OG size for Single Trader Concurrent Queries	134
Table 7 – OEOS size for Single Trader Concurrent Queries	135
Table 8 – OGs for Concurrent Importer/Exporter Trader Interaction	136
Table 9 – OG size for Multiple Traders	139
Table 10 – Scenario Stopping conditions	141
Table 11 – OG size for Stopping Scenario 1	143
Table 12 – OG size for Stopping Scenario 2	144
Table 13 – OG size for Stopping Scenario 3	146
Table 14 – OG size for Stopping Scenario 4	147
Table 15 – OG size for Stopping Scenario 5	149
Table 16 – OG size for Stopping Scenario 6	150
Table 17 – OG size for Deterministic Traversal	152
Table 18 – OG size for Combined Scenario after 519 seconds	158

Abbreviations

ADSL	Asymmetric Digital Subscriber Loop
CATV	Cable Television
CD	Committee Draft
CORBA	Common Object Request Broker Architecture
COS	Common Object Service
CPN	Coloured Petri Net
DCOM	Distributed Common Object Model
DII	Dynamic Invocation Interface
DIS	Draft International Standard
DOS	Distributed Operating System
DPE	Distributed Processing Environment
DSI	Dynamic Skeleton Interface
E-Commerce	Electronic Commerce
FDT	Formal Description Technique
FTTC	Fibre to the curb
FTTH	Fibre to the home
HFC	Hybrid Fibre Coaxial
IDL	Interface Definition Language
IIOP	Internet Inter-Orb Protocol
IS	International Standard

ISO	International Organisation for Standardisation
IT&T	Information Technology and Telecommunications
IEC	International Electrotechnical Commission
JVM	Java Virtual Machine
LAN	Local Area Network
LOTOS	Language for Temporal Ordering Specification
MAN	Metropolitan Area Network
MFP	Multi Function Polis
OEOS	Occurrence Graph with Equivalences and Symmetries
OG	Occurrence Graph
OGA	Occurrence Graph Analyser
OMG	Object Management Group
OOBDS	Open Object-based Distributed Systems
ORB	Object Request Broker
PSTN	Public Switched Telecommunications Network
QoS	Quality of Service
RMI	Remote Method Invocation
RM-ODP	Reference Model for Open Distributed Processing
SDL	Specification and Description Language
SML	Standard Meta Language
TINA	Telecommunications Information Networking Architecture
VOD	Video on Demand
WWW	World Wide Web

Summary

Trading is a fundamentally important aspect in the realisation of Open Object-based Distributed Systems (OOBDS), and has been the joint topic of standardisation by the International Organisation for Standardisation (ISO), the International Electrotechnical Commission (IEC) and the International Telecommunications Union (ITU-T) as part of their work on the Reference Model for Open Distributed Processing (RM-ODP).

Trading is an information infrastructure service that allows software entities to advertise, or *export* a service or resource to a trusted third party, known as the Trader. The service or resource being exported usually has properties associated with it, which may be static or dynamic. The Trader maintains a database of exported services which includes the type of service which is advertised and properties associated with the service.

The application of Trading to Electronic Commerce is an area yet to be explored, where the services traded are of a commercial nature, and the Trader is used to match clients with appropriate service or resource providers. With the potentially widespread use of Traders for electronic service location, it is important that the Trading standard is error-free and unambiguous. It is also important to ensure that the Trader is as efficient as possible, especially when interworking with other Traders. One way to ensure correctness of the Trader is to create a formal model which can be analysed to verify that the model behaves as expected under all conditions.

Coloured Petri Nets (CPNs) have been used to model the RM-ODP Trading standard. This has resulted in the creation of two models which view Trader interactions at different levels of abstraction. The first model provides a description of the interactions between the Trader, Importers and Exporters. The second model provides a more detailed description of the internal logic of the Trader and concentrates on the interworking protocol used by Traders when cooperating.

Analysis of the Trader models has been performed using Occurrence Graphs (OGs). In certain scenarios, it was found that the use of OG's with Equivalence Classes resulted in a significantly smaller equivalent OG. This was useful when verifying correct behaviour of the Trader, especially in scenarios involving interworking where the CPN model displays a great deal of concurrent behaviour.

As a product of the modelling exercise, a number of omissions and ambiguities in the RM–ODP Trading standard were detected, especially with regard to the interworking protocol. More specifically, these omissions relate to:

- an incomplete behavioural specification regarding detection of duplicate Queries,
- redundant forwarding of Queries to linked Traders whilst interworking,
- non–deterministic traversal of the entire linked Traders Offer Space which may result in an incomplete matching of appropriate offers.

As a consequence of detecting these ambiguities/omissions in the Trading Standard, a number of suggestions for improvement to the protocol have been proposed. These suggestions were incorporated into the CPN model and verified using OGs. Traders were placed in a number of scenarios and the resultant behaviour analysed using OGs. Using the results from a number of scenarios, conclusions about the behaviour of the Trader model in more complex scenarios have been made.

Having verified the Trader interworking protocol, the CPN model was used as the basis for creating a prototype Trader. This Trader was used in a concept demonstrator which applies Traders to the provision of a distributed electronic commerce lookup service. The prototype is implemented using Java and its built–in distributed object facility known as Remote Method Invocation (RMI).

Statement of Authorship

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

Andrew. A. Tokmakoff B. Eng. (Hons)

Acknowledgments

I would like to thank Professor Jonathan Billington for taking me on as a doctoral student six months into my doctoral studies. He was the driving force behind my research and indeed, introduced me to both Trading and Coloured Petri Nets. I am very grateful for his helpful comments and advice throughout my candidature. I would also like to thank the other members of TSEC who provided me with valuable feedback during our weekly meetings.

I would never have been able to undertake this research project without the assistance of the School of CIS which provided me with a UniSA Postgraduate Research Scholarship. Just as important was the financial and motivational support of my parents, Vic and Colleen Tokmakoff, who helped me attend a number of conferences in order to present the results of this research internationally.

I would like to thank my good friend Adam Harrison, with whom I have had many in-depth philosophical discussions, regarding computing, the future and other interesting topics. I am also extremely grateful to Alex Farkas who stepped in at the last hour as a proxy secondary supervisor and provided me with useful advice when it was required. Cheers to Steve Semple of Vision Abell for granting me study leave to finalise this thesis.

Finally, I would like to thank my girlfriend Rebecca Greenfield, for enduring the many sacrifices which were necessary in the name of research over the last three and a half years.

Chapter 1

Introduction

This chapter provides context and motivation for the investigation presented in this thesis. It also includes the study's aims, scope and structure.

1.1 Motivation

In recent years, it has become increasingly evident that society is becoming “wired”, and information technologies are finding their way from the business domain into the home. In particular, the Personal Computer has increased in performance whilst decreasing in real cost. Couple this with continually improving communications infrastructure and service provision by telecommunications providers [113], and society is poised for the emergence of wide-scale community networking [91]. This term refers to the widespread use of networked information technologies by the general community on a metropolitan scale.

Of particular interest to many businesses is the opportunity to implement an Electronic Commerce strategy, which will provide them with a new medium for conducting business with an audience ranging from the local to the global community. This raises a new problem for consumers of these services : the discovery of electronic services and resources [26].

Trading [16] is an information infrastructure service which allows software entities to advertise, or *export* a service or resource to a trusted third party, known as the Trader. The service or resource being exported usually has properties associated with it, which may be static or dynamic. The Trader maintains a database of exported services which includes the type of service being advertised and its associated properties.

Trading is a fundamentally important part of the realisation of Open Object-based Distributed Systems (OOBDS) [3] and has been the topic of standardisation by the International Organisation for Standardisation (ISO), the International Electrotechnical Commission (IEC) and the

International Telecommunications Union (ITU-T) as part of their work on the Reference Model for Open Distributed Processing (RM-ODP) [3].

The RM-ODP Trading standard was developed to facilitate the dynamic location of resources or services, such as for example, a printing service in an OOBDS. The application of Trading to Electronic Commerce is an area yet to be explored, where the services traded are of a commercial nature, and the Trader is used to match clients with appropriate service or resource providers.

With the possibly widespread use of Traders for electronic service location, it is important that the Trading standard is error-free and unambiguous. It is also important to ensure that the Trader is as efficient as possible, especially when interworking with other Traders.

In order to engineer reliable Open Distributed Systems, there is a need for formal modelling techniques. Such techniques must be capable of capturing the essence of a system in a manner which is readable and moreover, utilise a mathematical basis that allows the system to be formally analysed. One way to ensure correctness of the Trader is to create a formal model which can be analysed to verify that the model behaves as expected under all conditions. Coloured Petri Nets (CPNs) [66] are a formalism which has been developed for modelling and analysis of concurrent systems. They have a graphical form which is intuitively readable, and an underlying formal semantics which provides a basis for formal analysis. In addition, a tool-set exists for the creation, simulation and analysis of CPN models called Design/CPN[™] [83].

1.2 Research Aims

In order to obtain a clear and unambiguous description of the ODP Trader, it is important to make use of a formal description technique. The principal aim of this thesis is to use the RM-ODP Trading standard as a specification for the creation of an extendable, executable model of the Trader using CPNs. When developing the model, issues including concurrency and hierarchical structure will also be addressed.

Having created a CPN model of the Trader, a second aim is to verify this emerging standard, with special emphasis being placed on the interworking protocol used by the Trader. This will require analysis of its dynamic behaviour using the Occurrence Graph [66] and Occurrence Graphs with Equivalence Class [68] techniques, both of which are implemented in the Design/CPN[™] software package.

Analysis of the model aims to verify the correctness of the Trader, as specified in the ODP Trading standard. It also aims to identify any ambiguities and errors within the standard, and to suggest improvements to the standard in such cases.

Having verified the Trader interworking protocol, the third aim of the thesis is to use the verified CPN model as a specification for the implementation of a prototype Trader. The prototype aims to demonstrate the application of Traders to the domain of service location in an electronic commerce environment.

1.3 Scope

The scope of the investigations is large, ranging from smart cities and electronic commerce to formal modelling, verification and prototyping of an international standard.

Much of the work contained within this thesis deals with modelling and analysis of the ODP Trader using hierarchical Coloured Petri Nets. In particular, a focus is placed on the Trader's ability to cooperate or *interwork* with other Traders.

The application of Traders to the domain of a distributed commercial services lookup facility is investigated. Utilising the CPN model of the Trader and its environment, a mapping from the CPN model into a prototype implementation using Java as the implementation language is presented.

1.4 Organisation of the Thesis

The thesis is organised into the following chapters and appendices:

- **Chapter 2 : Smart Cities and Electronic Commerce**

This chapter introduces Community Networking, Smart Cities and the Information Infrastructure required for the widespread adoption of networked computing by the greater population. Sections of this chapter are based upon work published in [25].

- **Chapter 3 : Trading and Open Object-based Distributed Systems**

This chapter provides an introduction to Open Object-based Distributed Systems including standardisation efforts in the area and possible implementation strategies. It also provides an introduction to the Trading function, explaining the entities which are involved in service Trading and the mechanisms used to control cooperating Traders.

- **Chapter 4 : Formal Models of the Trader**

This chapter examines existing formal models of the Trader including those utilising Z, SDL and LOTOS. As part of the investigation, areas for improvement when modelling the Trader are identified.

- **Chapter 5 : Coloured Petri Nets and Design/CPN**

In this chapter, Coloured Petri Nets are described using a high-level model of Trader interactions as an example. This chapter also provides an introduction to the Design/CPN[™] tool, including an overview of its editing, simulation and analysis features. A paper based upon the example presented in this chapter has recently been published [35].

- **Chapter 6 : Modelling Traders with CPNs**

A CPN model of a multi-threaded interworking Trader is presented. The model is

described in significant detail, including a discussion of data and control flows throughout the model. The work presented in this chapter is based upon evolutionary modelling of the Trader and has been published at different stages of development [27, 28, 29, 31, 32]. This chapter prepares the reader for formal analysis of the model which is presented in Chapter 7.

- **Chapter 7 : Analysis of the Trader**

The model of the Trader is firstly analysed as a standalone entity which can service multiple requests concurrently. It is then analysed when interworking with other Traders, using a series of stopping conditions and a recursive perspective on the propagation of Queries. Occurrence Graphs and Occurrence Graphs with Equivalence classes are used to generate the state space of the Trader in a number of scenarios, where equivalence classes are used to reduce the state space under certain conditions.

- **Chapter 8 : Creating a Prototype of the Trader using Java**

In this chapter, a prototype Trading Environment based upon the CPN model of the Trader and its environment is presented. Requirements of the prototype and important design decisions are presented. In addition, a description of the the structure of the Java code is given, along with an introduction and operating instructions for the test-bed application domain.

- **Chapter 9: Conclusions and further work**

In this final chapter, the contributions of the thesis are identified and areas for further research are suggested.

- **Appendix A: Design/CPN model of the Trader**

This appendix contains some elements of the Trader CPN model which were not included in Chapter 6.

- **Appendix B: Source Code listing of Trader Prototype**

This appendix contains the Java source code listings of the prototype Trading environment.

- **Appendix C: Prototype Trader Trace Output**

This appendix contains trace-level output from a sample run of the Trading Environment.

- **Appendix D: Some thoughts on Trading Applications and Topologies**

In this appendix, a number of application domains for Trading are suggested, including load balancing in a Distributed Operating System, service location in mobile computing environments and an Electronic Commerce service location function. This appendix is based upon work published in a number of papers dealing with possible application

domains of Trading [26, 30, 33, 34]. In addition, some topologies for effective interconnection of Traders are proposed including hierarchical and tree-based link structures.

- **Appendix E : Publications**

This appendix provides abstracts of publications related to the work presented in this thesis.

- **Appendix F : CD-ROM**

This appendix contains the Design/CPN models of the Trading Environment presented in Chapter 6 and Analysed in Chapter 7. It also contains the prototype's Java source code, a copy of the JDK1.1 for Linux and the sample Trace output.

Chapter 2

Community Networking, Electronic Commerce and Information Infrastructure

This chapter investigates the current worldwide phenomenon of community networking and examines relevant projects around the world. The role of Electronic Commerce is discussed, as is the importance of information infrastructure to the realisation of widespread community networking.

2.1 Introduction

There are many aspects that are driving today's information revolution, ranging from the steady advances in the power of microprocessors [89] to an increasingly information-literate younger generation [90]. The widespread adoption of networked computers has advanced from the domain of business into the home. Users are no longer content to use a standalone computer and are demanding the ability to interact and exchange information with other users. As a consequence of these driving factors, the concept of community networking is becoming a reality, as witnessed by several pilot projects around the world.

Community networking provides an opportunity for citizens to interact as they have never done before. Information and services that are of a local nature can be made available in a simple and cost-effective manner. In order to make such an information medium effective, however, it is important that users have easy access to networked computing. The scale of community networking may range from city-wide to municipal to global.

As a consequence of community scale networking, Electronic Commerce (E-Commerce) is expected to become an increasingly significant aspect of businesses which utilise the rapidly emerging communication infrastructure being created by telecommunications companies. Education will benefit from the installation of an advanced information infrastructure, with the prospect of lifetime learning becoming an imperative [109].

This chapter will show how E–Commerce provides a significant application domain for service Trading, where the problem of resource location in a distributed system becomes apparent.

2.2 Smart Cities

There are a number of projects underway throughout the world which may be considered to be “Smart City” projects. In general, such projects are city–wide in scale and aim to provide significant improvements in services available to residents and businesses located there. These projects could be considered test–beds for the widespread use of leading edge information technology. It is anticipated that results obtained from Smart City projects will be used in the widespread adoption of electronic services networking in the community at large. This is the concept of Community Networking.

Smart City projects range from local government initiatives [92], through to large scale, long–term city improvements [95]. One of the common driving factors behind these projects is to improve the region’s commercial viability through the development and implementation of advanced information systems and infrastructure. In particular, there are major projects underway in Silicon Valley, U.S.A [93], Singapore [94], Malaysia[111] and Adelaide, Australia [95] which aim to provide improved business functionality to the region through the adoption of E–Commerce initiatives.

2.2.1 Smart Valley Inc.

This project is one of the initiatives of the Joint Venture Silicon Valley Network and aims to “demonstrate the positive value of the application of technology in education, healthcare, local government, business and the home” [93]. The Smart Valley Initiative aims to “improve the economic vitality and quality of life in the greater Silicon Valley region”. The main users of the regional network are schools, local businesses and members of the community. The Smart Valley Public Access Network has been created to provide free Internet access in public locations such as schools and libraries.

In 1994, CommerceNet was created by Smart Valley Inc. as a large scale trial of electronic commerce via the Internet. It is now a separate non–profit organisation and is growing rapidly, with a large number of affiliated organisations and business members around the globe. It aims to promote the use of Internet technologies for electronic commerce through the adoption of pilot projects and the dissemination of project results to CommerceNet member organisations. This means that members of CommerceNet are able to benefit from other members’ experience in the implementation of electronic commerce systems.

2.2.2 Singapore IT2000

The Singapore IT2000 : Intelligent Island project [94] aims to connect schools, factories, workplace and homes to a high speed data network by the year 2000, thereby improving communications and information opportunities for the island’s citizens.

The Intelligent Island project is aimed toward improving the way business is conducted and, in addition, to providing the people of Singapore with improved living conditions, through the application of technologies to leisure services such as cable television (CATV) and video on demand (VOD).

When developing the IT2000 plan, the major economic sectors of Singapore were consulted to determine their requirements for such a system. The designers obtained information from many diverse areas of business, thereby allowing the IT2000 project to provide benefits to more than just the traditional users of information technology. It is expected that this will foster the creation of many new applications.

2.2.3 The Multimedia Super Corridor

The Multimedia Super Corridor (MSC) is a project currently progressing in Malaysia as part of its 2020 vision national agenda. It has been described by Wired magazine as a project which will attempt to “turn .. jungle into a futuristic, fibered, 30-mile-long technopolis” [110]. A more official description indicates that the MSC aims to accelerate Malaysia’s entry into the Information Age [111].

The project is based upon a greenfield *corridor* which is 15 km wide and 30 km long, stretching from downtown Kuala Lumpur to the Kuala Lumpur International Airport which is also under construction. Within this corridor, two Smart cities are to be built: *Putrajaya* which will become the next administrative capital of Malaysia, and *Cyberjaya* which will be an “intelligent city with multimedia industries, R&D centres, a Multimedia University and headquarters for a number of multinational corporations” [111].

As a driving force behind the MSC, seven major *Flagship Applications* have been identified, which include the following: Electronic Government, Multi-Purpose Card, Smart Schools, Telemedicine, R&D clusters, Worldwide Manufacturing Webs and Borderless Marketing.

Cyberjaya is planned to accommodate a maximum population of 240,000 people and is to be constructed over four phases, commencing in 1997 and reaching completion in 2020. It is intended to be a “sustainable intelligent city tailored to meet the living and business needs of multimedia developers and users” [112].

2.2.4 The Multi Function Polis

The Multi-Function Polis (MFP) is a joint undertaking of the Australian Federal and South Australian governments. The project’s long-term aims concern economic, social and environmental issues with a global rather than local focus. The prime objective of the project is to create a community that is internationally recognised as a model for sustainable development based upon advanced information technology and telecommunications (IT&T) [96].

A 620 hectare urban community development known as “Mawson Lakes” is currently under construction and is expected to take 10 years to complete. The site includes the Levels campus of the University of South Australia, and the adjacent Technology Park. One of the main aims with

respect to information technology is to “demonstrate the utility of advanced IT&T services in the home ... measured by efficiencies in running costs and improved quality of life for residents” [96]. For the business community, the MFP aims to “deliver advanced IT&T-enabled services to businesses and institutions, both off and on the site, that are commercially sustainable and provide substantial and measurable gains in productivity” [96].

2.3 Community Networking

There is a growing acceptance of community networking as an important contributor to community well-being [107]. As information technology makes its way into the home, it is important that there are public electronic places that allow citizens to interact and discuss issues that are important to them. Such a facility can be provided by community networks, where members of the community can participate in discussions of local significance. In some cases, local government has been responsible for the creation of such electronic resources [92]. In other cases, it has been the enthusiasm and hard work of volunteers [97] that has created a public information service. In either case, the result is a freely accessible public resource that provides citizens with a means for obtaining relevant local information, and the opportunity to discuss matters on a local (*listserv* mailing lists) and global level (*Usenet* newsgroups).

As an example of where community networking could be useful, consider the ongoing installation of two Hybrid Fibre-Coaxial (HFC) cable networks in Australia (see section 2.4 on Distribution Technologies). The cable network is being strung between existing electricity poles and has been the subject of widespread objections by members of the community who consider it to be an ugly intrusion on the landscape.

If there was a community network in place, it would be possible for citizens to voice their opinion on this matter in specialised discussion forums, or directly via e-mail to their local representatives on the council. An informed debate on the matter would allow concerned citizens a forum to voice their opinions and increase the possibility of finding a solution to the situation. Without such a forum, the only forms of objection possible are a letter to council and the physical blockade of workmen attempting to install the network [108]. Unfortunately, little dialogue has been entered into, and the problem is far from being resolved.

Ironically, with the installation of the HFC network, more users will be connected to a high-speed data service and community networks will become possible, benefiting the community as a whole.

2.4 Local loop Transmission Infrastructure

In order to make Smart Cities, Community Networking and E-Commerce a reality, it is necessary to provide an advanced communications infrastructure. This includes technology

which “moves the bits” and also encompasses software that manages the infrastructure and provides users with different types of telecommunication services.

There are a number of different technologies that may be used for the provision of data communications services to consumers. They range from fibre-to-the-home (FTTH) [103], which provides the largest available end-user bandwidth, to analogue modems (see section 2.4.1) operating over the Public Switched Telephone Network (PSTN). FTTH requires significant infrastructure investment whereas analogue modems can make use of existing twisted pair telephone cables. An alternative to FTTH is the installation of a Hybrid Fibre Coax (HFC) system which requires significant investment, but utilises existing coaxial cable technologies, thereby reducing the complexity of the end-equipment in the consumer’s premises.

Australia’s two major Telecommunications Network Operators, Telstra [144] and Optus Communications [146] have been undertaking extensive improvements to the information infrastructure of Australia. Both companies have installed significant amounts of fibre optic cable throughout Australia’s capital cities. The immediate application which will utilise this investment in infrastructure is expected to be Video Entertainment in the form of CATV.

There is a great deal of competition for the telecommunications services market and it will become more competitive as the number of services offered to consumers increases. In an attempt to foster the rapid installation of information infrastructure, the MFP project has named Telstra as the “preferred supplier” for information services in the MFP stage one housing development [95].

2.4.1 Analogue Modems

At present, the fastest standardised data rate of analog modems is the V.34 bis standard that allows full duplex communication at 33.6 Kbps. This is a significant increase over the 2.4 Kbps modems that were prevalent in the late 1980’s. Proprietary solutions from US Robotics [142] and Rockwell [143] have extended this data rate to 56Kbps although neither of these competing solutions have yet been standardised. It is possible for users to obtain satisfactory performance from 33.6 Kbps modems when used for e-mail and World Wide Web (WWW) browsing. Of today’s Internet users, 70% consider e-mail and 30% the WWW to be the primary benefit of Internet connectivity [99]. If local calls remain un-timed, it seems certain that analogue modems will remain popular even into the 21st century since they are cheap and relatively easy to operate.

WebTV Networks launched a new technology known as WebTV [98] in 1996. It consists of the following components:

- a WWW browser that utilises a proprietary Graphical User Interface (GUI) designed for use with a remote control,
- image enhancement hardware which reduces flicker on conventional TV sets,
- a smart card slot and
- a 33.6 Kbps modem for communication.

The licensing of WebTV to a number of consumer electronics manufacturers has increased the probability of WebTV becoming a de-facto standard for Internet access using a television. Users are able to browse the WWW, send and receive e-mail and perhaps most importantly, be able to conduct commercial transactions electronically by the use of a smart card.

Most telephone switches are connected to the rest of the telecommunications network via fibre-optic backbones that are able to transfer large amounts of digital information at very high speeds. This means that most traffic is transmitted in a digital form, even if it originated as an analogue voice signal.

2.4.2 ADSL

The Asymmetric Digital Subscriber Loop (ADSL) [105] takes advantage of the existing investment in PSTN infrastructure, namely the copper twisted pair which is connected to most premises in the western world. Using a sophisticated compression technique, ADSL is capable of supporting downstream (to the customer) data rates from 608 Kbps up to 6 Mbps and upstream data rates ranging from 9.6 to 944 Kbps [106]. The technology is sensitive to the distance between the telecommunications service provider's ADSL hub, and the customer premises equipment.

As indicated by its name, ADSL is asymmetric, allowing data to be transmitted at a greater downstream rate than upstream. It is common for domestic Internet users to have a greater need for downloading information (downstream) than uploading it (upstream). There is also a technical rationale behind the asymmetrical nature of ADSL since it facilitates the minimisation of cross talk between multiple wires that have been collected together into a cable.

2.4.3 Cable Modems

The distribution of Cable Television (CATV) takes place over a coaxial cable connected to the customer premises. The market penetration of CATV is smaller than that of the PSTN twisted pair, but the use of a cable modem on the CATV connection promises to ultimately provide consumers with a faster data service than ADSL. Being asymmetrical in nature, cable modems offer a downstream data rate up to 30 Mbps, and an upstream data rate of 2–3 Mbps. Cable modems use a shared medium, which means that multiple consumers must share the available bandwidth.

Cable modems have appeared in a number of test-beds conducted by companies including @Home, Time Warner Cable and BellSouth Corp. In the United States of America (USA), coaxial cable is laid past 97% of all households [102], with 65% of all households being CATV customers. In Australia, two individual HFC networks have been under construction since 1997 by Optus Telecommunications and Telstra Australia.

2.4.4 ISDN

The Integrated Services Digital Network (ISDN) is a purely digital service which operates over an Integrated Digital Network which is now the major part of the PSTN. It provides consumers with a 144 Kbps data connection that is split into 2x64 Kbps + 1x16 Kbps (2 Basic and 1 D) channel. In addition, it is rarely available outside major metropolitan areas. When compared with the current data rates offered by analog modems, ISDN offers a marginally increased data rate for residential consumers, but is expensive and can be difficult to set up [98].

ISDN has been the subject of standardisation since the mid 1980's and has failed to claim a significant share of the market in the USA, UK or Australia, however it is reasonably widespread in Germany.

2.4.5 Fibre to the Curb/Home

Fibre to the Curb (FTTC) and FTTH [103, 104] are options with the greatest data rate potential for consumers. In the installation of FTTH and FTTC, labour costs are the greatest, and this cost is the same regardless of the medium being installed. This means that when a distribution technology is installed in a new residential development, it is economically viable to install fibre rather than coaxial cable because it is the digging of the trench and laying of the fibre which is expensive, rather than the cost of the fibre itself. It is more difficult, however, to get suitably qualified fibre optic technicians than coaxial technicians.

Greenfields sites such as new housing developments are prime candidates for the installation of FTTH or FTTC, since it is possible to install the infrastructure in common access trenches when telecommunications infrastructure is installed.

2.5 Electronic Commerce

The field of electronic commerce is built upon a number of research areas. Of primary importance is the telecommunication connection that links consumers and service providers which must provide users with adequate response times. Also of significant importance is the existence of a secure electronic transaction protocol, which ensures the privacy and integrity of electronic transactions.

Having obtained good access and secure transactions, the final step is to match consumers with service providers. Presently, the most common method of matching consumers with service providers is the use of search engines on the Internet. As will be discussed, this technique does not satisfactorily scale globally, and does not ensure the correct matching of services to requests. In fact, many users find search engines to be inaccurate and frustrating since they typically provide many thousands of matches, many of which are not relevant to the query.

2.5.1 Smartcards

A smartcard is a plastic card of similar dimensions to a credit card, but with one major difference: the inclusion of an integrated circuit electronic chip which is capable of performing calculations and storing data. A significant advantage of smartcards for consumers is the ability to combine a number of functions into one smartcard [100]. Smartcards have a much higher level of security than traditional credit cards, which utilise insecure magnetic strip technology. This means that smartcards have the potential to hold money, personal medical information or even biometric security information. As secure electronic transaction (SET) protocols [101] become adopted, it is likely that smartcards will be used to perform safe monetary transactions online and provide users with an electronic wallet.

2.5.2 Entrepreneurial Activity

Perhaps more than any technology of the 20th century, the advent of E-Commerce opens the way for entrepreneurs to create new and diverse businesses. At present, E-Commerce generally takes the form of selling products rather than services. In a virtual enterprise [33], location is less relevant to the consumer, since it is possible for products to be delivered via courier from anywhere in the world. Retailers with unique products are able to market them over the Internet to a global audience.

As the community becomes more wired, the market will expand to include local goods and service providers, who do not aim for a global market. For example, local businesses may use the network infrastructure to provide online ordering for fast-food through user interaction with an online ordering form. The order is transmitted to the business and processed immediately. An example of such a service can be seen at many fast food franchises, where orders appear on a video display. By incorporating a network presence into the system, customers may order products as they do with a telephone, with the added marketing bonus for retailers of advertising special offers and new products.

In the future, E-Commerce will not be limited to the sale of goods but will include the provision of services also. In such a scenario, services providers in areas such as Real Estate, taxation preparation, insurance brokering, banking and tradespeople could benefit from having an online presence, where services, pricing and availability information is provided to customers.

With the online presence of many local businesses, there is a need for the advertising and location of services, similar to the telephone system's Yellow Pages which are an effective way of locating service provider's telephone numbers. However, the Yellow Pages do not provide automated searching facilities, nor do they allow a person to search for product or service specific information. Any solution to the service lookup problem must:

- be scalable to allow many services and resources to be included,
- operate on a global scale to accommodate global online retailers and
- be dynamic to allow for the addition of new services and the removal of old services.

2.6 Motivation for Trading Infrastructure

In the near future, it will be possible for residential consumers to access high speed data communications networks, thereby increasing the number and diversity of Internet users. It is easy to envisage a large proportion of households having access to a computer with Internet access, since it is possible for such connections to be made through the existing PSTN or CATV infrastructure.

Coupled with the emergence of user–friendly secure online electronic transactions and a large growth in the proportion of businesses with an Internet presence, the scene is set for hundreds of millions of customers to participate in electronic commerce and the creation of new services and resources. Transactions will be performed on a global scale, where consumers are able to shop electronically and are not restricted to local markets. In such an environment, it is imperative that a flexible and scalable service lookup mechanism is available.

The Trader is capable of providing such a lookup service with the advantage of being distributed over multiple nodes. This is an important advantage since it removes the performance and reliability limitations of centralised lookup services such as existing search engines. The distributed nature of the Trader allows trading domains to be customised according to criteria, which allows a flexible solution to local service advertising.

Distributed systems are much more complex than centralised systems since the complications of concurrency and imperfect communications channels may cause the system to deadlock. It is therefore of great importance to verify the correctness of these systems, thereby ensuring that they function as intended. It is with this in mind that we proceed to model and verify the Trader.

2.7 Summary

In this chapter, the concept of Community Networking has been introduced and some examples of large scale Smart City projects were presented. Before Community Networking can become widespread, users require access to a high speed data network. There are many different technologies that may be used to provide such a network and they vary in cost and installation time. However, distribution technology alone is not enough to foster electronic commerce, since there is a need for a global and distributed service location facility as part of the Information Infrastructure.

Chapter 3

Trading in Open Object–based Distributed Systems

In this chapter, background on Open Object–based Distributed Systems (OOBDS) [7] and the Trading function is presented. The standardisation of the Reference Model for Open Distributed Processing (RM–ODP) is discussed, along with the Object Management Group’s (OMG) [145] Common Object Request Broker Architecture (CORBA) specification [44].

3.1 Open Object–based Distributed Systems

An OOBDS is a distributed system that is composed of objects that interact via well–defined interfaces. The system is considered *Open* since it is possible for objects to co–operate in a heterogeneous environment where operating systems, network protocols and hardware platforms may be from different vendors. It is very difficult to create successful implementations of these systems without the use of appropriate standards. In the next sections, standardisation efforts by international bodies regarding the design and practical realisation of OOBDS is covered.

3.2 The Reference Model for Open Distributed Processing

The Reference Model for Open Distributed Processing [3] is a joint effort by three international standards bodies: the International Organisation for Standardisation (ISO), the International Electrotechnical Commission (IEC) and the International Telecommunications Union (ITU). RM–ODP aims to provide transparent sharing of resources and services over different architectures, networks and operating systems [18].

The RM–ODP is composed of the following four parts (described by Raymond [7]):

- Part 1 **Overview** [3]: Describes the motivation behind the RM–ODP standardisation, giving scope, justification of key concepts and an outline of the ODP Architecture.
- Part 2 **Foundations** [4]: Contains the definition of the concepts and analytical framework for the description of arbitrary Open Distributed Systems.
- Part 3 **Architecture** [5]: Specifies the required characteristics that qualify a distributed system as being *open*. These are the constraints to which ODP standards must conform.
- Part 4 **Architectural Semantics** [6]: Contains a formalisation of the ODP basic modelling concepts defined in Part 2 of the RM–ODP. The formalisation is achieved by interpreting each concept using the constructs of a number of formal description techniques (FDT's) including Z [138], Lotos [132], ESTELLE [133] and SDL'92 [134].

The RM–ODP has the following aims [3]:

- portability of applications across heterogeneous platforms,
- interworking between ODP systems (including data transfer and sharing of services/resources) and,
- distribution transparency (hiding the consequences of distribution).

In order to facilitate distribution transparency, the RM–ODP identifies a set of transparency properties [5]. They were summarised by Linington [8] where the transparencies have the following functionality:

- **access**: masks differences in data representation and invocation mechanisms allowing objects to interwork,
- **failure**: masks the failure and possible recovery of objects from the ODP system designer. This enables the designer to assume an ideal world in which such failures do not occur,
- **location**: masks out the use of specific location information when identifying and locating objects,
- **migration**: masks from an object the ability of a system to change that object's location,
- **relocation**: masks the relocation of an interface from other interfaces that are using it. This may occur when an object migrates and one of its interfaces is being used by another object,
- **replication**: masks the use of behaviourally equivalent objects that are used to support a single interface. This situation may be used to improve performance or reliability of a service provided at an interface,

- **persistence:** masks from an object the activation and de-activation of objects (i.e. when an object is stored in a persistent database) and
- **transaction:** masks the co-ordination between objects required to ensure consistency.

These transparencies can be achieved by utilising standard mechanisms, thereby leading to software re-use. For a given system, the designer indicates the interactions which are required and the transparency properties associated with them.

In order to remain platform-neutral and to ensure that it does not restrict implementations to specific technologies, the RM-ODP is an abstract description of components necessary for the creation of open distributed systems.

3.2.1 Object Model

RM-ODP uses an object model [3], where an object (shown in figure 3.1) is defined as an autonomous entity that has:

1. local state that is represented by data values,
2. operations that may be performed on the object's local state (Methods) and
3. threads : sub-processes which operate concurrently within the object.

When modelling an RM-ODP system, it is important to allow for the encapsulation of data (data hiding), provide well-defined interfaces for manipulation of data (through method invocation), and provide support for the concept of threads in an object's control flow.

An RM-ODP system is composed of a number of autonomous objects which interact to provide a higher level of system functionality. These objects are likely to undergo periods of maintenance and may fail at any time. In addition, the location of an object is not guaranteed to remain constant since it is possible for it to migrate as system-level resource policy dictates such as, for example, in the case of load balancing [115]. Thus, it is important that RM-ODP provides distribution transparency for designers of OOBDS.

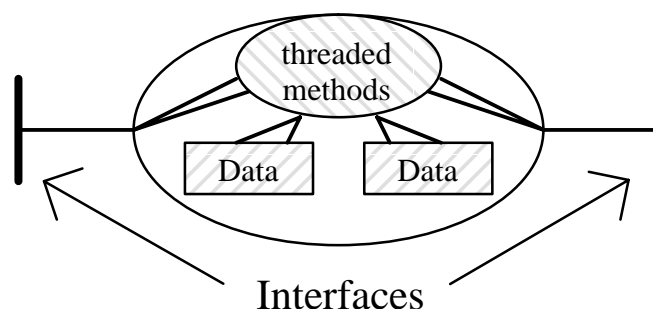


Fig. 3.1: The RM-ODP Object Model

3.2.2 Object Bindings

Before two objects can interact via their interfaces, a *binding* must be established [3] which creates an association between the interfaces. It is possible for a binding to be an object which

performs, for example, a Quality of Service (QoS) function. A binding may be shown *explicitly*, as depicted in figure 3.2, where objects A and B interact via interfaces a1 and b1 using an explicit binding object.

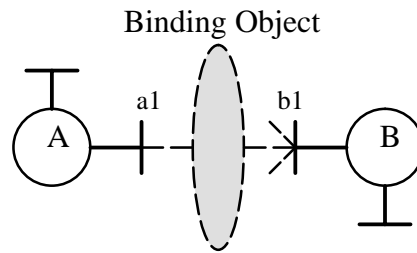


Fig. 3.2: Explicit Binding between objects

Alternatively, in the case of a simple client–server interaction, the binding may be considered *implicit* and not specifically included in the model, as illustrated in figure 3.3. Thus, a binding object is not required when objects interact using an implicit binding.

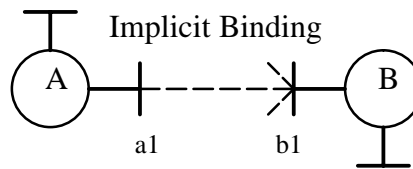


Fig. 3.3: Implicit Binding between objects

3.2.3 RM–ODP Viewpoints

RM–ODP systems are described using a number of different viewpoints as prescribed in Part 3 of the RM–ODP specification [5]. Viewpoints allow the system to be described from different perspectives and abstractions. The concepts, structures and rules for describing these viewpoints are specified in the RM–ODP documents, thereby allowing viewpoints to be described using a “language” for each viewpoint [7].

Using these viewpoints, a system may be described with an emphasis on information flow, computational entities or high–level corporate aims. One area of important research is consistency between viewpoints [49], which deals with the problem of ensuring that the five RM–ODP viewpoints provide a description of the same system and do not conflict with each other. The five viewpoints are described below.

3.2.3.1 Enterprise

The Enterprise viewpoint is used to describe the high–level organisational requirements of the system being designed. The description includes defining objects which may be active (entities) or passive (utilised by active objects). Communities of objects may be defined, thereby grouping together objects that are intended to achieve a specific purpose. The roles of objects in the system are described using policies that include permissions, prohibition and obligations [4]. Thus, the enterprise viewpoint provides an expression of purpose, policy and boundary for the system being designed [38].

3.2.3.2 Information

The information required by the ODP application is described in this viewpoint through the use of schemas. The schemas are used to describe the state and structure of an object. Three types of schemas are used:

- *static*: describes the state and structure of an object at a specific event;
- *invariant*: describes restrictions on the state and structure of the object and
- *dynamic*: defines the logic for modification of the object's state and structure.

Thus, the information viewpoint focuses on the information and information processing functions of the system being designed [38].

3.2.3.3 Computational

The computational viewpoint is a decomposition of the system into objects that interact via interfaces and is used to specify the functionality of the application. This description is object-based and uses the RM-ODP object model as described in section 3.2.1. A computational specification defines the objects in the system, the activities within those objects and the interactions that occur between objects [7]. Interfaces may use one of three interaction types : operational, stream or signal-oriented.

- operational interfaces are similar to Remote Procedure Calls and provide a client-server mechanism,
- stream-oriented interfaces provide a continuous stream of information that flows from the producer to the consumer,
- signal interfaces provide very low level communication actions. Examples include the OSI service primitives REQUEST, INDICATE, RESPONSE and CONFIRM.

3.2.3.4 Engineering

This viewpoint describes the distribution-oriented aspects of the system and is not concerned with the semantics of the application being described. The viewpoint focuses on the requirements for distribution such as transparency. The viewpoint contains objects (generally the same as in the computational viewpoint) and channels which correspond to a binding (interface connection) between objects. This viewpoint provides a description of the infrastructure required to support distributed processing in the system being designed [38].

3.2.3.5 Technology

This viewpoint concentrates on describing the choice of technologies for the implementation of an ODP application. The description includes information required for testing of the system and also, choices regarding:

- implementation language,

- middleware support,
- platform,
- operating system,
- protocol stack.

3.3 The Object Management Architecture

- The Object Management Architecture [46] (OMA) is composed of two models:
- An Object model that defines how heterogeneous distributed objects can be described. This model is very similar to the RM–ODP object model described in section 3.2.1.
- A Reference Model that characterises interactions between heterogeneous distributed objects. The main component of this model is the Object Request Broker (ORB) that provides a mechanism for objects to communicate with each other.

3.3.1 OMG

The Object Management Group, Inc. (OMG) is an international industry consortium established in 1989 to promote the theory and practice of object technology [1,47]. The OMG has more than 500 members, including large and influential companies from the computing and telecommunications sectors, software vendors, developers and users. As a significant consequence of OMG activities, a series of industry guidelines and specifications for Object-based software has been developed. These documents provide a common framework which allows conformant systems to operate in a heterogeneous environment encompassing differing hardware and software platforms.

The OMG aims to “foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based” [44]. It attempts to define the facilities required to allow distributed object oriented systems to be created. The work of OMG is aligned with ODP standards and there is a great deal of communication between the two groups. This is evident in the technical alignment between the ODP Trader [16] and the Common Object Request Broker Architecture (CORBA) Trading Object Service [45].

3.3.2 Common Object Request Broker Architecture

CORBA is a solution to the problem of allowing heterogeneous distributed objects to work together, sharing services and resources to create a distributed system. An important feature of CORBA is that implementation of an object is separate from the definition of its interfaces.

The diagram in figure 3.4 [60] is based upon a diagram first presented in [59]. It shows the CORBA architecture and illustrates the relationship between clients, Interface Definition Language (IDL) stubs, the Object Request Broker (ORB) core, IDL skeletons and object implementations (servers). These concepts will be explained later in this section.

3.3.2.1 The Object Request Broker

The ORB enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in heterogeneous environments [44].

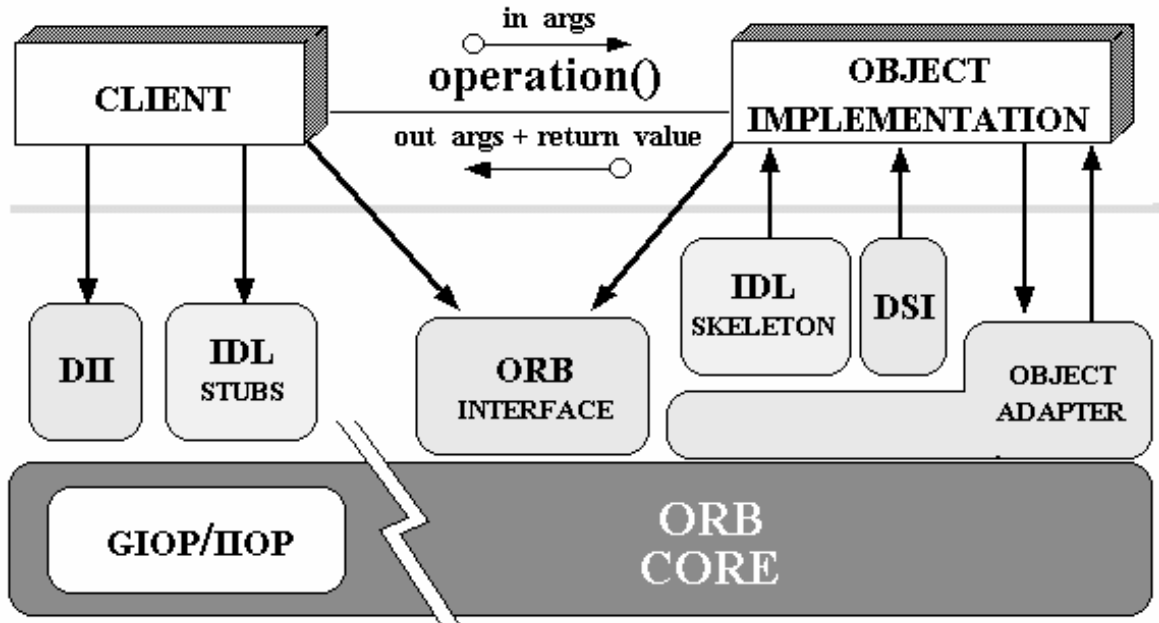


Fig. 3.4: Common Object Request Broker Architecture

A client is able to invoke a method on other objects in the system using the ORB. Whether a method is invoked on a server locally, or over a network is transparent to the client. It is the role of the ORB to provide a “common bus” which connects all objects in the system and provides a common medium, thereby allowing distributed objects to interact.

In figure 3.4, the client is invoking an operation on the object implementation (server). It passes arguments (args) and accepts a return value and the operation arguments in response to the operation invocation.

3.3.2.2 Interface Definition Language

Using CORBA’s Interface Definition Language (IDL) [44], an object’s interface may be defined in a manner that is independent of the application’s implementation language. The interface definition in IDL is compiled to create a language–specific stub and skeleton, used by the client and server respectively. The client may then use the stub to invoke operations on the server. The stub is used as an interface between the client implementation language and the ORB, providing the client with a simple mechanism for invoking methods on remote objects. The skeleton is used by servers as an interface between the ORB and their implementation language. The skeleton and the stub may be dynamic which allows objects to invoke methods on objects that

they did not know the interface of at compile time. These interfaces are known as the Dynamic Invocation Interface (DII) and the Dynamic Skeleton Interface (DSI) respectively.

3.3.2.3 Application Domains

CORBA is widely used for the implementation of heterogeneous distributed systems in the fields of telecommunications, medicine and finance [59]. In early 1996, the OMG was re-organised into two committees: the Domain Technical Committee (DTC) and the Platform Technical Committee (PTC) [59]. The DTC focuses on domain-specific technologies and the PTC concentrates on more general domain independent technologies. The DTC has recently issued some Request for Proposals (RFPs) in the Medical, Telecommunications and Business domains, with an aim to provide more integrated support for applications in these domains.

3.3.2.4 Inter-Orb Protocol

Initially, CORBA v1.1 [43] did not specify a protocol to allow ORBs from different vendors to communicate. However, in v2.0 [44], the OMG defined the Internet Inter-Orb Protocol (IIOP). This protocol allows distributed objects associated with compliant ORBs to interact with each other, greatly increasing the power of CORBA. A Java CORBA 2.0 compliant ORB is included in Netscape's Navigator 4.0, promising to bring CORBA to millions of desktops [58]. It is possible for an object to make use of services provided by an object connected to an ORB implemented by a different vendor. This raises the problem of locating services across the boundaries of ORBs which can be addressed using an object known as a Trader [16].

3.4 The Trading Function

In a distributed system, locating an appropriate server should ideally be performed dynamically at run-time to allow for the introduction of new services and service providers. As an important part of the RM-ODP standard, the Trading function [16] allows a service consumer (importer) to be matched with a service provider (exporter) using a trusted third party.

3.4.1 Basic Trading Entities

There are three main entities involved in the trading function, namely, the Exporter, Importer and the Trader itself. The sequence of interactions between these entities is shown in Figure 3.5 [2].

1. **Service Export:** The Trader receives a service *export* from the Exporter. This request includes a description of the service, a location at which the service may be accessed, properties of the service and optional selection criteria. The service interface's type and associated properties are checked with a Type Repository to ensure legality. Details of the exported service are then stored in a database.
2. **Import Request:** The Trader receives a service *import* request from a client. This request includes the type of service required and a list of desired attributes. The Trader checks with

the Type Repository *Infrastructure Object* to ensure that the requested service and properties are legal.

3. **Import Reply:** The Trader searches its database of exported services and returns any successful matches after applying selection criteria if included in the Importer’s Query.
4. **Service Invocation:** If a match is found, the Importer contacts the Exporter independently of the Trader to utilise the service.
5. **Service Reply:** The Exporter replies to the Importer’s service request.

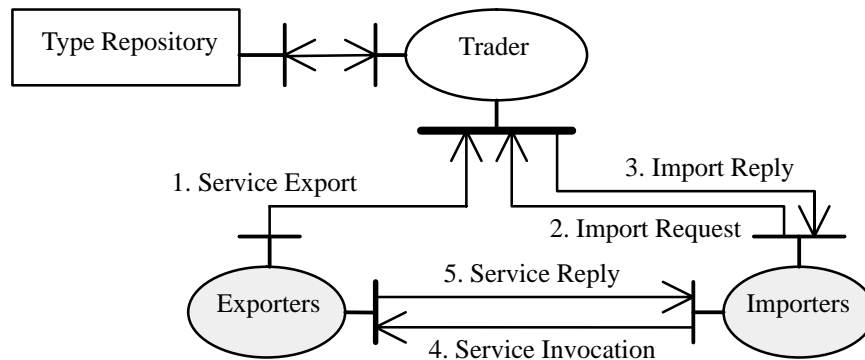


Fig. 3.5: Sequence of Interactions between Exporter, Trader, and Importer

A client, that is required to locate a specific type of service, may submit a request to the Trader, thereby acting as an Importer. The request contains the type of service required, properties associated with the service and optional selection criteria which may be used by the Trader to determine the “best” service offer from a set of matching offers.

The Trader searches its database of exported offers and locates a set of matching offers. This set of offers may be further reduced by applying the optional selection criteria and can be ordered using a preference parameter supplied with the Query. The Trader then returns the “best” offer(s) to the Importer which may then utilise the service.

An example of a service that may be exported is an e-mail application, that allows software entities to e-mail data. This service could be utilised by a migrating software entity that must e-mail the results of searching activities to its owner. Since the entity does not have a fixed location, it must locate a suitable server object as it moves. By utilising the Trading service, the entity is able to locate a server by specifying the required service type and important service attributes such as for example, the ability to send binary files.

3.4.2 Service Types

A service Type is defined in the Trading Standard in section 8.2.3.1.

“Associated with each traded service is a service type which represents the information needed to describe a service. It comprises:

- an interface type which defines the computational signature of the service interface,*
- and*

– zero or more named property types. Typically these represent behavioural , non-functional, and non-computational aspects that are not captured by the computational signature.”

“The property type defines the property value type, whether a property is mandatory, and whether a property is read-only. That is, associated with a property type is the triple of <name, type, mode>, where the modes are:

```
enum PropertyMode {  
    PROP_NORMAL, PROP_READONLY,  
    PROP_MANDATORY, PROP_MANDATORY_READONLY  
};
```

Thus, a service property type has an associated, name, value type and mode. A property with mode NORMAL is optional, one with mode MANDATORY must have a value, and one with READ_ONLY cannot have its value modified later by the Exporter. For each Trader, Service Types are maintained by the Type Repository object as discussed in section 3.4.1.

3.4.3 Service Offers

A service Type is defined in the Trading Standard in section 8.2.5.

“A service offer is the information asserted by an exporter about the service it is advertising. It contains:

- the service type name,
- a reference to the interface that provides the service, and
- zero or more named property values for the service.”

As discussed in section 3.4.2, the Exporter must provide values for all MANDATORY properties in the associated service type. An example `printer` service offer is given below:

```
(printer, lp0, [ps_capable = TRUE (PROP_MANDATORY),  
               page_size = A4 (PROP_NORMAL)])
```

This `printer` service is available at the `lp0` interface, is capable of interpreting postscript (a static property) and uses A4 size paper (a possibly dynamic property).

3.4.4 Trading Interfaces

The Trader provides a set of Functional and Abstract interfaces which are used by other objects in the system (such as the Importer and Exporter). They provide a well-defined mechanism for interacting with the Trader when using its services or providing administration of the Trader itself.

3.4.4.1 Functional Interfaces

The functional interfaces provide an access mechanism to the services provided by the Trader.

- **Register:** allows Exporters to *export* (advertise) services to the Trader. It also provides methods to *modify*, *withdraw* and *describe* offers that have previously been exported to the Trader. This interface can also be used to *resolve* a Trader's name which results in a reference to its Register interface.
- **Lookup:** allows Importers to *query* the Trader's offer space for services that match their requirements. Optional selection criteria may be included in the query using a constraint recipe language defined in Annex C of the Trading Standard [16].
- **Admin:** utilised by a Trading Administrator object to set Trading policy values shown in the Abstract interface and to list all of the Trader's offers.
- **Link:** utilised by an Administrator object to *add*, *remove*, *describe* and *modify* the Trader's links with other Traders.
- **Proxy:** allows an Exporter to *proxy_export* an offer. A proxy offer contains an associated service type and service parameters, but does not indicate the interface at which the service may be obtained. Instead, it includes a reference to an interface upon which the Trader performs a secondary Query operation in order to locate the interface at which the service is provided. Thus, the Trader is able to determine the interface at which the service is provided dynamically at run-time.
- **Dynamic Property Evaluation:** used by offers which need to utilise dynamic values for some of their properties. An example of this is a printing service, where the queue length is a dynamic value.

3.4.4.2 Abstract Interfaces

The Abstract Interface is provided by a Trader with the following Read-only attributes which can be tested by clients:

- *TraderComponents* which provide interface references to all the Trader's interfaces. This allows clients to obtain a reference to any of the Trader's interfaces, via an existing reference to one of the Trader's interfaces.
- *SupportAttributes* which indicate the optional functionality supported by the Trader. These attributes include boolean values for:
 - modifiable properties
(does the Trader allow exported offer properties to be modified?)

- `dynamic_properties`,
(does the Trader allow exported offer property values to be dynamic, thereby requiring a run-time value lookup for each Query?)
- proxy offers,
(does the Trader allow exported offers to be Proxy Offers?)
- and an interface reference to the Type Repository object used by the Trader.
- *ImportAttributes* which indicate the Trader's policy with respect to Query operations. These attributes are defined in section 3.4.8.1.
- *Link Attributes* which indicate the maximum `link_follow_policy` to be allowed for new links created by the Trader (as discussed in section 3.4.6.1).

3.4.5 Iterator Interfaces

The Offer and Offer Id Iterator interfaces identified in the Trading Standard [16] provide a mechanism for extracting results from operations using multiple method invocations which return a single value, rather than using a single method invocation which returns a set of values.

3.4.5.1 Offer Iterator

The Offer Iterator interface allows Importers to extract a single matching offer at a time from Queries which result in multiple matching offers, rather than obtaining them all at once. For example, if a Query results in five matching offers, five method invocations on the Offer Iterator interface can be used to retrieve the matching offers, one at a time. This facility may be useful for an Importer who only requires the “best” service offer, rather than all matching service offers.

3.4.5.2 Offer Id Iterator

The Offer Id Iterator interface is used in conjunction with the `list_proxies()` and `list_offers()` methods available at the Admin Interface. It allows the Trading Administrator to extract an arbitrary number of matching offer identifiers from the set of offer identifiers returned by the methods.

3.4.6 Interworking Traders

Since the total number of offers available in a global distributed system would be enormous, Traders were designed to be scalable [2]. Traders are said to *interwork* when multiple autonomous, possibly heterogeneous Traders cooperate to provide an increased range of matching services. This is facilitated by the creation of *Links* which describe the knowledge one Trader has of another and with whom it may interwork (see section 3.4.6.1). Using these links, a Trader may act on behalf of one of its clients as the client of another Trader. This propagates Queries to linked Traders.

Traders may be bi-directionally linked to each other by having two uni-directional links in both directions between the Traders. Interworking has the effect of:

- increasing the number of clients that have access to an Exporter's offers
- increasing the number of potential offers available to Importers.

Traders cooperate using an interworking protocol [16] that must be flexible enough to allow arbitrary Trader interconnection topologies and support processing of duplicated Queries. This is accomplished by the use of Import, Link and Trader policies as discussed in section 3.4.8.

Traders use a `hop_count` to limit the propagation of Queries along links, as stated in the Trading Standard [16], section 8.2.8.1 Link Traversal Control, page 33.

“To ensure that a search does not enter into an infinite loop, a hop_count is used to limit the depth of links to propagate the search. The hop_count is decremented by one before propagating a query to other traders. The search propagation terminates at the trader when the hop_count reaches zero.”

3.4.6.1 Links

The Trader uses links to other Traders for interworking. A link contains the following information [16]:

- unique Link name,
- the Lookup Interface of the Trader being linked to,
- the Register Interface of the Trader being linked to,
- a default link follow behaviour policy,
- limiting link follow behaviour policy.

A link follow behaviour policy can take one of the following values:

- **local_only:** restricts the Query to being processed by the local Trader only. (It is not permitted to forward the Query to linked Traders.)
- **if_no_local:** limits the Trader to local processing except in the case where there are no local matches to the Query. In this situation, the Trader is permitted to forward the Query to linked Traders.
- **always:** permits the Trader to forward Queries to linked Traders.

These behaviour policies are listed from weakest to strongest and can be used to restrict the propagation of requests to linked Traders. This ordering makes it possible to calculate the smallest or minimum behaviour from a set of behaviours. This is important when multiple policies are compared, as described in section 3.4.8.4. For example,

```
min(always, if_no_local, local_only) = local_only
min(always, if_no_local, always) = if_no_local
```

The link *name* can be used by the Trader to identify a link when it requires administration. The Lookup and Register interfaces are references to the respective interfaces of the Trader being linked to. The default follow behaviour policy is used when the importer's Query does not specify a follow behaviour. The limiting follow behaviour is the strongest follow policy that the link will allow when forwarding queries to the Trader associated with the link.

3.4.7 Offer Space

In figure 3.6, the total trading Offer Space is the set of offers that have been exported to all of the Traders that are interworking. From this set of offers, there is a sub-set of offers that matches the parameters of the Importer's Query operation. Another sub-set of the total offer space is the set of offers that are able to be searched due to the Query policy restrictions (Trader, Import and Link) described in section 3.4.8. The intersection of these two sub-sets is the set of offers that match the requested service parameters and are returned to the Importer as the result of a Query.

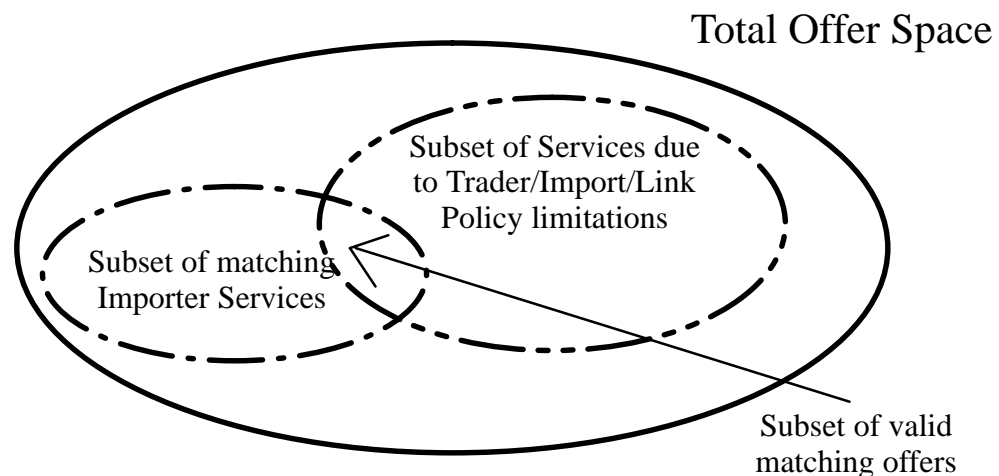


Fig. 3.6: The Trading Offer Space

3.4.8 Trading Policies

The Trading standard allows arbitrarily complex interworking relationships to be created and thus, requires a flexible mechanism for controlling the propagation of requests. This is accomplished by using three policies [16]: Trader Policy, Import Policy and Link Policy.

3.4.8.1 Trader Policy

The Trader policy is used to control the global behaviour of the Trader. There are 5 policy settings in the Trader policy:

- `search_cardinality`: defines the number of offers to be searched,
- `match_cardinality`: defines the number of matched offers to be ordered,

- `return_cardinality`: defines the number of ordered offers to be returned,
- `hop_count`: defines the number of times a Query should be forwarded to other Traders,
- `follow_policy`: defines the Trader's follow behaviour which is applied to all Queries that it services.

Each of these settings are included in the Trader's policy with default and maximum values. Default values for a setting are used when a value is not supplied with a Query operation and maximum values are applied when unifying multiple policies, as described in section 3.4.8.4.

3.4.8.2 Import Policy

An Import policy is associated with each Query and is used to control the behaviour of the Query from the Importer's perspective. In the Trading standard [16], the Import policy is sometimes referred to as the Query policy. It contains the following information:

- `search, match and return cardinality`: defines the maximum number of Offers to be searched, matched and ordered, and returned for the Query. These elements perform the same function in the Import policy as they do in the Trader policy described in section 3.4.8.1.
- `exact_match_type`: boolean value which indicates whether the service type of matching offers must be exactly the same as that requested by the Query.
- `hop_count`: defines the maximum number of times that the Query is to be forwarded to linked Traders as defined in section 3.4.8.1.
- `link_follow_rule`: defines the follow behaviour to be used when determining whether the Query is to be forwarded to linked Traders.
- `starting_trader`: used by the Importer to specify the Trader at which the Query begins. Traders are expected to forward the Query down all links which are specified in the name which dictates a path to the Trader through a series of link names. This format is used since it allows a Query to follow links to the Trader which is to commence the Query. However, this scenario fails if links are destroyed since the name (series of links to follow) is no longer valid.
- `request_id`: is a unique identifier that identifies the originator of the Query and also the Query itself. According to the Trading Standard [16], a Request_Id is generated for each Query using the Trader-specific `request_id_stem` attribute as a basis. It is utilised by the Trader for detecting duplicate Queries, as described in section 6.5.2.

3.4.8.3 Link Policy

A Link policy is created for each link and contains elements which define follow rules for a given link. These elements are:

- `default_pass_on_follow_rule`: passed to the next Trader(s) if the Importer does not specify a `link_follow_rule`,
- `limiting_follow_rule`: defines an upper bound on the `follow_rule` for a given link.

3.4.8.4 Resolving Policies

When determining whether to forward a Query, the Trader performs a unification on the three policies (Trader, Link and Import). It utilises the following logic which is specified in the Trading standard, section 8.2.7.6 entitled “Link Follow Behaviour” [16].

```
if the importer specified a link_follow_rule policy
    min(trader.max_follow_policy, link.limiting_follow_rule,
        query.link_follow_rule)
else
    min(trader.max_follow_policy, link.limiting_follow_rule,
        trader.def_follow_policy)
```

This means that when the importer does not specify a `link_follow_rule`, then the Trader’s `def(ault)_follow_policy` is substituted. The *smallest* (or *minimum*) of the `follow_policies` is used as a `unified_follow_policy` which dictates whether a given Query is forwarded to other traders using a specific link. It can be seen that the Trader’s `max_follow_policy` is a limiting factor when determining the scope of a forwarded request.

Having decided that a Query is to be forwarded down a link, then the `link_follow_rule` policy that is to be passed down the link must be determined. This is accomplished using the following logic:

```
if the importer specified a link_follow_rule policy
    pass on min(trader.max_follow_policy,
                link_limiting_follow_rule, query.link_follow_rule)
else
    pass on min(trader.max_follow_policy,
                link.default_follow_rule)
```

This means that if the importer specified a `link_follow_rule`, then the rule passed on with the Query to linked Traders is the same as the `unified_follow_policy` determined initially. If the Query did not specify a `link_follow_rule`, then the pass on policy value is the link’s default pass on value, limited by the trader’s maximum `follow_policy`.

The use of default values means that it is possible to calculate a unified policy value even when the Query does not contain values for some of the follow rules in its Import policy.

Using these policy values, it is easy to customise the scope of a Query as it is propagated to linked traders. This may be accomplished on a Trader-wide, per Query or per Link basis,

depending upon circumstances. For example, the Trader may decide to stop interworking with another Trader temporarily and thus, would not want to remove a link completely. In this case, it can set the link's `limiting_follow_rule` to `local_only`, thereby ensuring that the link is not followed. For the same effect on all links, the Trader can set `max_follow_policy` to `local_only`, thereby limiting all links to local searching only.

3.4.8.5 Example of Query Propagation

An example of how a Query is propagated between Traders along links is shown in figure 3.7 which is based upon a diagram in the Trading Standard [16]. A Query from an Importer is received by Trader 1 (**T1**) with a `hop_count` value of 3. **T1**'s `default_follow_policy=if_no_local`, its `max_follow_policy=always` and its `max_hop_count=1`. Since the Query did not specify a `link_follow_policy`, the Trader's `default_follow_policy` is used when determining a unified `follow_policy`, along with the link's `limiting_follow_rule` and the Trader's `max_follow_policy`.

```
unified follow_policy =
    min(trader.max_follow_policy, trader.def_follow_policy, query.link_follow_rule)
```

The result of this function is a `link_follow_rule` equal to `if_no_local`. Assuming that **T1** does not obtain a matching offer, the Query is forwarded to **T2** using this `link_follow_rule` and a `hop_count` parameter equal to 1 since it is limited by **T1**'s `max_hop_count`.

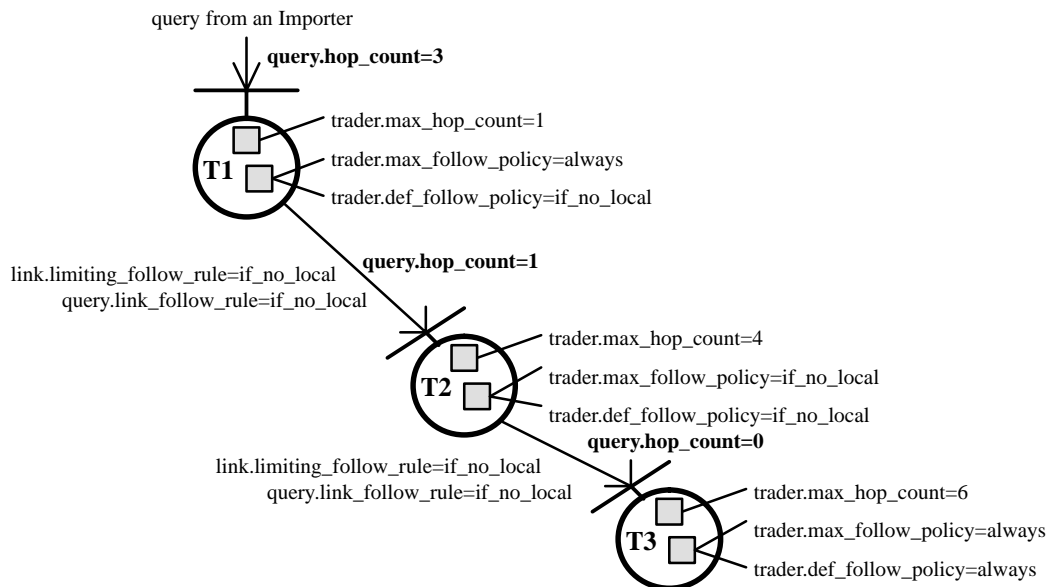


Fig. 3.7: Example of unifying policies

Given that **T2** does not obtain a matching offer, the Query is forwarded to **T3** since the unified `follow_policy` equals `if_no_local`. **T2**'s `max_hop_count` policy value is greater than the Query's `hop_count` value and thus, does not limit the `hop_count` value passed on with the Query to **T3** (which is simply decremented by 1 to a value of 0). The Query is not propagated further at **T3** since the `hop_count` equals 0 and there are no links to follow. **T3**

returns its search result to **T2**, which returns its result to **T1** which in turn returns the final search result to the Importer.

3.5 OMG's Object Trading Service

Originally, most of the work in the area of Service/Resource Trading was undertaken by the ISO/IEC JTC1/SC21/WG7 standardisation group. When the standardisation reached the Committee Draft (CD) stage, it was used as the basis for a modified Trading specification that was adopted by the OMG and set in concrete as a CORBA Common Object Service (COS) [44]. The RM-ODP Trading specification was later brought into technical alignment with the CORBA Trading specification, with CORBA-specific sections removed in order to retain a truly generic Trading specification.

3.5.1 OMG's Object Trading Service

In 1996, the OMG published Request for Proposal (RFP) 5, Trading Object Service. This service has similar requirements to the ODP Trader and has been included in CORBA 2.0 as a Common Object Service (COS). The Trading Object Service is identified as an important infrastructure element used by objects to bootstrap themselves. In response to RFP5, a joint submission [45] was presented to the OMG by the following seven organisations:

- AT&T/Lucent Technologies
- Cooperative Research Centre for Distributed Systems Technology (DSTC)
- Digital Equipment Corporation
- Hewlett-Packard Company
- International Computers Limited
- Nortel Limited
- Novell Inc.

The specification was adopted by the OMG in October 1996 and differed from the ODP Trader specification [14] in a number of major areas such as the Interworking protocol (discussed in sections 3.4.6 and 3.4.8.5) and the interfaces presented to clients. The specification also included a sample description of an Object Type Repository. The specification is not generic like the RM-ODP specification, but tailored directly for a CORBA-based implementation. As a result of the combined submission, the ODP DIS for Trading was edited to provide technical alignment between the two specifications [16].

With the addition of the IIOP, it is possible for Objects to be separated by large distances and be able to utilise each other's services using CORBA and high-speed networks. In such an

environment, an Object Trading Service is necessary to allow objects to locate each other on a global scale. A commercial CORBA Trader has been developed by DSTC in Australia [22].

3.6 ANSAware Trader

The ANSAware distributed systems platform [19] provides programmers with a location-independent object model. As with the RM-ODP object model (section 3.2.1), objects are encapsulated and interact with each other via operational interfaces. There is also a Trader which can be used for locating objects [20].

The ANSAware Trader does not conform to the ODP Trading Specification. It provides a mechanism for Interworking which mandates a hierarchical structure [53], where all traders must have a reference to a master Trader. This does not allow unrestricted interworking topologies to be created and raises reliability concerns since the trading system relies upon the master trader to be available. Despite these limitations, the ANSAware Trader was the first implementation of a Trader and has been used in a number of research projects [36,53,54].

3.7 Discussion

The RM-ODP Trader is an important infrastructure object which is required for the creation of dynamic OOBDS. By utilising a Trader, objects are able to locate services at run-time which results in increased flexibility and improved availability since it becomes possible for objects to locate new services when existing service providers become unavailable. In the case of distributed systems, this is very important since objects can move or become unavailable at any time.

The RM-ODP Trader standardisation has been adopted by the OMG's CORBA 2.0 specification as a Common Object Service. This has the effect of ensuring that there will be many Trader implementations that which will need to inter-operate using the IIOP. With the prospect of a large number of Traders operating and interworking, it is important to verify the correctness of the Trading standard.

For example, it is important to verify that when Traders are interworking, they do not become deadlocked and that the entire Offer Space of Interworking Traders is effectively traversed. In order to verify that the Trading standard's specification of Interworking Traders is correct, there is a need for a formal, verifiable model of the Trader. This model should be extensible, in order to allow modifications as the standard changes, provide the capability of creating multiple instances of Traders to model Interworking, and provide a "readable" specification, thereby providing a basis for implementation.

Chapter 4

Formal Models of the Trader

This Chapter provides the reader with background information on existing formal models of the ODP Trader. It briefly discusses some examples of using formal description techniques (FDT's) to model the Trader, including Message Sequence Charts [135], SDL'92 [134], Lotos [132], Z [138] and Object-Z [139].

As discussed in section 3.2.3, the RM-ODP uses five viewpoints to describe a system. There have been a number of attempts to model the Trader from different viewpoints [5]. Depending upon the FDT used, various limitations of the model/specification are evident. This chapter will give an overview of existing models/specifications of the Trader and where appropriate, point out areas for improvement.

4.1 RM-ODP Architectural Semantics

In part 4 of the RM-ODP standard [6], an Architectural Semantics is presented. This entails describing the RM-ODP architecture using a set of formal description techniques, including Z, SDL92, LOTOS and ESTELLE. According to Turner [39], these languages have been used because they are widely known or have been standardised. Turner also notes that political issues may prohibit the inclusion of new FDTs (such as Object-Z [139]) in the Architectural Semantics standardisation process, along with the fact that the standardisation work is already well progressed.

The Architectural Semantics provides a mapping from RM-ODP concepts to the appropriate constructs in the FDTs being considered. This provides system designers with a basis for specifying ODP systems since they can utilise the suggested constructs when modelling the system of interest.

Thus, the architectural semantics are intended to be an aid to the formal specification of ODP systems since the fundamental concepts such as object, interface and method invocation have

been mapped to specific representations in the FDTs. A thorough discussion of the Architectural Semantics work is given in [38].

4.2 SDL '92 Trader Specification

In this section, two examples of specifying the Trader using SDL'92 are examined.

4.2.1 Draft Trading Standard Annexes

In an early draft of the ODP Trading Function Specification [10], Annexes F and G [11, 12] contain an interface and behavioural specification of the ODP Trader using SDL'92 [134]. The specification includes the notion of a Trading Service Interface (TSI) and a Trading Management Interface (TMI), both of which were not included in recent drafts of the standard [14,16].

Annex G uses interface signatures contained within Annex F to provide a specification of the Trader's behaviour. The Interface signature of Annex F is a textual description of the data types and interfaces that the Trader provides.

Annex H [13] contains an incomplete specification of a *concrete* Trader and its application to a distributed Travel Agency. In this context, concrete refers to an instantiation of the object templates defined in Annex F. In the example, Traders are linked to allow interworking, although the protocol used for interworking is based upon an old draft standard [10] which does not take into account the import policy structure defined in the most recent Trading standard [16].

The specification scenario in Annex H includes three Traders, three Importers, three Exporters and also:

- a Security Policy Maker: used by the Traders to provide a security enforcing function,
- an Administrator : manages the Traders and links between them,
- a Type Management Object : ensures that *types* are valid (i.e. interface and service types)

Each of these objects are *virtual* since their functionality had not been defined in the standard. The objects are utilised by each of the three Traders, using a separate interface for each object interaction. Thus, the top-level model of the system details a mesh of connections between each of the Traders and the three shared objects.

The concrete specification in Annex H is incomplete and thus, does not provide details of the final application of the specifications defined in Annexes F and G. However, it does provide a model of the Trader which shows it using simple interworking and sharing the services of other objects in the system.

4.2.2 Fischer, Prinz and Vogel

In [24], the Enterprise, Information, Computational and Engineering viewpoints are specified using SDL'92. The specification utilises *virtual* transitions which allow the functionality of the transition to be refined at a later date and includes the following processes (or objects):

- **Manager** process which accepts and processes Queries,
- **Authentication** process which performs authentication of the Query's sender,
- **Database** process which maintains offers.

The Trader's behaviour is specified at quite a high level, where the mechanics of offer matching has not been included. Offers are received by Manager Process and the sender is authenticated by the Authentication process. If the sender is authorised, then the request is processed by Database. No mention is made of required behaviour if the authentication fails. The Database process is defined as virtual, leaving the specification of its behaviour for further refinement.

The specification is extended to define a *FederatedTrader* which is able to forward requests to other Traders. It **only** does this when it is unable to match a request locally using the Database object and forwards the request to **all** linked Traders. The final specification describes a *DistributedTrader* which is derived from the *FederatedTrader*, where the Database process is no longer centralised and is instead, a database manager that communicates with several database processes.

4.2.3 Discussion

These models are based upon an old version of the Trading standard which detracts from their usefulness for analysing the Trader's interworking protocol. However, the SDL'92 formalism does allow for high-level specification using the virtual transition construct which is a useful feature since it allows structure to be defined without extending into low-level details.

4.3 Lotos Trader Specifications

Sinnott et.al. [38] created a specification of the ODP Trader which includes descriptions of the Trader from the Enterprise, Information and Computational viewpoints. These specifications utilise aspects of the Formal Semantics section of the RM-ODP standard [6] which concerns the mapping of RM-ODP concepts into FDTs such as Lotos.

Using the mappings as a guideline, the authors have created a specification of the Trader using Lotos to specify three of the Trader's viewpoints. In the Enterprise viewpoint description, Lotos is used to specify some "high-level policy statements that govern the activities of the trading enterprise". The information viewpoint only considers a small sub-set of the information items that are required for Trading (service offers and the Trader itself). Matching criteria are modelled, but only in a very limited manner. Ordering of matching offers and interworking with

other Traders have not been included in the model. In the computational viewpoint, the Trader's operational interfaces have been considered (for importing and exporting of offers). Communication between entities uses the RM-ODP interaction model [3], where entities (importer and exporter) may *request* a service from the Trader which then gives a *response*. This model appears to include many important aspects of Trading, although the authors acknowledge that some features have not been included due to "complexity" issues.

In [24] a partial LOTOS specification of the Trader's computational and engineering viewpoints is presented. Much of the detail associated with the specification has been omitted which makes it difficult to fully appreciate the model. The model is very high-level and places more emphasis on identifying the objects present in the system and their methods, rather than specifying their behaviour.

4.4 Z Trader Specification

According to Sinnott and Turner [41], Z is most appropriate for modelling ODP systems in the Enterprise and Information viewpoints, rather than the Computational viewpoint. This is attributable to the fact that it offers a high level of abstraction but does not support multiple object instantiation and interaction, or encapsulation of objects. In the Enterprise viewpoint, policy rules for the system can be specified, thereby providing invariants for the entire system [41].

Fischer et. al. [24] have "partially" specified the Trader's Information viewpoint using Z. In the specification, the Trader was approached as an Abstract Data Type (ADT). The specification includes descriptions of service Exporting and Searching, but does not include any parallelism or concurrency.

A Z specification can be daunting to read for engineers who do not have a strong background in set theory and discrete mathematics. Its syntax and typeface requirements do not lend the FDT to widespread industry use since it is seen to be overly abstract and cryptic. Its inability to provide simulation of specifications also reduces its usefulness for engineers who require a facility for testing their designs and specifications using graphical simulations.

4.5 Object-Z Trader Specification

A specification combining the information and computational viewpoints of the Trader was created by Dong and Duke [42] using Object-Z. Object-Z is an object oriented variant of the Z FDT and includes temporal logic operators that allow temporal ordering and concurrency constraints to be expressed [24]. The specification provides a very high-level view of the Trader, and "implementation aspects of the Trader are not covered". It is a specification of "what needs to be done" and is therefore generic and avoids becoming implementation specific.

There is no attempt to model the mechanics of how the Trader operates, rather it provides a specification of objects in the system and data which they may contain. The specification is also

concerned with defining a low-level mechanism (a generic set class) that describes a collection of matching offers. The need to develop such low-level building blocks detracts from the readability of the specification, since the specification becomes cluttered with details associated with basic modelling functions which should ideally be part of the specification formalism itself.

The specification makes no attempt to include the “information matching required to pair importer requests to exporter services”. Instead, abstract data types are defined, with their internal structure remaining undefined. This means that the specification does not prescribe how the matching is to be done which is important for a high-level specification.

When specifying the links between Traders used for interworking, the model assumes that all links are symmetric. This assumption simplifies the specification but is unrealistic, since there is absolutely no guarantee that Traders will be bi-directionally linked. The problem of infinite cycling of forwarded requests within an Interworking Trader system has not been addressed in this specification. Since the model is based upon a very early working Document of the ODP Trader [17], it does not contain the recent interworking protocol, or the new external interfaces provided by the Trader (Admin, Proxy).

This example is a high-level specification which does not model the information or computational viewpoints of the Trader to any depth and cannot be used for simulations or for formal proof of functionality. It is more of a requirements specification based upon an old Trading Standard [10]. The model has limited usefulness since it contains a number of omissions (matching of offers) and assumptions (bi-directional links between Traders).

4.6 Message Sequence Chart Trader Specification

This specification of the Trader Import Operation is included in Annex B of the 1994 Draft ODP Trading Standard [10]. MSCs have been the topic of standardisation [137] and can be considered the synthesis of concepts taken from process algebra, Petri Nets and object-oriented modelling [140]. Since this specification only considered the Import Operation, it can only be used as a pointer to the technique’s usefulness as a specification technique for the ODP Trader.

The description makes use of generic placeholder objects including:

- Security Policy Maker,
- Search Policy Maker,
- Type Manager,
- LinkSpace Object,
- OfferSpace Object.

By using these objects, the system is described as a group of cooperating objects which communicate through message passing. As stated in section 3.1, this model is well accepted as the basis for Open Object-based systems.

The specification successfully describes the messages which are passed between objects in the system and can accommodate concurrent message passing. This is evidenced when linked Traders are queried in parallel. The graphical representation is good at illustrating message flow and sequence for a limited number of scenarios, but does not include any information regarding data/parameters required in the message interactions (OMG IDL can provide such a description). In addition, there are few tools for performing simulations/verification of the specification. This is a limitation since the problems associated with distribution/concurrency such as deadlock and livelock cannot be investigated. MSCs appear to be a useful semi-formal technique for describing a message passing system in a readable graphical form, but they fail to provide a strong semantic basis for formal analysis and wide-spread tool support.

4.7 Comparison of FDTs

In [24] and [41], a comparison of different FDTs when used to specifying RM-ODP systems from different viewpoints is presented. In both cases, the authors conclude that Lotos is most applicable for specifying the Computational viewpoint of a system. SDL'92 is also considered an appropriate FDT for specifying the computational viewpoint and allows the creation of object classes which can be multiply instantiated.

Z is considered a good FDT for modelling the RM-ODP Information viewpoint [24,41]. Thus, it is a useful FDT for describing the Trader in terms of invariants and information. It is not strong when used for behavioural description, as required by the Computational viewpoint. It does not accommodate multiple objects and thus, does not lend itself to describing a system of co-operating objects. Object Z allows multiple objects to be defined but lacks a suitable graphical form and is considered by many to be difficult to read like Z.

4.8 Summary

In this chapter, existing formal specifications of the Trader were examined. Each of the specifications provide an opportunity for improvement since:

- most are based upon early versions of the Trader Standard dating from as early as 1993 [9]. This resulted in specifications which do not address issues contained within the most recent Trading Standard [16] such as the new interworking protocol,
- some FDTs do not provide tools to allow simulations and lack an intuitive graphical representation which limits the usefulness of some models,
- there are no reports of analysis of the Trader based upon the formal models which have been created and
- many of the specifications are incomplete.

If a specification is to be used as a software engineering design tool, then features such as a graphical representation, executability and support for formal analysis become important factors in selecting a FDT. This leads us to the next chapter which introduces Coloured Petri Nets [66] as a FDT that can provide such features with the use of an associated tool, Design/CPN which provides support for specification creation, simulation and analysis.

Chapter 5

Coloured Petri Nets and Design/CPN

This chapter provides the reader with background information on Coloured Petri Nets (CPNs) and Design/CPN[™], the software package used for modelling and analysis of the ODP Trader. The first section introduces Coloured Petri Nets informally, and aims to provide the reader with a basic understanding of CPN mechanics through the description of a simple CPN. Those interested in a formal definition of CPNs are invited to consult [66]. When describing Design/CPN, a high-level model of the interactions between the Exporter, Trader and Importer is presented to illustrate some of the more complicated aspects of CPNs.

5.1 Coloured Petri Nets

Coloured Petri Nets (CPNs) were defined in 1981 by Kurt Jensen at Aarhus University in Denmark [62]. Jensen's trilogy of volumes covering CPN Basic Concepts [66], Analysis Methods [68] and Practical Use [69] are readily available and cover CPN theory, analysis and application. CPNs are a form of High-level Petri Net [63], and have been applied to a wide variety of applications including the Description of Services in Intelligent Networks [64], Specification and Verification of Protocols [65] and the design and validation of hardware at the register level [66]. The description of CPNs presented in this thesis uses the syntax of Design/CPN [83] which is a software tool for the creation, editing, simulation and analysis of Coloured Petri Nets.

CPNs are an extension of the concepts introduced by C. A. Petri in the original Condition/Event nets [67]. There are a number of features that have been added to C/E nets which improve their application to the modelling of complicated systems.

Like C/E nets, CPNs have a graphical form and a well-defined semantics suitable for formal analysis of the modelled system. A CPN is composed of the following elements:

- **Colour Sets:** Analogous to a "type" in a Programming Language.

- **Places:** Represented by ellipses, which are typed (thus being of a certain colour).
- **Transitions:** Represented by rectangles.
- **Arcs:** Represented by arrows, which define relationships between places and transitions.
- **Tokens:** Data values contained in places (consistent with the place's type).
- **Inscriptions:** Arcs and transitions may be inscribed with expressions containing constants, variables and functions. An inscription on a transition is called a *guard* which always evaluates to a boolean value.
- **A set of declarations:** used for defining Colour Sets and functions and declaring variables and constants.

The association of tokens with places in the CPN is known as the *marking* (or global state) of the CPN. The marking of a CPN is changed by the *occurrence* of transitions. A transition may occur if:

- there are appropriate tokens residing in its input places (determined by the arcs and inscriptions connecting places to the transition) and,
- the transition inscription (*guard*) is true.

When a transition occurs, tokens are removed from input places and created in output places according to the inscriptions on arcs connecting input and output places with the transition.

We will introduce these concepts with the help of Figure 5.1 which shows three places and one transition using Design/CPN™ syntax. The places are named **ProductA**, **ProductB** and **Result**. The transition is named **Go**. A single colour `size` has been defined in a Declaration Node. `Size` is a set with three elements: `big, medium, small`. The two variables which have been declared (`a` and `b`), are restricted to having a value which is a member of the colour set `size`. All of the places in this example are of colour `size`, as seen in figure 5.1, where `size` annotates each place. A place's colour set limits the tokens which it may contain to values within that colour set.

The presence of tokens in a place is indicated by a bold circle containing an integer which indicates the number of tokens located there. The text above a place indicates the multiset of tokens which are present. For the place named **ProductA**, `1 big` indicates that it contains one token which has the value `big`. Thus, `1` indicates the number of tokens, `'` is a separator and `big` indicates the value of the token.

Similarly, the **ProductB** place contains one token with value `small`. It can also be noted that **Result** contains no tokens. If desired, the textual representation of the place's marking may be hidden, allowing the model to remain uncluttered. There are three arcs in the example. They are from places **ProductA** and **ProductB** to transition **Go** and from transition **Go** to place **Result**. The

arc from *ProductB* to *Go* is bi-directional, and represents two arcs which are in opposite directions but have the same inscription.

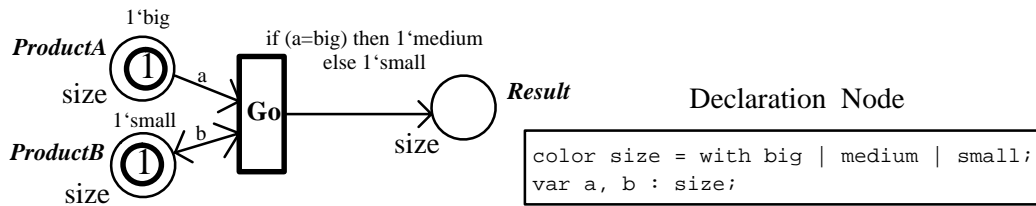


Fig. 5.1: A simple CPN with one transition and three places

There are three inscriptions in this CPN, each of which is associated with an arc. Both of the input arcs to transition *Go* have inscriptions (*a* and *b*) that are variables. The output arc from *Go* has a more complex inscription which involves a conditional test on the variable *a*.

When a variable is assigned a value, it is known as *binding*. For a particular binding to variables, if all the input places connected to a transition by arcs have at least all of the tokens of the required number and colour (as defined by the arc inscriptions), then the transition is said to be *enabled* in that binding. More than one transition may be enabled at once. A transition that is enabled may *occur* or *fire* (these terms will be used synonymously throughout the thesis). When a transition occurs, tokens are removed from the input places and added to output places, as indicated by the arc inscriptions. This is known as the *token game* and is a feature of CPNs that make them effective at illustrating data and control flows. In Figure 5.1, we can see that transition *Go* is enabled when *a* is bound to the value *big* and *b* is bound to *small*. When transition *Go* occurs, both of the tokens in the input places are consumed, and a token with value *medium* is placed into the output place as shown in figure 5.2. Since there is a bi-directional arc between *ProductB* and *Go*, a *small* token is also created in *ProductB*. This effectively replaces the token which was consumed when the transition occurred.

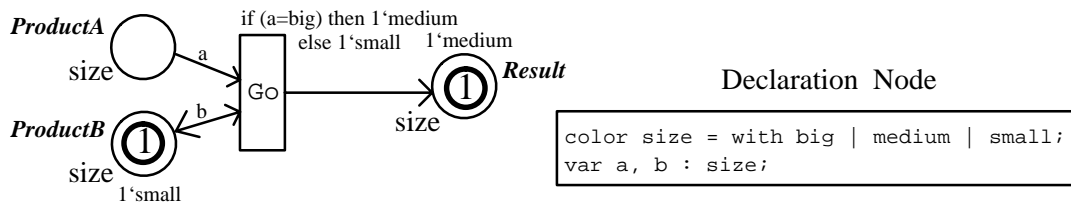


Fig. 5.2: Simple CPN after transition *Go* occurs

CPNs support hierarchical modelling. This means that a model may be spread out over a number of *pages*, where each page models different aspects of the system, allowing modularity. A page that is *used* by another is known as a *sub-page* of the *super-page*. The CPN on the sub-page is known as a sub-net. The main construct for this is a *substitution transition*.

We will introduce many of these concepts later in section 5.4 using a very high-level model of the Trader. It is an abstract model of multiple objects that communicate through message passing. The use of hierarchies will be illustrated in Chapter 6, where the Trader is modelled in greater detail.

5.2 CPN Analysis Techniques

When modelling a system with CPNs, one of the main aims is to analyse the model to verify its correctness. Since CPNs have an underlying formal semantics, it is possible for a number of techniques to be used to analyse a model's dynamic behaviour. Methods for the analysis of CPNs are addressed by Jensen in [66] and [68]. This section provides an overview of these analysis methods.

5.2.1 Occurrence Graphs

An Occurrence Graph [66,68] (OG) represents the entire state space of the model and can be used to identify behaviour that may not be initially evident through simulation runs. When a simulation is run, it is possible that there are multiple transitions that are enabled at one step. Since the user or simulator chooses only one of these enabled transitions for firing, there are many possible behaviours for the model that are not investigated. An occurrence graph may be generated by a tool for a given initial state.

An occurrence graph consists of nodes connected by arcs where each node represents a marking (or state) of the CPN and each arc represents the occurrence of a transition with a particular binding to variables that leads to the next CPN marking. Arcs have an associated binding element that records the bindings for variables used to enable the transition which resulted in the new CPN marking. For each of the many legal combinations of bindings for variables on input arcs to the transition, a node is created that represents a resulting marking of the system when the transition is enabled with that specific binding.

This process is repeated for all enabled transitions and with each possible binding for variables on transition input arcs. Since a new node is created for all of the possible markings of the system, the occurrence graph of any non-trivial CPN becomes very large and consumes a great deal of computer memory. In order to manage systems with large state spaces, it is important to use some reduction techniques to reduce the size of the model's occurrence graph.

5.2.2 Occurrence Graphs with Equivalences

One technique which allows the modeller to reduce the size of an OG under certain conditions is Occurrence Graphs with equivalence classes [68]. An add-on tool exists for Design/CPN[™] that allows the user to apply Equivalences and Symmetries (see section 5.2.3) when generating the OG [85]. It is known as known as OEOS (Occurrence Graphs with Equivalence and Symmetry).

In some situations, it may be desirable to consider two markings equivalent by ignoring a certain tuple in the marking. For example, a model that associates a transaction identifier with operations non-deterministically has a node in the OG for each of the possible transaction identifiers. The value of the transaction identifier is important for the correct operation of the model, but is unimportant for the OG. This is because the value of the transaction identifier used in a transaction is not important to the high-level functionality of the model. However, it must be

unique to ensure that transactions can be differentiated. Thus, if it were possible to map out the transaction identifier from the marking of the CPN, then slightly different markings could be considered the same. This is the rationale behind using OGs with Equivalence Classes, where similar markings are considered the same, which reduces the number of nodes in the OG. This technique is used later in the thesis for reducing the size of the OG (sections 5.4.5.3 and 7.1.4), thereby reducing the computational resources required to analyse the model.

5.2.3 Occurrence Graphs with Symmetries

Occurrence Graphs with Symmetries use the same reduction technique as OGs with Equivalence Classes except the technique takes advantage of symmetries within the system being analysed. For example, consider a system which models resource usage by four processes (p1, p2, p3 and p4) which have the same resource usage requirements. The behaviour of the model when p1 is consuming a resource while p2, p3 and p4 are idle is the same as the behaviour of the model when p2 is consuming the resource and p1, p3 and p4 are idle (since we have stated that the processes have the same resource requirements).

It is possible to see that there is a symmetry throughout the markings of the CPN which allows a direct substitution of p2 for p1 which results in an identical marking. This is known as a symmetrical marking since it is possible to obtain one from the other by applying a consistent substitution of one identifier for the other. Thus, using symmetries, it is possible to consider only one OG which contains one class of the symmetrical markings to describe the entire system's behaviour. Using a symmetry mapping function, it is possible to map all other OG sequences onto the reduced OG. For a precise definition of OGs using Symmetries in the context of CPNs, see [68].

5.2.4 Stubborn Sets

Stubborn Sets can be used to dramatically reduce the size of an OG when applied to models that contain a number of independent concurrent processes. Valmari states that “the basic observation behind the stubborn sets method is that *concurrency introduces irrelevant duplication of states*” [77]. An example of this is a message passing system containing objects with internal state which can execute independently of each other. In such systems, it is possible for there to be a number of concurrently enabled transitions which are not in conflict with each other. In such a situation, the order in which the transitions occur is not important since the marking obtained after each transition occurs is the same.

This can be seen in figure 5.3 [78], in which two transitions (τ and τ') are concurrently enabled with marking m (circles represent markings whilst arrows represent transitions that may occur in that marking). If τ occurs first, the system marking changes to m' whereas if τ' occurs first, the system marking changes to m'' . Whilst in m'' , only τ may occur which changes the marking to m''' . Similarly, when in m' , only τ' can occur which changes the marking to m''' . It is easy to see that independent of the order in which τ and τ' occur, the same marking (m''') is always reached.

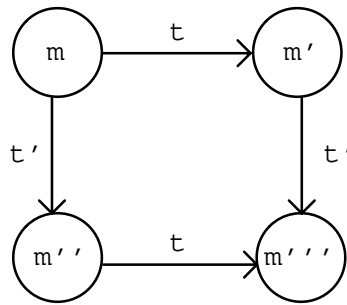


Fig. 5.3: Example of the diamond sub-structure

Stubborn sets control the state-space explosion problem by removing redundant markings and transition occurrences from the OG resulting in a Reduced Reachability Graph (RRG). For an introduction to the theory behind stubborn sets and their realisation in CPNs, the reader is directed to [79].

5.2.5 Invariants

Using place invariants allows a CPN model to be analysed without resorting to the brute-force calculation of a large OG. However, it requires significant knowledge of the model and mathematical skill in order to apply them effectively. The basic principle of operation is to determine equations which are expected to hold true regardless of the order in which transitions occur. Having proved these equations to be correct, the equations can be used to prove many important dynamic properties of the modelled system such as reachability, boundedness, home, liveness and fairness properties [68], all of which are defined in [66]. As with all CPN analysis techniques, computer-based tool support is critical for effective analysis of non-trivial models.

5.3 Design/CPN

The Design/CPN[™] [83] software package was originally developed by Meta Software Corporation of Massachusetts, U.S.A in collaboration with the University of Aarhus. It was created as a commercial product and provides the user with a suite of integrated tools that enable the editing, simulation and verification of Coloured Petri Net models.

The Trader has been modelled and analysed using Design/CPN[™], which is currently at a major release of Version 3 and is supported by Aarhus University in Denmark. There are versions of the package for Solaris, Linux and Macintosh operating systems.

Information on obtaining a free license for Design/CPN[™] may be obtained at: <http://www.daimi.aau.dk/designCPN/>

5.3.1 Editing

The Editing facilities of the tool are used for the creation, modification and syntax checking of hierarchical Coloured Petri Net models. The following sections describe important aspects of the editor when creating a CPN in Design/CPN[™].

5.3.1.1 Global Declaration Node

In order to define colour sets and functions, and declare variables and constants, a Global Declaration Node (GDN) must be created. In Design/CPN™, these elements are written using a customised version of the Standard ML (SML) [86] functional programming language known as CPN/ML. For clarity, the three major elements of the GDN (colour sets, functions and variables) are separated and annotated with comments for readability and maintainability. Firstly, colour sets are defined and annotated with comments to allow for easy reading of the colour set definitions. The SML interpreter allows the declarations to be split over a number of lines which in some cases, improves the readability of large colour sets having many elements. Having declared the colour sets, functions used throughout the model are defined.

5.3.1.2 CPN Structure

Using the editor, the user is able to create a model of the system being investigated. It allows the creation of places, transitions and arcs for the CPN structure and auxiliary objects that are used to improve the model's readability. An object in this context refers to the graphical components that are drawn by the editor, such as places, transitions and arcs. All graphical objects have attributes that may be modified.

Each place has an associated colour set which must be assigned before the CPN passes a syntax check. Places may be given an initial marking which details the contents of the place when the model is initialised. Places and transitions may also have an associated name. Arcs have an associated inscription and transitions may have a guard attached that limits their enabling conditions.

When the model is complete, Design/CPN™ may be instructed to perform a syntax check that ensures that the CPN is syntactically correct. Of course, this check does not ensure that user-defined functions are logically correct, or ensure that the model is semantically correct.

5.3.1.3 Hierarchies

Design/CPN™ allows the creation of hierarchical models which may be spread out over a number of **pages**. Each page models different aspects of the system which allows a model to be created in a modular manner.

A page which is “used” by another is known as a sub-page of a super-page. The CPN on the sub-page is known as a sub-net. Figure 5.4 shows how a CPN may be composed of other CPN pages. CPN1 models 3 objects that are connected to each other. CPN2 is used to define the internal structure of objects 1 and 3, whereas CPN3 is used to define object 2's behaviour.

Design/CPN™ implements hierarchies by the use of substitution transitions and place fusion. A substitution transition is a transition that acts as a “placeholder” for a sub-net. All of the input and output places associated with the substitution transition have a corresponding place in the sub-net. Thus, there is a mapping for each place to a corresponding place in the sub-net. This is implemented in Design/CPN™ using place fusion since a place is considered to exist at more than

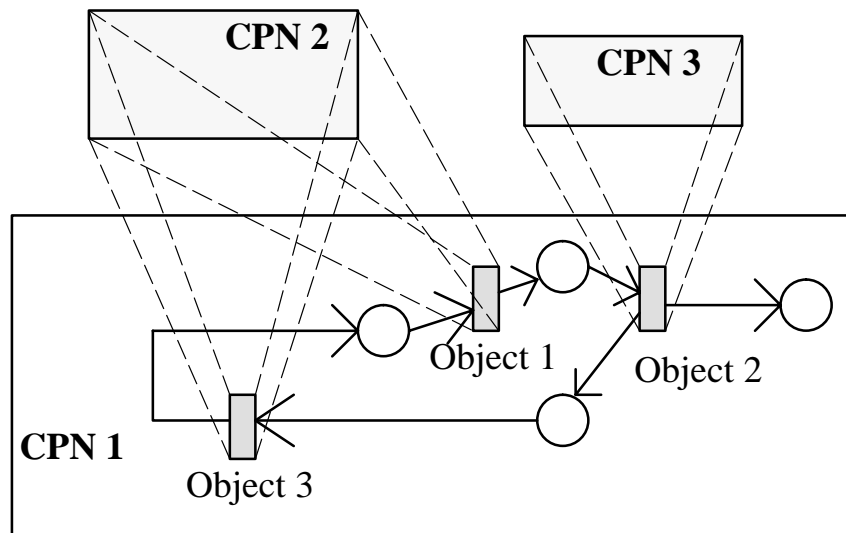


Fig. 5.4: An example of CPN Hierarchy

one place in the CPN. This concept will be illustrated in Chapter 6, where a detailed description of the CPN model of the Trader is presented.

5.3.2 Simulation

The simulator provides the user with a dynamic view of the model. This is a very important feature of Petri Nets, since it allows the user to **execute** the CPN and observe the model's behaviour. There are a number of simulation parameters that may be configured by the user. For a highly controlled simulation, the user may require graphics to be updated and for a breakpoint to be activated after each transition is fired. This allows the user to watch the *token game*, which is the movement of tokens throughout the CPN as transitions fire. It is also possible for the user to select individual transitions that should be fired. In this manner, the user has control over the sequence of events that occur within the model. Therefore, the user can debug the model since they have complete control over which transitions are fired and can interrogate the marking of the CPN after each firing.

Alternatively, for a more random simulation, the user may decide to perform an automatic simulation, where the user has no control over the selection of transitions for firing. Instead, the simulator randomly selects a transition for firing from the set of enabled transitions. With this mode, it is possible to have simulation graphics updated at the end of a simulation run rather than as each step occurs, resulting in faster simulations. The automatic simulator configuration can be used for testing models that are considered to be bug-free and should therefore execute and terminate in a legal state.

The simulator creates a Simulation Report that records details of simulation runs. For each transition that is fired, the report records the values that were bound to variables on input arcs connected to the transition. This provides a trace of the model's execution and is an effective tool for debugging a model since it can highlight transitions that are being enabled when they should not according to the semantics of the model.

5.3.3 Analysis

Design/CPN's main analysis technique is Occurrence Graphs [84] since it incorporates a tool that may be used for the generation of basic occurrence graphs. There is also an additional library available for using OGs with equivalences and symmetries [85].

As previously discussed in section 5.2.1, OGs are a representation of the entire state space of the CPN model for a given initial condition. Design/CPN[™] may be used to analyse the dynamic behaviour of the model under consideration.

In systems that are cyclic, OG analysis can be used to detect deadlocks and/or livelocks that are caused by concurrently operating entities. This is achieved by generating the OG of the system and then applying well-known algorithms to detect these properties [66]. In non-cyclic systems, OG analysis may be used to verify the correct functionality of a model in specific scenarios. This is where the tool is used to determine all of the possible termination states of the model. This information can be used to verify that the model always behaves in a desirable manner and terminates correctly.

When creating an OG, it is possible for the user to specify a time or node limit for the generation of the OG. This is useful since it allows the user to terminate the generation of enormous OGs. The user may also control the branching factor when generating the OG, thereby allowing an OG to be created depth or breadth first. This is discussed in more detail in section 7.1.3.

5.4 A Top-level Trader Interaction CPN

The CPN model in figure 5.5 is based upon the basic Trader interactions described previously in section 3.4. The model contains three object types: a Trader, an Exporter and an Importer. There are two instances of both the Importer and Exporter objects and thus, including the Trader, the system contains a total of 5 object instances. These objects communicate through dedicated places that represent simple unidirectional communication paths between objects.

The model contains two exporters and two importers which models a Trading system in which there are multiple importers and exporters. The number of possible services in the system is limited to two in order to ensure that the state space of the system is manageable. In the following sections, the Colour Sets, variables and functions used in the model will be presented, as will the structure of the model and the results of simulation and analysis.

It may be useful for the reader to refer to the colour set descriptions in section 5.4.1 when they are reading section 5.4's description of the Top-Level Trader model.

5.4.1 Global Declaration Node

A fundamental component of the Global Declaration Node (GDN) are colour sets, which are analogous to types in a programming language. They allow the inclusion of arbitrarily complex data structures, thereby enabling richer models to be constructed. The GDN of the Top_Level

Top-Level Trader Interaction Model

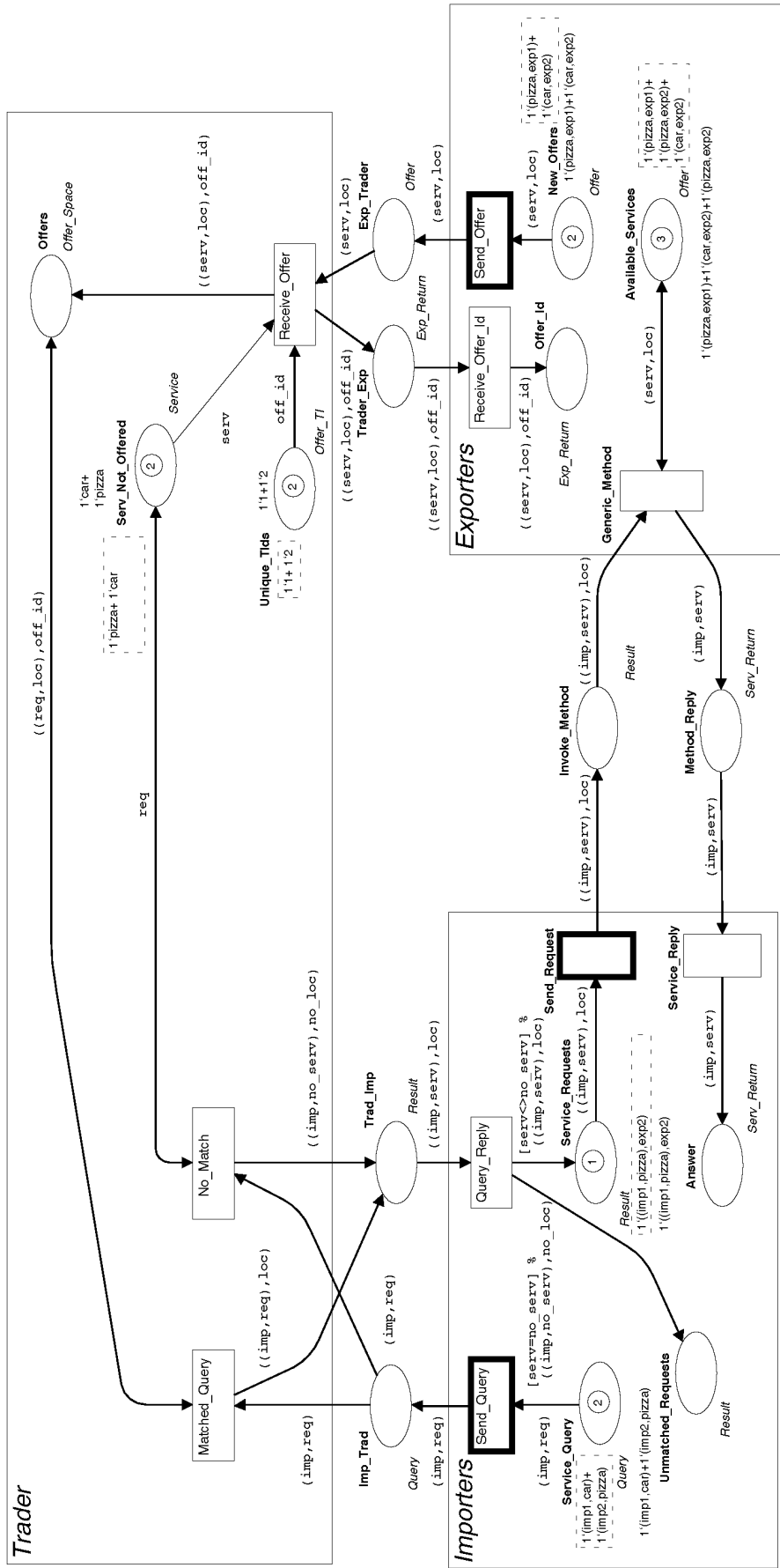


Fig. 5.5: Top-Level Trader Interaction Model

model in fig. 5.5 is given in fig. 5.6. A detailed explanation of the colour sets used in the CPN is given below.

```
color Offer_TI = int with 1..2;
```

Colour set `Offer_TI` contains integer values 1 and 2. The colour is used in the model for representing unique Offer Transaction Identifiers (OTI). As there are only two offers in the model, only two unique identifiers are required. With a larger offer space, the number of unique identifiers in `Offer_TI` must also increase.

```
color Service = with pizza | car | no_serv;
```

Colour set `Service` contains two elements that represent service types available for import or export by objects in the system. It also has an element (`no_serv`) that represents the absence of a server.

```
color Importers = with imp1 | imp2;
color Location = with exp1 | exp2 | no_loc;
```

Colour sets `Importers` and `Location` represent unique object identifiers that are associated with object instances. `Location` is used rather than `Exporter` since it is possible for a service to be available at an object other than the exporter. As with the `Service` colour set, `Location` has an element which is used to represent the absence of a location when matching services.

```
color Offer = product Service*Location;
```

Colour set `Offer` is defined as a cartesian product of two previously defined colour sets. The colour set is a tuple that contains a *Service* and a *Location* where the service is provided.

```
color Offer_Space = product Offer*Offer_TI;
```

Colour set `Offer_Space` is also a cartesian product of two previously defined colour sets. It is used to model the Offer Space since it contains information associated with an Offer and also a unique transaction identifier.

```
color Exp_Return = Offer_Space;
```

This colour set contains the same elements as the `Offer_Space` colour set. It is renamed so that the colour set name reflects the context of use in the model.

```
color Query = product Importers*Service;
```

This colour is used to model a Query request. It contains the identifier of the Importer that is issuing the request, and also the *Service* type that is to be matched.

```
color Result = product Query*Location;
```

The `Result` colour set is defined as a cartesian product of the `Query` and `Location` colour sets. It is used to represent the trader match return data, where the original Query data is coupled with a service location.

```
color Serv_Return = Query;
```

The `Serv_Return` colour set is defined to be the same colour set as `Query`. It is used in the model to represent the return value from a method invocation on a server object.

```

(** Global Declaration Node **)
(** Colour Definitions **)
(** Used for Unique Offer Transaction Identifiers **)
color Offer_TI = int with 1..2;

(** Possible service types **)
color Service = with pizza | car | no_serv;

(** Possible Importer and Service Location Identifiers **)
color Importers = with imp1 | imp2;
color Location = with expl | exp2 | no_loc;

(** Exported Offers **)
color Offer = product Service*Location;

(** Offer with a unique identifier **)
color Offer_Space = product Offer*Offer_TI;

(** Exported offer with unique identifier **)
color Exp_Return = product Offer*Offer_TI;

(** Import Query **)
color Query = product Importers*Service;

(** Query and service location **)
color Result = product Query*Location;

(** Service invocation return type **)
color Serv_Return = Query;

(** Variable Declarations **)
var serv : Service;
var req : Service;
var loc : Location;
var off : Offer;
var off_id : Offer_TI;

(* Functions used in the OEOS Occurrence Graph Creation. *)
(* Implementation of a utility function which maps out *)
(* the Offer_Id in an Offer_Space or Exp_Return. *)

fun Mapout_Offer_Id Offer_ms =
  let
    (* Convert the multi-set into a list *)
    val Offer_Id_list = ms_to_list Offer_ms
    (* MapoutOne removes Offer_Id from an Offer_Space or *)
    (* Exp_Return using a pattern match *)
    fun MapoutOne ((Service,Location),_) = (Service,Location)
  in
    (* mapout and convert back to multi-set *)
    list_to_ms (map MapoutOne Offer_Id_list)
  end;

```

Fig. 5.6: Global Declaration Node of the Top-Level CPN

The model also requires five variables which are defined below and used on arcs.

```

var serv : Service;
var req : Service;
var loc : Location;
var off : Offer;
var off_id : Offer_TI;
var imp : Importers;

```

Variables are placed on arcs and allow tokens with different values to be moved through the CPN. Without variables, arcs could only move tokens with a single value, as designated by the arc inscription.

The final component of the GDN is the declaration of functions. In this model, only one function is defined. It is used later in the analysis stage and will be described in more detail in section 5.4.5.3.

5.4.2 Pages in the model

The model contains a total of six pages including a hierarchy page that contains nodes representing pages in the model (Hierarchy#10010) as depicted in figure 5.7. The CPN of the model is located on the Top_Level#1 page. The GDN is located on the GDN#2 page, and two other pages are used for drawing the OG and OG with Equivalence Classes that were generated by the tool (Draw_OG#4 and Draw_OEOS#5). By using multiple pages, it is possible to encapsulate information on separate pages for greater readability.

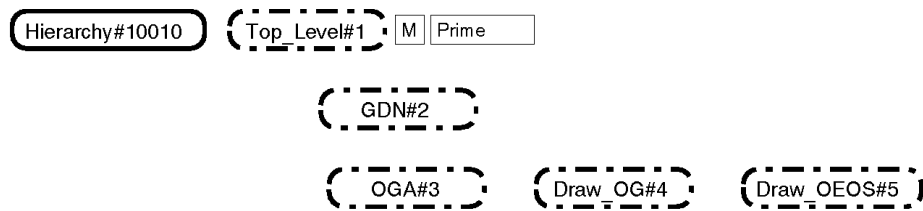


Fig. 5.7: Hierarchy page of the Top-Level CPN

5.4.3 Initial Marking

The initial marking of the model is used to define the initial state of the system. In this model, there are six places with non-empty markings. They are *New_Offers* and *Available_Services* in the Exporter object, *Service_Query* and *Service_Requests* in the Importer object and *Unique_Tids* and *Serv_Not_Offered* in the Trader object.

New_Offers contains two tokens representing the availability of a pizza service at exp1 and a car service at exp2. These Offers will be added to the Trader's Offer Space when they are exported. The marking shown to the right of *New_Offers* is represented as follows:

```
1'(pizza,exp1)+1'(car,exp2)
```

Service_Query contains Query tokens destined for the Trader's Query Operation, requesting it to search its Offer Space for matching Offers. These Queries represent requests by imp1 for a car service and imp2 for a pizza service. The initial marking of *Service_Query* is given by:

```
1'(imp1,car)+1'(imp2,pizza)
```

There is also a token at *Service_Request* that represents a request by imp1 for a pizza service provided at exp2. In this case, the imp1 object already knows that exp2 provides a Pizza service and thus, it does not require the use of a Trader.

```
1'((imp1,pizza),exp2)
```

The Exporters' *Available_Services* contains three tokens representing the services offered by exp1 and exp2. In this model, exp1 provides a pizza service while exp2 provides both a car and pizza service.

```
1'(pizza,exp1)+1'(pizza,exp2)+1'(car,exp2)
```

Unique_Tids contains two tokens that represent unique offer transaction identifiers.

```
1'1 + 1'2
```

The Transaction identifier returned to an Exporter is modelled by the *Offer_TI* colour set. An identifier must be unique but can be assigned an arbitrary value. These identifiers are returned by the Trader to the Exporter and allow the Exporter to reference Offers that have been exported at a later date for deletion or modification. This model does not include such Trader operations since it is limited to the modelling of basic Trading Environment interactions.

The initial marking of the model (indicated by 1 in the first row) is shown in figure 5.8. *Top_Level* is the name of the Page on which the places are located. The place name is given next and is separated from the Page name by an apostrophe. The page instance number is given next and since there is only one instance of the *Top_Level* page, this is equal to 1. Following the colon, which terminates the place identification string, is the marking of the place. Where the place does not contain any tokens, the string *empty* is used.

```
1
Top_Level'Offers 1: empty
Top_Level'Serv_Not_Offered 1: 1'pizza+ 1'car
Top_Level'Exp_Trader 1: empty
Top_Level'Trader_Exp 1: empty
Top_Level'Offer_Id 1: empty
Top_Level'Trad_Imp 1: empty
Top_Level'Imp_Trad 1: empty
Top_Level'Service_Query 1: 1'(imp1,car)+ 1'(imp2,pizza)
Top_Level'Service_Requests 1: 1'((imp1,pizza),exp2)
Top_Level'Invoke_Method 1: empty
Top_Level'Method_Reply 1: empty
Top_Level'Answer 1: empty
Top_Level'Unique_Tids 1: 1'1+ 1'2
Top_Level'Available_Services 1: 1'(pizza,exp1)+ 1'(pizza,exp2)+ 1'(car,exp2)
Top_Level'New_Offers 1: 1'(pizza,exp1)+ 1'(car,exp2)
Top_Level'Unmatched_Requests 1: empty
```

Fig. 5.8: Initial Marking of the Top-Level CPN

5.4.4 Dynamic behaviour

Initially, there are three concurrently enabled transitions in figure 5.5, each of which is indicated by a bold rectangle. These transitions are **Send_Query**, **Send_Offer** and **Send_Request**. It is possible for any combination of these transitions to be fired by the simulator. For the purposes of this discussion, we will select the **Send_Offer** transition in the Exporter for firing.

5.4.4.1 Exporter-Trader Interaction

There are two tokens in *New_Offers* and no restriction placed on the binding for variable *serv* (such as the use of a guard on transition **Send_Offer**). There are three possible values for the binding of *serv*, namely *pizza*, *car* and *no_serv*. The simulator arbitrarily chooses one value from the set of bindings which enable the transition and allows the transition to occur with that binding for variables. Note that there are only two values that *serv* can be bound to which

results in the enabling of *Send_Offer* (pizza or car) For the purposes of this example, we will select a binding where `serv = pizza` and `loc = exp1`.

As previously explained in section 5.1, when a transition occurs, appropriate tokens in the input places are consumed and tokens are created in output places. When *Send_Offer* occurs, (pizza, exp2) is removed from *New_Offers* and created in *Exp_Trader*, which is a communications place between the Exporter and the Trader. Thus, *Exp_Trader* contains one token with value (pizza, exp1).

At this point, the *Send_Query* and *Send_Request* transitions are still enabled, the *Receive_Offer* transition in the Trader has become enabled and the *Send_Offer* transition is still enabled using the remaining binding for (serv, loc) of (car, exp2).

Unique_Tids is a place that contains unique transaction identifiers that can be associated with an Offer when it is processed. *Offers* models the Trader's database of service offers and *Serv_Not_Offered* is a place that is used to flag services that are not yet offered by the Trader. This place allows the Trader to determine when a request cannot be matched.

After the occurrence of *Send_Offers* (as above), *Receive_Offer* is enabled with the following bindings:

```
serv = pizza, loc = exp1, off_id = 1 or
serv = pizza, loc = exp1, off_id = 2
```

This is because the value of `off_id` is chosen from the tokens in *Unique_Tids*. The CPN simulator arbitrarily selects one of these possible bindings, such as for example, `off_id = 1`. When the transition occurs, the Offer-Token with value 1 is removed from *Unique_Tids*, a Service token with value `pizza` is removed from *Serv_Not_Offered*, an Offer_Space token with value ((pizza, exp1), 1) is placed in *Offers* and an Exp_Return token is placed in *Trader_Exp* with the same value as that placed in *Offers*.

When *Receive_Offer_Id* occurs, the token returned by the Trader to the Exporter (after an Offer is exported) is moved into the Trader (*Exported_Offers*). In a Trader implementation, this information would be stored for later use when withdrawing or modifying Offers.

5.4.4.2 Importer–Trader Interaction

Importers may send requests to the Trader via the *Send_Query* transition. When *Send_Query* has occurred, either of the *Matched_Query* or *No_Match* transitions can become enabled. If the service has been exported, then the *No_Match* transition cannot become enabled because *Serv_Not_Offered* does not contain a token that matches the requested service. Thus, only *Matched_Query* can become enabled since there must be at least one matching offer token in *Offers*. When *Matched_Query* occurs, the matching offer is placed in *Trad_Imp* so that it may be returned to the calling Importer object via the *Query_Reply* transition.

If a query is received by the Trader for which it cannot find a matching service in *Offers*, then *No_Match* becomes enabled. When it occurs, a token is created in *Trad_Imp* that indicates to the importer that the resulting matched service is *no_serv*, which represents a null service. Due to the

concurrent nature of the model, it is possible for a query token to stay in *Imp_Trade* until a matching service is exported to the Trader via **Receive_Offer**. This models the fact that incoming methods may not be serviced immediately after they are received by an object.

When **Query_Reply** occurs, the matching offer is placed in *Service_Request* if the query resulted in a successful match, or *Unmatched_Requests* if the Trader was unable to match the query.

5.4.4.3 Importer–Exporter Interaction

Send_Request models the invocation of methods by the Importer on one of the Exporters and when it occurs, has the effect of sending the request to the Exporter objects. **Generic_Method** is a transition that models all legal Exporter object method invocations. The transition will only be enabled if the incoming request has a matching token in *Available_Services*. At this level of abstraction, the model assumes that Exporters support the services that they have exported to the Trader. The model also assumes that the initial request contained in *Service_Requests* is for a valid service.

There are two input arcs to **Generic_Method**. Both of these arcs have an inscription that refers to `serv` and `loc` variables. Thus, in order for the transition to be enabled, there must be tokens in both of the connecting input places that have the same value for `serv` and `loc`. This models the fact that certain services are available at certain exporters, but not all services are available at all exporters.

When the **Generic_Method** transition has occurred, the final step in the model is to return the result of the method to the calling object (an importer). This is modelled by the **Service_Reply** transition that passes the return value back to the Importer object that invoked the method.

Due to the high degree of concurrency within the model, there are many possible orderings for the firing of transitions. The sequence of transition firings described in this section is only one of the many possible combinations of bindings and occurrence orderings.

5.4.5 Analysis of the Top–Level Trader Model

When analysing the model, the first step is to test the model using the simulator which can be used to check for desired behaviour. Having tested the model, the next step is to use occurrence graph analysis techniques to verify correct terminating conditions and the absence of deadlock or livelock in the model.

5.4.5.1 Simulation

The most basic of analysis techniques is interactive simulation, where the user selects a transition to fire from the set of all enabled transitions. In figure 5.5, the model is shown in its initial state, where three transitions are enabled (**Send_Offer**, **Send_Query** and **Send_Request**). Enabled transitions are denoted by a bold outline of the transition rectangle.

Alternatively, it is possible for the user to perform an automatic simulation run. This is where the simulator automatically selects an enabled transition to be fired and continues to do so until a

user-defined number of steps has been reached, or there are no more enabled transitions. The user may disable graphical feedback until the simulation is completed which results in much faster simulations.

Simulation is a simple way to locate errors in modelling logic. A complete automatic mode simulation run of the CPN in figure 5.5 requires 18 steps. This means that there are 18 enabled transitions that occur during the simulation run. At the end of the run, it is possible to save a simulation report that contains an ordering of transitions that were fired and the bindings of variables used in the enabling of the transition, as shown in figure 5.9.

Each line of the simulation report shows the transition that fired, the page instance that it is located on and the values of all bindings to variables at that transition. It is useful for this report to be followed step by step while referencing the model in figure 5.5 since it illustrates how the occurrence of concurrently enabled transitions is interleaved.

```

Simulation Report
1 A Send_Offer@(1:Top_Level#1)
  { loc = exp2,serv = car}
2 A Send_Request@(1:Top_Level#1)
  { imp = imp1,loc = exp2,serv = pizza}
3 A Send_Query@(1:Top_Level#1)
  { imp = imp2,req = pizza}
4 A Receive_Offer@(1:Top_Level#1)
  { loc = exp2,off_id = 2,serv = car}
5 A Send_Query@(1:Top_Level#1)
  { imp = imp1,req = car}
6 A No_Match@(1:Top_Level#1)
  { imp = imp2,req = pizza}
7 A Generic_Method@(1:Top_Level#1)
  { imp = imp1,loc = exp2,serv = pizza}
8 A Send_Offer@(1:Top_Level#1)
  { loc = exp1,serv = pizza}
9 A Service_Reply@(1:Top_Level#1)
  { imp = imp1,serv = pizza}
10 A Matched_Query@(1:Top_Level#1)
  { imp = imp1,loc = exp2,off_id = 2,req = car}
11 A Receive_Offer@(1:Top_Level#1)
  { loc = exp1,off_id = 1,serv = pizza}
12 A Query_Reply@(1:Top_Level#1)
  { imp = imp2,loc = no_loc,serv = no_serv}
13 A Receive_Offer_Id@(1:Top_Level#1)
  { loc = exp2,off_id = 2,serv = car}
14 A Receive_Offer_Id@(1:Top_Level#1)
  { loc = exp1,off_id = 1,serv = pizza}
15 A Query_Reply@(1:Top_Level#1)
  { imp = imp1,loc = exp2,serv = car}
16 A Send_Request@(1:Top_Level#1)
  { imp = imp1,loc = exp2,serv = car}
17 A Generic_Method@(1:Top_Level#1)
  { imp = imp1,loc = exp2,serv = car}
18 A Service_Reply@(1:Top_Level#1)
  { imp = imp1,serv = car}

```

Fig. 5.9: Simulation report from the Top-Level CPN

As can be seen, the model terminates after both requests are serviced by the Trader. One of the queries is processed by the Trader before it has a matching service, since **No_Match** occurred before a matching service was exported. It is possible for a query to stay in *Imp_Trade* until a

matching service is exported to the Trader by **Receive_Offer**, but this depends entirely upon the arbitrary occurrence order which is determined by the simulator in automatic mode.

5.4.5.2 Occurrence Graph Analysis

When verifying a model, it is important to ensure that all possible behaviours are checked. This may be accomplished by generating an occurrence graph for the model when it has been placed in its initial state. No user input is required as the OGA tool defaults to a time limit of 300 seconds of processing when generating the OG. The results obtained from the OG generation using an Intel Pentium 133 computer with 80 Mbytes of RAM are shown in Table 1 (FULL means that the entire OG was generated).

OG
Nodes=5152
Arcs=18616
Seconds=337
FULL

Table 1 – OGA Generation statistics from the Top–Level CPN

Having generated the entire OG, the next step is to perform analysis. For this model, the most important test required is the detection of dead markings in the OG. These are markings that do not lead to successor markings and are therefore terminal nodes in the OG. A call to the `ListDeadMarkings()` function call returns the following node list: [4944, 4945, 5060, 5061, 5070, 5088, 5151, 5152], two of which are shown in figures 5.10 and 5.11.

There are eight possible states in which the model can terminate. This is expected for the following reasons:

- it is possible for requests not to be matched, thereby creating four permutations of Query matching results:
 - 1 both Queries matched successfully,
 - 2 imp1’s car Query is successfully matched but imp2’s pizza Query is not,
 - 3 imp2’s pizza Query is successfully matched but imp1’s car Query is not,
 - 4 neither of imp1 or imp2’s queries are successfully matched.
- For each of these four result permutations, there are two possible values for the transaction identifier which is arbitrarily assigned to an offer when **Receive_Offer** occurs. This results in eight possible terminal markings in the OG.

Inspection of these eight markings reveals that four of the eight terminal nodes are equivalent except for the Offer Identifier that was associated with Offers as they were entered into *Offers*. In each marking, the `pizza` and `car` services have been associated with either offer identifier 1 or 2.

The marking shown in figure 5.10 shows when `imp1`'s request is not matched but `imp2`'s request is. Figure 5.11 shows the same final marking except for the exported offer's transaction identifiers.

```

5060
Top_Level'Offers 1: 1'((pizza,exp1),1)+ 1'((car,exp2),2)
Top_Level'Serv_Not_Offered 1: empty
Top_Level'Exp_Trader 1: empty
Top_Level'Trader_Exp 1: empty
Top_Level'Offer_Id 1: 1'((pizza,exp1),1)+ 1'((car,exp2),2)
Top_Level'Trad_Imp 1: empty
Top_Level'Imp_Trad 1: empty
Top_Level'Service_Query 1: empty
Top_Level'Service_Requests 1: empty
Top_Level'Invoke_Method 1: empty
Top_Level'Method_Reply 1: empty
Top_Level'Answer 1: 1'(imp1,pizza)+ 1'(imp2,pizza)
Top_Level'Unique_TIds 1: empty
Top_Level'Available_Services 1: 1'(pizza,exp1)+ 1'(pizza,exp2)+ 1'(car,exp2)
Top_Level'New_Offers 1: empty
Top_Level'Unmatched_Requests 1: 1'((imp1,no_serv),no_loc)

```

Fig. 5.10: Marking of node 5060 from OGA analysis of the Top-Level CPN

```

5061
Top_Level'Offers 1: 1'((pizza,exp1),2)+ 1'((car,exp2),1)
Top_Level'Serv_Not_Offered 1: empty
Top_Level'Exp_Trader 1: empty
Top_Level'Trader_Exp 1: empty
Top_Level'Offer_Id 1: 1'((pizza,exp1),2)+ 1'((car,exp2),1)
Top_Level'Trad_Imp 1: empty
Top_Level'Imp_Trad 1: empty
Top_Level'Service_Query 1: empty
Top_Level'Service_Requests 1: empty
Top_Level'Invoke_Method 1: empty
Top_Level'Method_Reply 1: empty
Top_Level'Answer 1: 1'(imp1,pizza)+ 1'(imp2,pizza)
Top_Level'Unique_TIds 1: empty
Top_Level'Available_Services 1: 1'(pizza,exp1)+ 1'(pizza,exp2)+ 1'(car,exp2)
Top_Level'New_Offers 1: empty
Top_Level'Unmatched_Requests 1: 1'((imp1,no_serv),no_loc)

```

Fig. 5.11: Marking of node 5061 from OGA analysis of the Top-Level CPN

The model in figure 5.5 associates a random transaction identifier with Offers that are entered into the Trader's Offer Space. Since there are only two offers that are exported to the Trader, there are two possibilities when associating transaction identifiers with requests. This has the effect of generating two terminal nodes as analysis of the model has shown.

The only difference between the two dead markings is the marking of *Offers* and *Offer_Id*, in which the transaction identifier has been associated with the other exported service. Whether the `pizza` service is associated with transaction identifier 1 or 2 is inconsequential to the semantics of the model. In generating the OG, both possibilities are treated separately which results in a much larger OG.

Part of the first three steps of the OG are illustrated in figure 5.12. Node 1 represents the initial marking of the model and is located at the top of the graph. There are five arcs leading from node

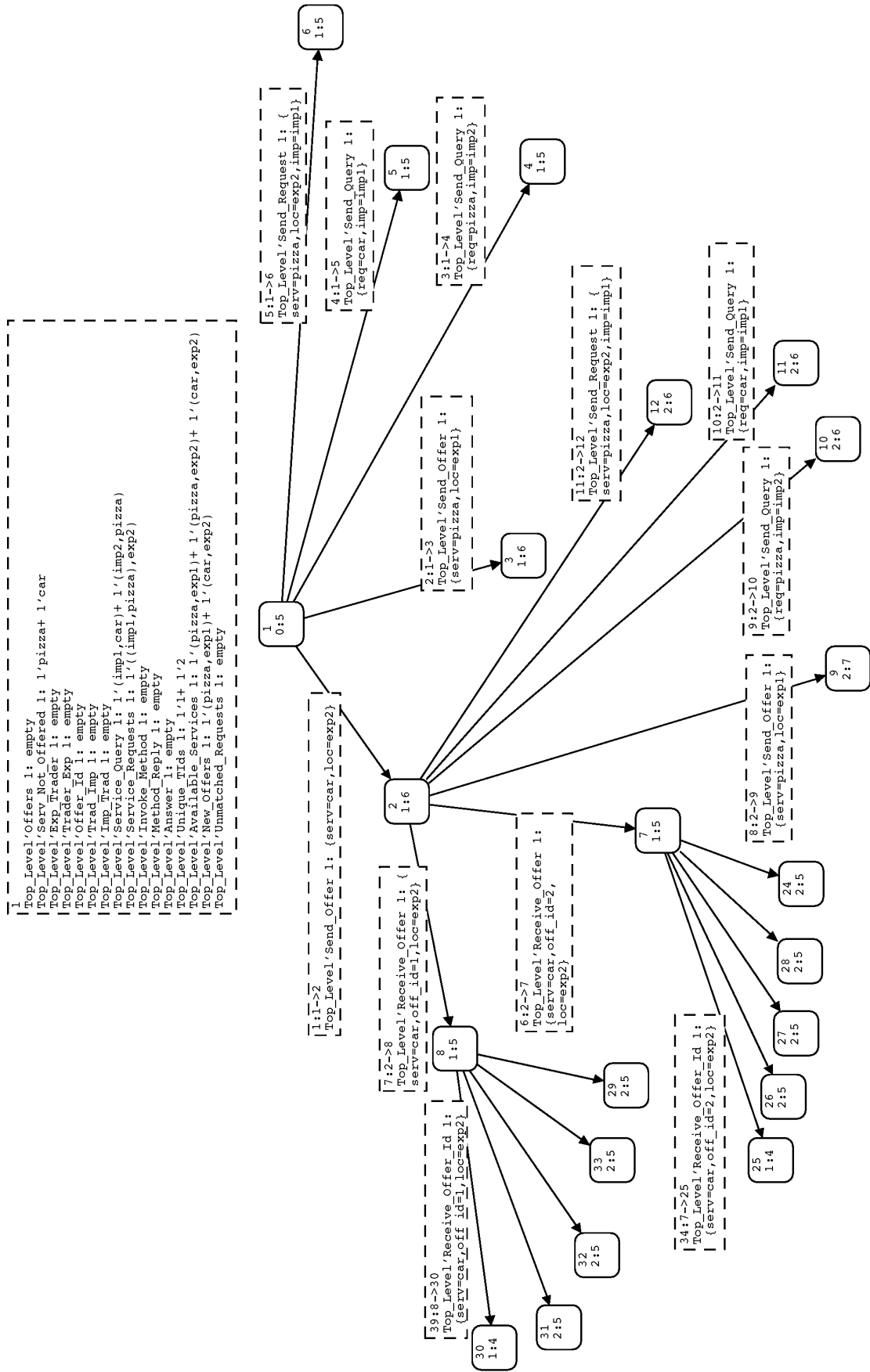


Fig. 5.12: Top-Level Model Occurrence Graph

1, each of which has a binding element associated with it and connects with a new node. This represents the firing of a transition, the bindings for the variables used to enable the transition and the new marking of the system after the transition fired.

Having 5 arcs from node 1 is expected since there were three transitions that were enabled in the initial marking and two of these transitions (**Send_Query** and **Send_Offer**) each had two possible bindings.

The arc from node 1 to 2 has a binding element indicating that **Send_Offer** was fired with the binding `serv=car` and `loc=exp2`. From node 2, there are another 6 arcs representing the possible transitions that may occur with a specific binding from that marking. Of most interest for this discussion are the arcs to nodes 7 and 8 which both represent the occurrence of **Receive_Offer** but with different bindings.

The binding used to get from node 2 to node 7 is `serv=car, off_id=2` whereas the binding used to get to node 8 is `serv=car, off_id=1`. It is easy to see that the only difference between the two markings is the value bound to `off_id`. Since the resultant markings are different, the OG tool created a node for each marking. From each of these markings another 5 nodes are connected. This continues until the graph terminates at two dead nodes that represent both of these possible values for `off_id` matched with the offers. Since `off_id` represents a unique but arbitrary number, we do not care what value it takes in the OG. A technique that ignores the value assigned to `off_id` is required.

5.4.5.3 Occurrence Graphs with Equivalences

In order to generate an OG with equivalences using the OEOS tool, the modeller must define two functions; one that detects equivalent markings and another that detects equivalent bindings. In this example, they are named `EquivMark()` and `EquivBE()` and are shown in figures 5.14 and 5.15 respectively. Each of these functions take a pair of markings (`n1` or `n2`) or bindings and return a boolean value that represents whether the two markings or bindings are equivalent. `EquivMark()` requires a utility projection function that maps out unwanted tuples from a marking as shown in figure 5.13.

```

(* Function used in the OEOS Occurrence Graph Creation. *)
(* Implementation of a utility function which maps out *)
(* the Offer_Id in an Offer_Space or Exp_Return. *)

fun Mapout_Offer_Id Offer_ms =
  let
    (* Convert the multi-set into a list *)
    val Offer_Id_list = ms_to_list Offer_ms
    (* MapoutOne removes Offer_Id from an Offer_Space or *)
    (* Exp_Return using a pattern match *)
    fun MapoutOne ((Service,Location),_) = (Service,Location)
  in
    (* mapout and convert back to multi-set *)
    list_to_ms (map MapoutOne Offer_Id_list)
  end;

```

Fig. 5.13: A MapOut function used by the OEOS tool

The `Mapout_Offer_Id()` function converts a multi-set parameter to a temporary list. It then defines the `MapoutOne()` projection function that takes a tuple of the form $((x,y),z)$ as an input parameter and returns (x,y) , mapping out the z component of the tuple. `Mapout_Offer_Id()` applies `MapoutOne()` to all elements in the temporary list and then converts the temporary list back into a multi-set.

For example, if the multiset $1'((\text{pizza}, \text{expl}), 2)$ is passed into `Mapout_Offer_Id`, then the function returns $1'(\text{pizza}, \text{expl})$. Similarly, if the function was passed $1'((\text{pizza}, \text{expl}), 1)$ the same result is returned where `off_id` is mapped out of the tuple.

`Mapout_Offer_Id()` is used by `EquivMark()` to map out the `off_id` from places that have `off_id` as part of their colour set. This allows `EquivMark()` to detect markings that are the same marking except for the value of `off_id`.

For other places that do not contain an `off_id` element, `EquivMark()` is required to directly compare the marking of the place in each node using the following notation:

```
(Mark.Top_Level'Offers 1 n1) == (Mark.Top_Level'Offers 1 n2)
```

```
(** Equivalent Marking Function **)
fun EquivMark (n1,n2) =
(** Marking of Places in the Exporter Objects **)
(Mark.Top_Level'New_Offers 1 n1) == (Mark.Top_Level'New_Offers 1 n2) andalso
(Mark.Top_Level'Available_Services 1 n1) == (Mark.Top_Level'Available_Services 1 n2)
andalso
Mapout_Offer_Id(Mark.Top_Level'Offer_Id 1 n1) ==
Mapout_Offer_Id(Mark.Top_Level'Offer_Id 1 n2) andalso

(** Marking of Places between the Exporters and the Trader **)
(Mark.Top_Level'Exp_Trader 1 n1) == (Mark.Top_Level'Exp_Trader 1 n2) andalso
Mapout_Offer_Id(Mark.Top_Level'Trader_Exp 1 n1) ==
Mapout_Offer_Id(Mark.Top_Level'Trader_Exp 1 n2) andalso

(** Marking of places within the Trader **)
Mapout_Offer_Id(Mark.Top_Level'Offers 1 n1) ==
Mapout_Offer_Id(Mark.Top_Level'Offers 1 n2) andalso
(Mark.Top_Level'Serv_Not_Offered 1 n1) == (Mark.Top_Level'Serv_Not_Offered 1 n2)
andalso

(** Marking of Places between the Trader and the Importers **)
(Mark.Top_Level'Imp_Trad 1 n1) == (Mark.Top_Level'Imp_Trad 1 n2) andalso
(Mark.Top_Level'Trad_Imp 1 n1) == (Mark.Top_Level'Trad_Imp 1 n2) andalso

(** Marking of places within the Importer **)
(Mark.Top_Level'Service_Query 1 n1) == (Mark.Top_Level'Service_Query 1 n2) andalso
(Mark.Top_Level'Service_Requests 1 n1) == (Mark.Top_Level'Service_Requests 1 n2)
andalso
(Mark.Top_Level'Unmatched_Requests 1 n1) == (Mark.Top_Level'Unmatched_Requests 1 n2)
andalso
(Mark.Top_Level'Answer 1 n1) == (Mark.Top_Level'Answer 1 n2) andalso

(** Marking of Places between the Importers and the Exporters **)
(Mark.Top_Level'Invoke_Method 1 n1) == (Mark.Top_Level'Invoke_Method 1 n2) andalso
(Mark.Top_Level'Method_Reply 1 n1) == (Mark.Top_Level'Method_Reply 1 n2);
```

Fig. 5.14: The Equivalent Marking function used by the OEOS tool

`Mark` refers to a data structure that has been created by the OEOS tool that contains the marking of the node. For example, `EquivMark()` checks instance 1 (the only instance) of the `Offers`

place located on the `Top_Level` page. If the marking of the place in node `n1` is equal to the marking of the place in node `n2`, then the result of the test for that place is `TRUE`.

`EquivMark()` performs a series of tests on each place in the marking and performs a logical AND on the results to determine if both of the markings in `n1` and `n2` are equivalent.

The `EquivBE()` function shown in figure 5.15 performs a similar test to `EquivMark()`, except that it determines whether two binding elements that are input parameters to the function are the same. Two bindings are considered equivalent if they have the same transition and the values for the bindings are the same. When a tuple is to be ignored, it is matched with an underscore character which represents a match with any value. For example,

```

(** Equivalent Binding Element Function **)(** Exporter Transitions **)
fun EquivBE (Bind.Top_Level'Send_Offer (1,{serv=s1, loc=l1}),
            Bind.Top_Level'Send_Offer (1,{serv=s2, loc=l2})) =
    (s1=s2) andalso (l1=l2)

| EquivBE (Bind.Top_Level'Receive_Offer_Id (1,{serv=s1, loc=l1, off_id=_}),
          Bind.Top_Level'Receive_Offer_Id (1,{serv=s2, loc=l2, off_id=_})) =
    (s1=s2) andalso (l1=l2)

| EquivBE (Bind.Top_Level'Generic_Method (1,{imp=i1, serv=s1, loc=l1}),
          Bind.Top_Level'Generic_Method (1,{imp=i2, serv=s2, loc=l2})) =
    (i1=i2) andalso (s1=s2) andalso (l1=l2)

(** Trader Transitions **)
| EquivBE (Bind.Top_Level'Receive_Offer (1,{serv=s1, loc=l1, off_id=_}),
          Bind.Top_Level'Receive_Offer (1,{serv=s2, loc=l2, off_id=_})) =
    (s1=s2) andalso (l1=l2)

| EquivBE (Bind.Top_Level'Matched_Query (1,{imp=i1, req=r1, loc=l1, off_id=_}),
          Bind.Top_Level'Matched_Query (1,{imp=i2, req=r2, loc=l2, off_id=_})) =
    (i1=i2) andalso (r1=r2) andalso (l1=l2)

| EquivBE (Bind.Top_Level'No_Match (1,{imp=i1, req=r1}),
          Bind.Top_Level'No_Match (1,{imp=i2, req=r2})) =
    (i1=i2) andalso (r1=r2)

(** Importer Transitions **)
| EquivBE (Bind.Top_Level'Send_Query (1,{imp=i1, req=r1}),
          Bind.Top_Level'Send_Query (1,{imp=i2, req=r2})) =
    (i1=i2) andalso (r1=r2)

| EquivBE (Bind.Top_Level'Query_Reply (1,{imp=i1, serv=s1, loc=l1}),
          Bind.Top_Level'Query_Reply (1,{imp=i2, serv=s2, loc=l2})) =
    (i1=i2) andalso (s1=s2) andalso (l1=l2)

| EquivBE (Bind.Top_Level'Send_Request (1,{imp=i1, serv=s1, loc=l1}),
          Bind.Top_Level'Send_Request (1,{imp=i2, serv=s2, loc=l2})) =
    (i1=i2) andalso (s1=s2) andalso (l1=l2)

| EquivBE (Bind.Top_Level'Service_Reply (1,{imp=i1, serv=s1}),
          Bind.Top_Level'Service_Reply (1,{imp=i2, serv=s2})) =
    (i1=i2) andalso (s1=s2)

| EquivBE(_,_) = false;

```

Fig. 5.15: The Equivalent Binding Element function used by the OEOS tool

```

EquivBE (Bind.Top_Level'Receive_Offer_Id (1,{serv=s1, loc=l1, off_id=_}),
        Bind.Top_Level'Receive_Offer_Id (1,{serv=s2, loc=l2, off_id=_})) =
    (s1=s2) andalso (l1=l2)

```

The function compares the value for the bindings of `serv`, `loc` and `off_id`. Since `off_id` is to be mapped out using `'_'`, it may be matched with any value. The other variables are compared and the function evaluates to TRUE if they are the same. `EquipvBind()` performs a logical OR on these comparisons since it is searching for at least one match for a pair of bindings.

Thus, `EquipvMark()` is able to detect markings that are equivalent in all aspects except for a specific identifier that is mapped out of consideration. Similarly, `EquipvBE()` is able to detect binding elements that are equivalent except for an identifier that is ignored in the comparison.

Having defined these functions which are required for determining an OG with Equivalences, they must be registered with the OE SML routines. This is accomplished by executing the command shown in figure 5.16.

```

OESet.Equivalence {
  Mark = EquipvMark,
  Bind = EquipvBE,
  Spec = OESpec};

```

Fig. 5.16: Command used to set Equivalence Functions for the OEOS tool

Having registered the Equivalence relations, the final step is to generate the OG which is accomplished by executing:

```
CalculateOEGraph();
```

The results of the analysis using the OEOS tool are shown in Table 2.

OG
Nodes=2704
Arcs=9724
Seconds=154
FULL

Table 2 – OEOS Generation statistics from the Top-Level CPN

Figure 5.17 shows a small portion of the OG generated using Equivalence class. The OG shown in figure 5.12 contains 2 arcs associated with the occurrence of **Receive_Offer**, as explained in section 5.4.5.2. By contrast, the OG shown in figure 5.17 contains a single arc connecting nodes 2 and 7 which represents both of the possible binding values. In this example, the technique has removed an entire branch of nodes from the OG, thereby significantly reducing the OG's size. This is because all successor nodes in the branch can be represented by existing nodes in the OG.

Executing `ListDeadMarkings()` on the OG with Equivalence Classes resulted in the following node list: [2525, 2684, 2685, 2704]. This is as expected since these four dead nodes correspond with the permutations of successful Importer Query matches.

When comparing the markings of the dead nodes obtained through OGA and OEOS analysis (figures 5.10 and 5.18), it is evident that the markings are identical. This indicates that the OEOS

tool was able to detect the same property (terminal marking) as the standard OGA tool but was also able to reduce the OG size by 48%, a significant reduction.

```

2685
Top_Level'Offers 1: 1'((pizza,exp1),2)+ 1'((car,exp2),1)
Top_Level'Serv_Not_Offered 1: empty
Top_Level'Exp_Trader 1: empty
Top_Level'Trader_Exp 1: empty
Top_Level'Offer_Id 1: 1'((pizza,exp1),2)+ 1'((car,exp2),1)
Top_Level'Trad_Imp 1: empty
Top_Level'Imp_Trad 1: empty
Top_Level'Service_Query 1: empty
Top_Level'Service_Requests 1: empty
Top_Level'Invoke_Method 1: empty
Top_Level'Method_Reply 1: empty
Top_Level'Answer 1: 1'(imp1,pizza)+ 1'(imp2,pizza)
Top_Level'Unique_TIDs 1: empty
Top_Level'Available_Services 1: 1'(pizza,exp1)+ 1'(pizza,exp2)+ 1'(car,exp2)
Top_Level'New_Offers 1: empty
Top_Level'Unmatched_Requests 1: 1'((imp1,no_serv),no_loc)

```

Fig. 5.18: Marking of node 2685 from OEOS analysis of the Top-Level CPN

5.5 Discussion of the Model

The model contains a number of interesting behaviours including concurrency, non-determinism and variables. An understanding of these aspects provides greater insight into the benefits of modelling systems with CPNs.

5.5.1 Concurrency

There are two levels of concurrency evident in the model. The first is inter-object concurrency, where different objects contain transitions that are concurrently enabled. This represents autonomous objects that are executing independently. An example of inter-object concurrency is seen at the initial state of the model, where **Send_Offer** and **Send_Query** transitions are both enabled. Either of these transitions may be fired by the simulator, where a transition is chosen by the simulator from the complete set of enabled transitions.

The second level of concurrency is intra-object concurrency, where an object has multiple transitions enabled. An example of this is the `imp1` Importer object, where **Send_Query** and **Send_Request** are both enabled. This represents threads of control within the object.

5.5.2 Variables on Arcs

The model uses variables on arcs to provide a flexible means of modelling the flow of data within the system. For example, the input arc to the **Send_Offer** transition is inscribed with `(serv, loc)`. This means that the `serv` and `loc` variables can be bound to any legal value in their respective colour sets (`Service` and `Location`). Since there are two tokens in *Offers*, there are two possibilities for bindings of `serv` and `loc` that result in the enabling of **Send_Offer**.

```
serv = pizza, loc = exp1 or  
serv = car, loc = exp2
```

Other bindings of `serv` and `loc`, such as

```
serv=car, loc = exp1 or  
serv = pizza, loc = exp2
```

do not result in **Send_Offer** being enabled since a token matching those bindings does not exist in *Offers*.

The model uses variables throughout which has the effect of creating a smaller, more compact model. This is accomplished by effectively folding different bindings of variables on transitions into one transition, rather than using a different transitions for all possible bindings as is necessary when variables are not used.

5.5.3 Non-determinism

In the top-level model, non-determinism is used when associating an `Offer_Id` with exported offers at the **Receive_Offer** transition. When modelling systems with CPNs, the ability to use non-determinism is very powerful since it allows the modeller to abstract away from implementation details and concentrate on high-level behaviour.

Using non-determinism with colour sets that contain many elements results in a huge increase in OG size. Depending upon the model, it may be possible to map out the non-deterministic part of the colour set to reduce the OG size using the OEOS tool.

The Trader may be able to match both import requests, or it may be unable to match the import request from `imp1`, `imp2` or both. This is a total of four possible outcomes from the object interactions. When an offer is exported to the Trader, one of the two unique transaction identifier is arbitrarily associated with the Offer. Thus, there are two possible outcomes for the binding and therefore a total of eight possible dead nodes. The second set of nodes differs from the first in only one respect, that being the transaction identifier associated with the exported offers.

5.6 Summary

In this chapter CPNs have been introduced with the aid of a simple example. The basic constructs, dynamic behaviour and analysis techniques of CPNs were described. Using a more complicated example of Trader/Importer/Exporter interaction, Design/CPN was introduced as a tool that may be used for the creation, modification, simulation and formal analysis of CPNs. The model was analysed using the OGA and OEOS tools with the OEOS tool being used to reduce the OG size significantly.

Chapter 6

Modelling the Trading Environment with Coloured Petri Nets

This chapter describes how the ODP Trader and its Trading Environment were modelled using Coloured Petri Nets [66] and the Design/CPN• [83] software tool. The chapter is divided into two parts. Part I discusses the major modelling decisions and assumptions associated with creating the model. It also explains some modifications that have been made to the Trader Interworking protocol and identifies the objects in the model. In Part II, a Design/CPN model of the Trader and its environment which includes the modified interworking protocol is presented. Initially, the Colour Sets, variables and functions used throughout the model are described. The model is presented in a top-down manner, where CPN models of objects in the system are individually described. The model is based upon the Trader's computational and information viewpoint specifications included in the Trading standard [16].

Part I – Modelling Issues

6.1 Scope

The CPN model presented in this chapter follows the Trading Standard [16] specification for the Information and Computational viewpoints (sections 7 and 8 of the Trading Standard respectively), except where noted otherwise.

The scope of the model is restricted to modelling the Lookup and Register interface functionality of the Trader, specified in sections 8.5.1 and 8.5.3 respectively of the RM-ODP Trading Standard [16].

Only the Register interface's `export` method (specified in section 8.5.3.1 of the Trading Standard [16]) is included in the CPN model. The other Register interface methods (`modify`, `withdraw`, `describe` and `resolve`) are not included in the model since they are not required for analysis of the Trader's interworking protocol.

The model does not attempt to reproduce the constraint/preference language which is specified in Annex C of the RM-ODP Trading Standard [16]. Instead, the model assumes that this functionality can be provided by the OfferSpace object which maintains the offer space (see section 6.6.4) for Traders in the system.

The model does not include the functionality of the RM-ODP Trader's Administration, Link or Proxy interfaces. The investigation presented in this thesis does not aim to verify the functionality of these interfaces, but focuses instead on the Lookup and Register interfaces.

The model does not include the functionality of the Iterator interfaces as described in section 3.4.5. This is because the Iterator interfaces do not provide functionality that is essential for the Trader to provide a basic service location function which is capable of interworking.

Exception handling associated with Interfaces specified in the Trading standard has not included in the model although some error handling/reporting has been included, such as security and parameter errors.

6.2 Petri Nets and Objects

A survey on the topic of Petri Nets and objects has been performed by Bastide [87] which identifies two main approaches to combining the concepts of Object-Oriented and Petri Nets.

The first method, known as "Objects inside Petri Nets" considers tokens to be objects, where each token is an instance of the object class. In order to perform functions (or methods) on objects, transitions occur where inscriptions on outward arcs may be evaluated using object method operations. Figure 6.1 [87] illustrates this concept, where objects of different colour (residing in Places A and B) are bound to variables `x` and `y`. When transition T1 occurs, the outgoing arcs are evaluated, with the result of a method invocation on object `x` (using parameter `y`) being placed into C which is of colour `Colour1`. Output arcs indicate the flow of objects throughout the system. This philosophy allows objects (tokens) to be created and destroyed dynamically which occurs in real dynamic object-based systems.

This approach is object based, where tokens are objects with their internal data structure defined by their colour. Transitions associated with a place provide the methods which may operate on the data within the object (token). The net structure determines the ordering of these methods, hence defining the system's functionality.

The second method, known as "Petri Nets inside Objects" (shown in figure 6.2), is where a Petri net is used to model the internal workings of an object, and the inter-object communication of an Object-Based system. In this modelling methodology, a token is used to hold data (depending upon its colour) and the marking of a net represents the internal state of an object. Using this

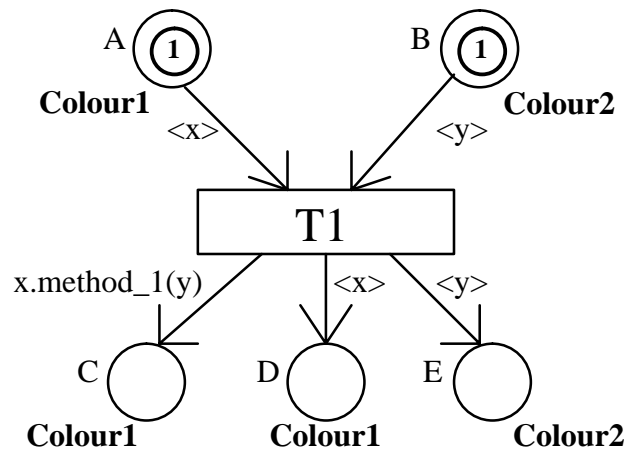


Fig. 6.1: The “Objects inside Petri Nets” concept

methodology, it is not possible to dynamically create objects, as they are part of the CPN structure. A clearly defined interface may be created which indicates the “methods” which an object may perform and data hiding is easily represented. It is this methodology which has been used to model the ODP Trader using Design/CPN.

In figure 6.2, multiple objects (Objects 1 and 3) are described by Net2, where each object exists as its own instance of Net2. Thus, Net2 defines the behaviour of the objects whilst the marking of each Net2 instance defines the state of an object.

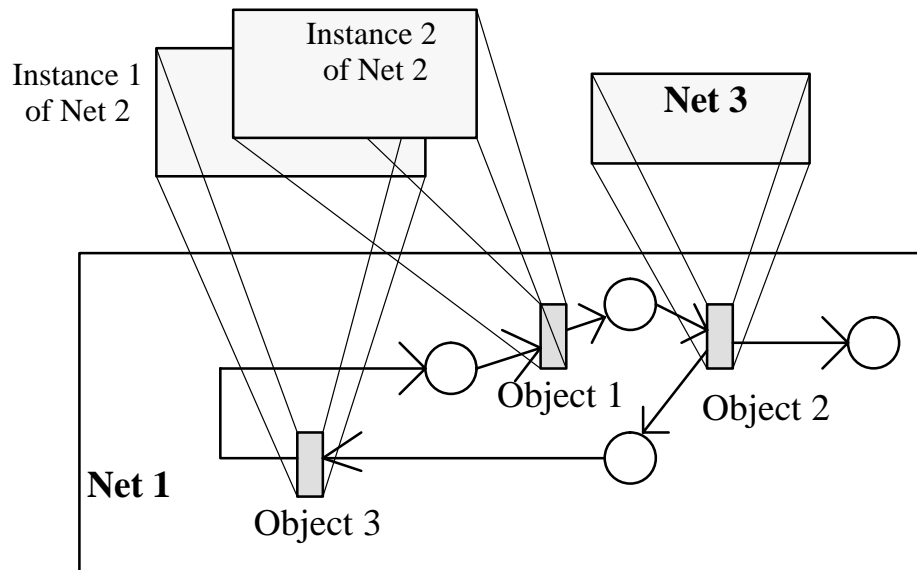


Fig. 6.2: The “Petri nets inside objects” concept

A discussion of different hierarchy abstractions (such as substitution transitions which are used in Design/CPN) and their impact on modelling objects in systems is presented in [82]. It includes a discussion on the use of Petri Nets for modelling the client–server paradigm which is commonly used by objects when interacting.

In [81], Lakos includes a survey of multiple Object–Oriented Petri Net formalisms. The discussion refers to a number of High–level Petri Net formalisms that may be used to model object–oriented systems including Design/CPN, which is used in this project.

6.3 Modelling Assumptions

6.3.1 Static Link and Trader attributes

The model assumes that the Trader and link attributes are static throughout the processing of a Query operation. This assumption means that if Trader or link attributes are altered during the processing of a Query, the new values would not affect the processing of transactions currently being processed. With this assumption, the model must ensure that the Trader and link attributes are only accessible in an atomic manner. Assuming static Trader and Link attributes, there is no need to include in the model, interface methods which are used to alter Trader and Link attributes.

6.3.2 RPC Messaging

Objects in the system are assumed to communicate using an RPC-like messaging system, where a request always blocks until it receives a reply, as described in section 3.2.3.3. When a message is sent, objects are able to continue internal processing of non-blocking threads. Coupled with a reliable communications medium, there is no need for a time-out mechanism to be added to the model, since it is assumed that all responses are successfully transmitted to the object which is blocked, waiting for the response.

6.3.3 Type Repository object

In the RM-ODP model, types are used to define service interfaces and properties. In RM-ODP systems, there is a need for a Type Repository object which manages and maintains types which have been defined throughout the RM-ODP system. This allows type checking during trading and also when establishing bindings. The Type repository is also responsible for maintaining a type hierarchy to allow sub-typing to be described[7].

The functionality of the Type Repository is a research area unto itself [23] and has recently been included as Annex D of the Trading Standard [16]. Thus, it was decided that the model would abstract away from the requirement of checking types during processing of Trading Queries. Instead, it was assumed that all types used throughout the model were legal. This greatly simplifies the model and reduces the size of the state space since less external messaging is required by entities in the system (as will be shown in Chapter 7).

6.4 Modelling Decisions

When modelling a complex system such as the Trader, there are significant modelling decisions that must be made. They include deciding upon a representation for the communications medium and how to model multiple object instantiation and threading within those objects.

6.4.1 The Communications Medium

The model contains multiple objects that communicate using message passing which is RPC-like, as specified in the RM-ODP standard (see section 3.2.3.3). The communications medium is represented by a shared place which contains message tokens while they are in transit between objects. This is a simple model of a communications medium which has the following properties:

- messages are neither lost nor duplicated,
- ordering of messages is unimportant (overtaking is allowed),
- there is no mutilation (corruption) of messages.

Having modelled the communications medium, the next decision was to determine a suitable object connection topology which allows multiple objects to communicate in a simple manner. Any solution should allow all objects to communicate between each other, even if this is not specifically required by the Trading application. This ensures that the model of the Communications medium truly reflects that observed in an OOBDS. For the purposes of readability [66], it was also important to minimise the number of places required and crossed arcs in the CPN.

The solution adopted for use in the Trader model is a single shared place which represents the communications medium and allows all objects in the system to communicate with each other. Objects are connected to the shared place and may write or read messages using the place. To allow messages to be sent between objects, sender and receiver information must be included in the message. This is accomplished by the *Addresses* colour set which is defined in section 6.9 as a tuple with two elements of colour *Id* (Colour set *Id* contains all of the possible unique object identifiers). It was decided to use a shared place topology as shown in figure 6.3, where two objects can communicate using place *A*. Each of the objects have two methods which are separated from the communications medium *A* by Inter-Object interfaces.

This solution does not remove the possibility of crossed arcs in the CPN, but it does provide a clean interface for inter-object communication and allows the easy addition of object instances to the model, as will be seen in section 6.4.3.

An alternative is to provide each object and method with its own input (I) and output (O) places as shown in figure 6.4. It shows a mesh of connections which allows all outputs from the methods to feed in to the inputs of each method, thereby allowing inter-object communications. Note that each of the connections would require an additional transition and the readability of this solution is greatly reduced.

By using an additional interface layer as illustrated in figure 6.3, the model provides the same functionality but is greatly improved for the following reasons:

- improved readability since there is less clutter from additional transitions and arcs connecting all method outputs to method inputs,

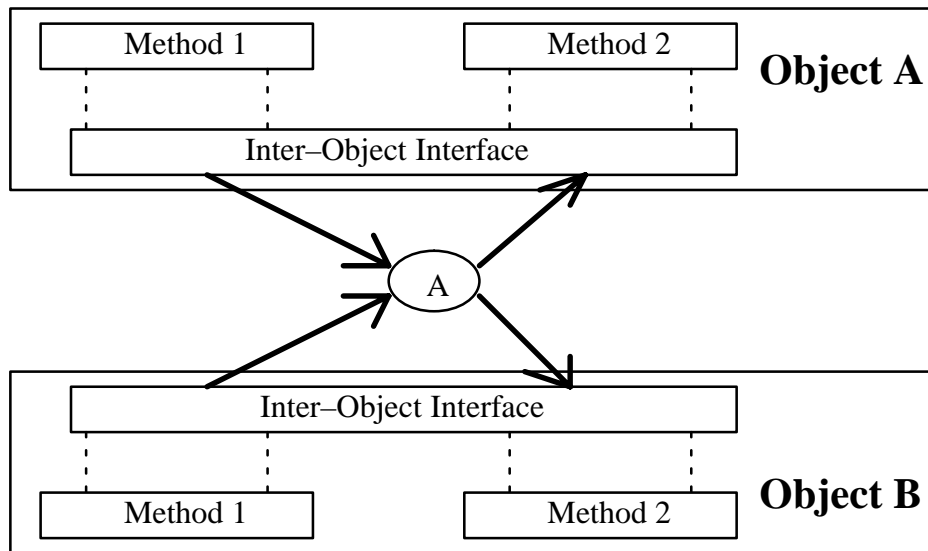


Fig. 6.3: 2 Objects, 4 methods and 1 common place

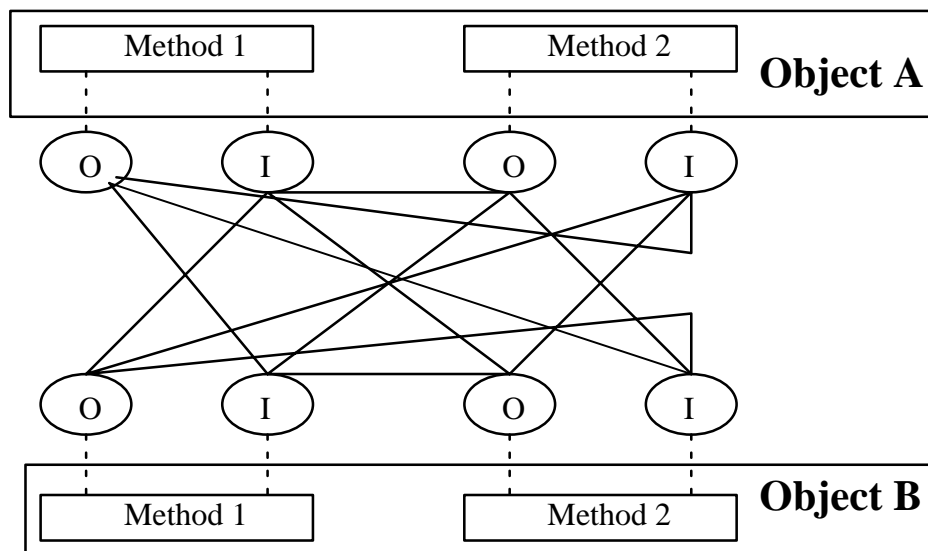


Fig. 6.4: 2 Objects, 4 methods and a mesh of inter-connecting arcs

- easy addition of objects to the model since there is a single common connection point (the **Comms_Medium**) as described in section 6.4.3. This is important in the Trader model since analysis of the Trader's interworking protocol in Chapter 7 requires multiple Trader instances to be included in some scenarios.

6.4.2 Message Format

In its most basic form, a message requires a unique destination address and provision for payload data. It is also useful to include the unique address of the sender so that the receiver can reply to messages it receives. Thus, a message has the following form:

```
(** Message is defined as a product of addressing information and the data contained
within the message **)
color Message = product Addresses*Data;
```

where

```
(** Addressing mechanism that requires a Sender and Receiver object
    identifier **)
color Addresses = product Id*Id;
```

This provides a mechanism to send and receive messages based upon unique Sender and Receiver object identifiers. The specific format of the payload data used in the model will be discussed later in section 6.9.

6.4.3 Multiple Object Instantiation

There are many objects in the system and as scenarios change, more objects may need to be added. Multiple instances of objects are represented by hierarchical substitution which provides a simple way to re-use the CPN page which models the object although it only allows a static number of instances during a scenario.

Objects have been modelled using interfaces that contain unique object identifiers. These identifiers make it possible to distinguish between instances of the same object type. When modelling interworking Traders, it is necessary to have up to four Trader instances for the purpose of protocol verification (see section 7.7). Each of the Traders must be individually addressable to allow message passing.

An alternative mechanism to using a substitution transition for each object instance is to append an extra dimension to all tuples in the model, effectively folding all instances of objects into one CPN [80]. This additional element of a tuple represents an object's unique instance identifier and removes the need for multiple substitution transitions at the top-most level. This approach however, reduces the overall impact of the top level CPN since it no longer graphically illustrates the number of instances used within the model, only the object types which are instantiated.

6.4.4 Threading within Objects

Threading within objects is modelled by appending a transaction identifier to appropriate colour sets in the model. This allows Trader instances to have multiple threads of execution operating concurrently. This is the same as Moldt's [80] approach to modelling multiple instances as discussed in section 6.4.3, except in this case, the appended tuple element represents distinct threads of execution, thereby allowing multiple threads of execution to be represented by one CPN.

The model must support multiple threading if it is to be used for the verification of interworking Traders. When a Query is propagated to linked Traders, it may be re-sent to the source Trader via another Trader which is linked to it. The source Trader must be capable of servicing both the original Query and the incoming request to avoid deadlock as described in section 7.5.2.

Since multiple Queries can be processed concurrently, it is important to ensure consistency between data items throughout the model. If not, the state of the model will become corrupted, since tokens associated with different threads of execution can be used by other threads that are operating concurrently.

There are two reasons why it is important for the Trader to be able to concurrently process requests. The most important reason is to avoid deadlock when Traders interwork as shown in figure 6.5. During interworking, it is possible for Query requests to be propagated back to the source Trader (i.e. from **T1** to **T2** and then back to **T1**). Since the source Trader (**T1**) is already processing a request, it must concurrently service the propagated request from **T2**. This is because **T2** blocks waiting for a response to the Query which was re-sent to **T1**. Thus, the source Trader (**T1**) must be capable of servicing the duplicate incoming Query, or it will never obtain a response from **T2** which is blocked, waiting for **T1**'s reply to the duplicate Query, thereby causing a deadlock

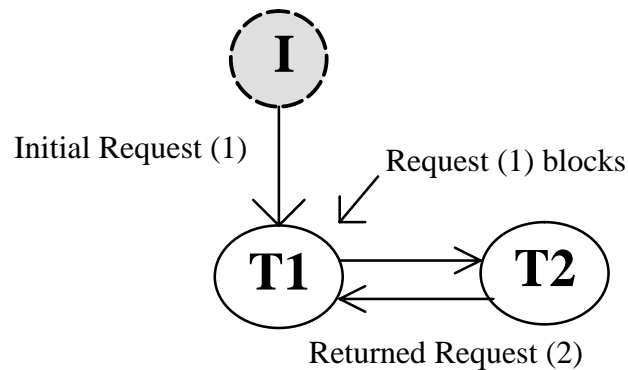


Fig. 6.5: Trader blocks while it processes Initial Request

The other reason for concurrent servicing of Queries is more pragmatic. That is, for commercial implementations of a Trader, performance concerns dictate that Queries must be processed as soon as possible. This can be achieved by using a multi-threaded implementation that services Queries as they arrive.

6.4.5 Link pre-fetching

In the case of a merged follow policy equal to `if_no_local`, a Query is only propagated if the local search of the OfferSpace object does not result in at least one matching Offer. In order to increase the Trader's internal concurrency, it was decided that the Trader should obtain its links whilst performing a local search of the OfferSpace object. This may be considered a *pre-fetch* of links in anticipation of following them when the Trader fails to obtain a local match to the Query. If the Query is not required to be propagated, then the pre-fetched links are discarded.

6.5 Improved Interworking Protocol

There are a number of ambiguities and areas requiring optimisation in the Trading standard's specification of the interworking protocol [16] as described in section 3.4.6. The protocol ensures that infinite looping of Queries cannot occur but does not achieve this goal efficiently. In particular, the following aspects were identified for clarification:

- there is no need for a Query to be propagated back to the Trader from which it came (in the case of a bi-directional link).

- the standard does not prescribe what happens when a duplicate is detected. It only states that the Query is not “processed”. Does this mean that duplicates are to be ignored?
- the standard does not ensure deterministic traversal of all linked Traders. This means that under certain circumstances, a Query with the same parameters may return a different list of matching Offers.

Each of these points will be discussed individually in the following sections.

6.5.1 Redundant Forwarding of Queries

In the current Trading Standard, it is possible for Traders which are tightly coupled to forward redundant Queries between themselves until the `hop_count` reaches zero.

6.5.1.1 Example of Redundant forwarding

Consider figure 6.6 which shows an Importer (I) and two Traders (T1 and T2) which are connected via a bi-directional link (i.e. two uni-directional links in opposite directions). A Query with `hop_count=4` is sent by I to T1 which forwards it to T2 with `hop_count=3`. The Query is received by T2 which re-sends the Query to T1 and also forwards it to T3. T2’s re-sending of the Query to T1 is redundant since the source Trader (T1) has already performed a local search for the Query (if possible according to policy constraints) and has already forwarded the Query to all of its legally linked Traders.

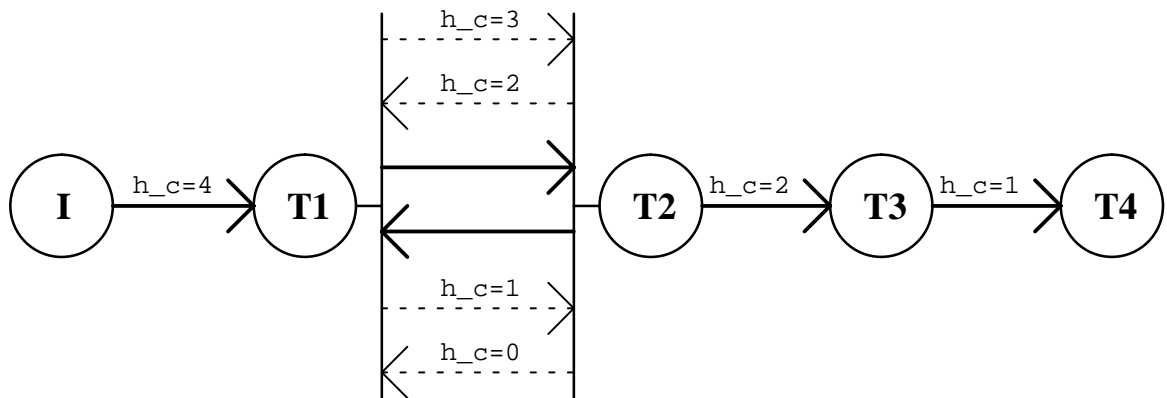


Fig. 6.6: Redundant Query propagation

Thus, re-sending a Query to its source Trader fails to improve the scope of the search and does not contribute to the search for matching offers (assuming that links are static). In fact, it causes Traders to become overloaded since they become burdened with processing Queries that they have already processed! This results in a large amount of wasted bandwidth through messaging and wasted processing time as shown by the example in figure 6.6. The example shows a Query which is “bounced” between T1 and T2 four times, three of which do not benefit the search yet consume Trader resources during processing. This example assumes that T1 does not detect duplicate Queries. If it can detect such Queries, the Query stops being propagated at T1 when `hop_count=2`, rather than when it reaches 0, a much better result.

6.5.1.2 Proposed Solution

The Trading Standard [16] recognises that this scenario may occur and states in section 8.2.8.1:

“Loops can occur. The most trivial example of this is where two previously disjoint trader spaces decide to join by exchanging links. This can result in the first trader propagating a query to the second and then having it returned immediately via the reverse link.”

The specification should be altered to make such a scenario impossible. Thus, it is proposed that the Trading standard [16], section 2.7.6 Link Follow Behaviour page 13, should be amended to include the following text shown in bold:

*“After searching its local offers in response to a query, a trader must decide whether to propagate the query along its links and, if so, what value for the link_follow_rule to pass on in the policies argument. **When testing a link to determine if it is to be used for propagating a query, a Trader shall not use any link which is connected to the external Trader which invoked the query on it. (i.e. a bi-directional link back to the Trader which invoked the query on it).**”*

This modification ensures that Traders do not attempt to propagate queries down bi-directional links to the Trader from which the Query was issued. Although bi-directional links are legal links, they are not considered *valid* for the purposes of Query propagation.

6.5.2 Actions to perform when a Duplicate Query is detected

The Trading standard does not explicitly specify the behaviour required when a Duplicate Query is detected. The Trading Standard [16], section 8.2.8.1 Link Traversal Control page 33, states:

“To avoid the unproductive revisiting of a particular trader while performing a query, a RequestId can be generated by the source trader for each query operation that it initiates for propagation to a target trader. The trader attribute of request_id_stem is used to form RequestId.”

*typedef sequence<octet> OctetSeq
attribute OctetSeq request_id_stem
(definition of request_id_stem as a sequence of octets)*

“A trader remembers the RequestId of all recent interworking query operations that it has been asked to perform. When an interworking query operation is received, the trader checks this history and only processes the query if it is the operation’s first appearance.”

This statement does not specify the behaviour required when it is not the operation’s first appearance (i.e. it is a duplicate). It specifies that the Trader does **not** *process* the operation, but the meaning of the term *process* has not been defined (does this only refer to local searching or does it include link searching?).

The Trading standard [16] also states:

“If the target trader does not support the use of the RequestId policy, the target trader need not process the RequestId but it must pass the RequestId onto the next linked Trader if the search propagates further.”

Possible interpretations of a Trader’s behaviour when a duplicate Query is received include:

1. do nothing except pass on the Query where appropriate, or
2. return an empty list of matching offers and forward the Query as appropriate.

This ambiguity allows the possibility of different behaviours occurring in different implementations of the Standard, and where each implementation claims compliance with the Trading Standard [16] Specification. In such a scenario, interworking between heterogenous Traders would be impossible.

In the Trader model presented in this Chapter, option 2 was selected as being most appropriate, since option 1 would result in a deadlock. This is because according to the communications model used by all objects in the model, all messages expect a resulting reply message. Using option 1 would result in a Trader blocking forever, waiting for a reply to a message which was discarded by a linked Trader. A suggestion for the clarification of this point will be presented in section 6.5.3.2.

6.5.3 Non-deterministic Traversal of Linked Traders

When a Query is forwarded to a Trader, the Importer specifies import policy parameters such as `hop_count` and `link_follow_rule` in order to limit the propagation of Queries to linked Traders. With these parameters, importers should be able to predict all of the paths a Query will take, and thus, the scope of the Query. With the current Trading standard [16], it is not possible to deterministically traverse the trading graph as will be described.

6.5.3.1 Example of Non-deterministic Traversal

To demonstrate the non-determinism of the current interworking protocol, consider the trading graph illustrated in figure 6.7. There are 4 traders that are linked, where T1 receives a Query from the Importer with the following import policy values:

```
hop_count=2
link_follow_policy=always
```

Initially, the Query is processed by T1 and forwarded to both T2 and T3. The ordering of the forwarding is not defined and it is possible that the Query reaches T2, is processed and then forwarded to T3 (indicated by italicised *h_c* values) before it is forwarded to T3 from T1 (path indicated by bold **h_c** values). Thus, it is possible for the Query to reach T3 via T2 with a `hop_count=0` prior to it reaching T3 from T1 with `hop_count=1`.

When this occurs using the current interworking protocol, the Query with `hop_count =1` is discarded since it is a duplicate of a previously received Query, even though it has a greater value

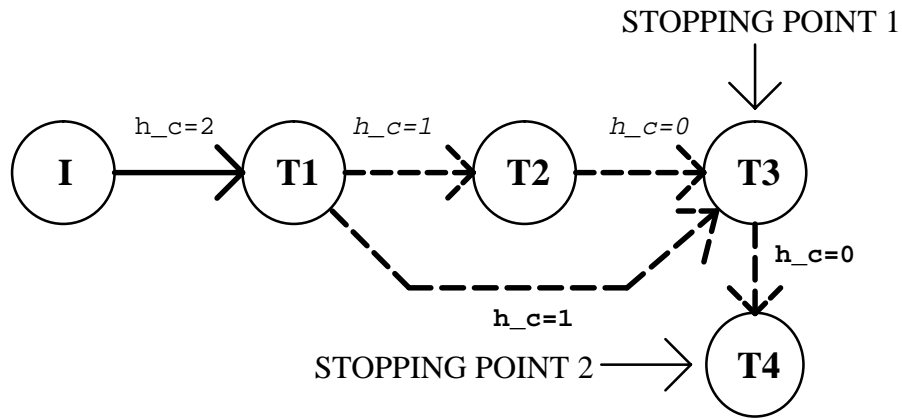


Fig. 6.7: Non-deterministic Query propagation

for `hop_count` than the previously serviced Query. Since the second Query is discarded because it is a duplicate, the Query never reaches T4 despite the fact that T4 is within the `hop_count` limit of 2 which was specified by the importer in its initial Query.

Alternatively, it is possible for the Query with `hop_count=1` to be the first to arrive at T3 and thus, is propagated to T4. In this case, when the Query with `hop_count=0` arrives at T3, it stops being propagated since its maximum `hop_count` has been reached. This example has illustrated the non-deterministic behaviour of the current interworking protocol.

6.5.3.2 Proposed solution

To ensure that Queries are allowed to traverse the entire trading graph as specified by their Import Policy, it is important to take a Query's `hop_count` into account when determining whether a request should be processed and/or forwarded to linked Traders. If a Query is found to be a duplicate, then it cannot be immediately discarded. It must be checked to determine whether it should be forwarded to linked Traders, even though local processing may not be necessary (depending upon whether the `unified_policy = if_no_local` in which case a local search must be performed to determine whether links are to be pursued.).

If a duplicate Query is received with a `hop_count` less than that of the previously received Query, then it offers no possibility of discovering additional matching offers. Thus, it is proposed that the Trading standard [16], Section 2.8.1 Link Traversal Control, page 14, should be amended to include the following text in bold:

*“When an interworking query operation is received, the trader checks this history and only processes the query if it is the operation’s first appearance ... and if the **hop_count of the new query is greater than the hop_count of the query which has already been received.**”*

When processing a duplicate query, a local search is only performed when the Query’s `merged_policy_options()= if_no_local`. In such a case, the Trader attempts to propagate the query to all linked traders.

If the Query is not required to be processed, the Trader shall return an empty list of matching offers.”

Thus, when a Query is processed, the `hop_count` of the Query must also be recorded as part of its `Request_Id`. This is because even if a duplicate is detected, it still may need to be forwarded to linked Traders if it has a greater `hop_count` limit than the Query which has already been processed.

6.5.4 Summary

The proposed amendments to the Trading standard provide the following:

- remove redundant messaging between Traders, responsible for non-productive messaging and increased Trader resource requirements,
- an indication of what the Trader is to do when it is **not** the Query's first appearance,
- inclusion of extra testing criteria to ensure that Queries that should be propagated to linked Traders are propagated, thereby providing deterministic traversal of the Trading offer space.

6.6 Objects in the System

In order to model an Object-based system, it is necessary to identify the entities that comprise the Open Object-based Distributed System (OOBDS). This defines the system at its highest level of abstraction, where autonomous entities operate concurrently. As part of this process, objects may be instantiated multiple times, as discussed in section 6.4.3. In order to allow communication between objects, a communications medium must be included in the model, as described in section 6.4.1.

Objects operate independently and only communicate through message passing. Thus, the internal state of objects is hidden from other objects and the only means of object interaction is via the interfaces which objects present to their external environment.

6.6.1 Trader

This object provides the Trading service described in section 3.4. The Computational viewpoint of the RM-ODP Trading standard [16] has been used as the basis for modelling. The Trader is used by both Importers and Exporters for the Trading of services. The Trader uses other objects in the system such as the Offer Space and Link Space objects to provide its functionality.

6.6.2 Exporter

The Exporter is an object which advertises a service to the Trader by invoking an *Export* operation on the Trader. The Export operation includes the service type, location and optional

parameters associated with the service. It is not strictly required that the Exporter provides the services which it has exported, since it may export services on behalf of other objects in the OOBDS.

6.6.3 Importer

This is the object which attempts to find a matching service provider for a specific service type, through the use of a Query method invocation on the Trader. The Query includes the service type and parameters associated with the desired service. In addition, the Query operation may include optional selection criteria which are used by the Trader to determine the “best” service from a set of matching services.

6.6.4 OfferSpace

The OfferSpace object maintains the Offers which have been exported to Traders. It is an autonomous object in the OOBDS and as such, may be accessed by any other object. In the case of a single Trader, the OfferSpace object has only one client. However, when modelling *Interworking* of Traders, the OfferSpace object may be utilised by multiple Traders. When objects provide services to other objects in the OOBDS, the need for a Trading function becomes apparent since the service consumer requires a mechanism to locate an appropriate service provider.

The OfferSpace object provides an `add_offer()` method which allows clients (in this case, Traders) to add offers to the database maintained by the OfferSpace object. It also provides a `get_offers()` method which allows clients (Traders) to retrieve matching offers stored in the OfferSpace object using its initial marking or the `add_offer()` method.

6.6.5 LinkSpace

The LinkSpace object is used to maintain Link information for Traders in the system. As with the OfferSpace object, the LinkSpace object provides its service to multiple Traders when Traders interwork. The LinkSpace object is not prescribed in the most recent Trading standard [16], but was part of the Trading Environment in both the 1994 and 1995 Trading Standards [10,14].

The LinkSpace object provides a `get_links()` method that Traders (acting as clients) can use to determine a Trader’s links. This is the only method required by its client objects when modelling the Trader’s operation since the mechanics of adding, deleting and modifying link information is not an area of Trader functionality within the scope of investigation. Thus, no methods for maintaining links are included in the model of the LinkSpace object.

6.7 Part I Summary

The CPN model will include a communications medium that allows all objects to pass messages to each other. All objects in the system shall have a unique object identifier. Some suggestions for improving the Trader interworking protocol have been presented. These suggestions will be included in the CPN model described in Part II of this chapter.

Part II – CPN Model of the Trading Environment

6.8 The Model Hierarchy

Design/CPN provides the opportunity to create a hierarchy of CPNs which as a whole, model the Trader. These CPNs can be used to represent objects which exist in the Trading Environment, or to hide complexity within the model. Figure 6.8 shows the Hierarchy page of the model (named Hierarchy#10010 where #10010 refers to the page number). Pages in the model are designated by nodes which are connected by arcs.

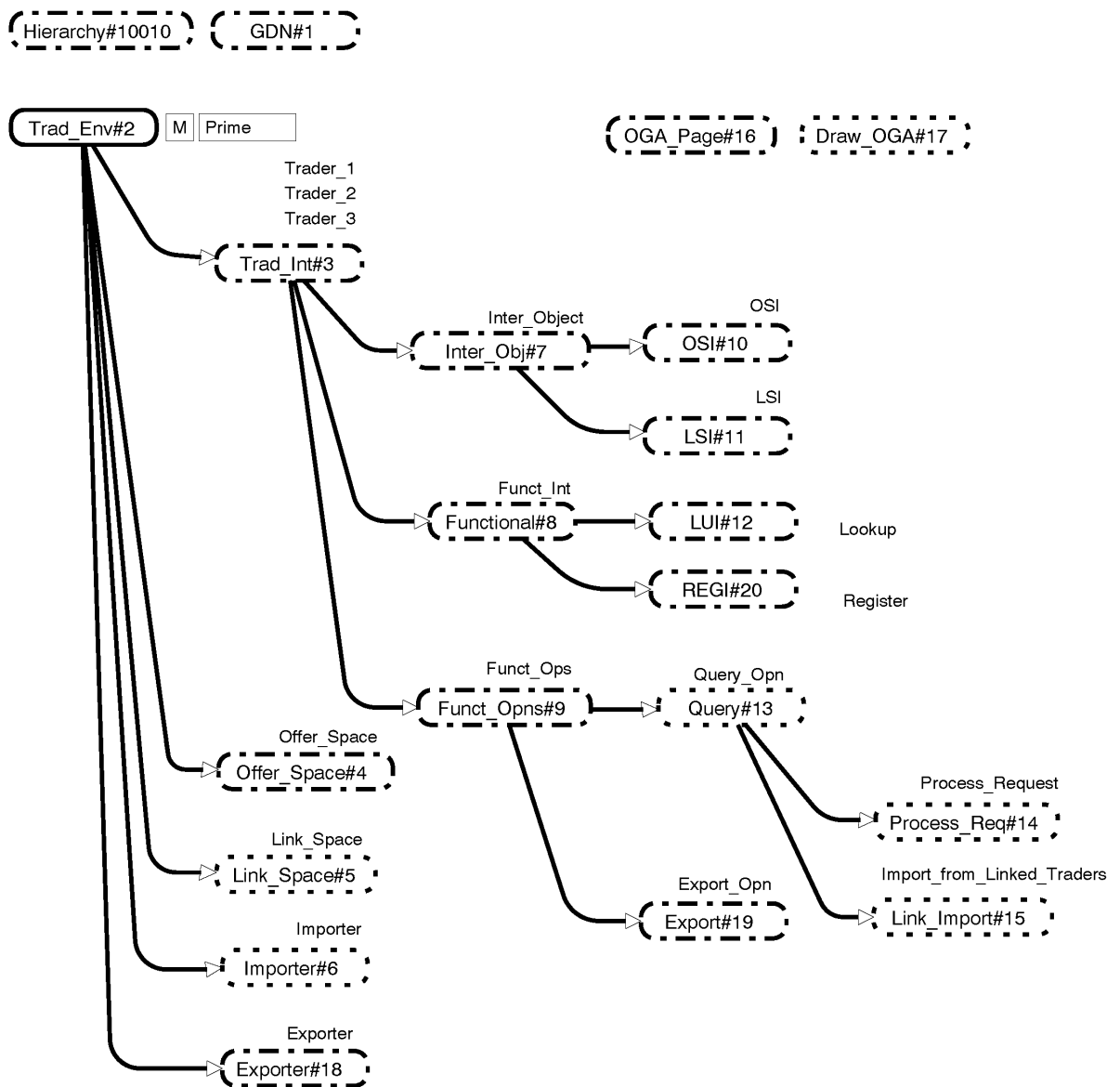


Fig. 6.8: Hierarchy page (Hierarchy#10010)

Relationships between pages are indicated by a directed acyclic graph. The hierarchy page indicates a “uses” relationship between pages and is created by Design/CPN upon request. The function of the pages is given below.

- GDN#1 is a page which contains the Global Declaration Node, where all colour sets (types) and variables are declared, and Standard ML function code is located.
- Trad_Env#2 is the top-most page of the model which defines a topology for communication between objects in the Trading Environment. It is also the *Prime* Page which is used by Design/CPN to designate it as the top-most page of the model (analogous to the `main()` function in a C program).
- Trad_Int#3 represents a Trading Interface within the Trading Environment. This is the interface which the Trader presents to other objects in the system. It may contain other interfaces and has been instantiated three times for modelling interworking Traders (Trader_1, Trader_2 and Trader_3).
- Inter-Obj#7 represents inter-object interfaces which are used by the Trader to communicate with other objects in the system, such as the OfferSpace and LinkSpace objects (using OSI#10 and LSI#11).
- OSI#10 and LSI#11 represent Interfaces to the OfferSpace and LinkSpace objects respectively. The interfaces are used by the Trader when communicating with the OfferSpace and LinkSpace objects.
- Functional#8 models the functional interfaces provided by the Trader. It contains a sub-page that models the Lookup Interface provided by the Trader. It also contains transitions that can be converted to substitution transitions, thereby allowing the other Trading functional interfaces (Register, Proxy, Link and Admin interfaces are discussed in section 3.4.4) to be added to the model at a later date.
- LUI#12 models the Lookup Interface provided by the Trader to importers. It models the interface used by the Trader when it accepts Query methods from Importers.
- REGI#20 models the Register Interface provided by the Trader to importers. It models the interface used by the Exporters when the export offers to the Trader.
- Funct_Opns#9 models the functional operations or *methods* provided by the Trader. It contains two substitution transitions that refer to CPN pages which model the Query (Query#13) and Register (Register#18) interfaces.

- Query#13 contains a model of the steps that occur when a Query operation is performed by the Trader. It references two sub-pages which are used to reduce this page's complexity (Process_Req#14 and Link_Import#15).
- Export#19 contains a model of the methods provided by the Register interface. This interface allows Exporters to add an offer to a Trader's Offer Space using the `Export ()` method.
- Process_Req#14 contains a CPN which describes the initial processing of a Query request, including checks for duplication of messages and the merging of Import Request and Trader policy information.
- Link_Import#15 is a page which describes the steps involved in forwarding Query requests to Linked Traders. It contains a CPN which models the forwarding of Query requests for both the "always" and "if_no_local" unified follow policies.
- Offer_Space#4 represents the OfferSpace object which is used by a Trader to manage offers which are submitted to the Trader by Exporters. It is a Generic Service object and may be used by any Trader to store Offers.
- Link_Space#5 represents the Link Space object which manages all of the Traders' links. It maintains a record for each Trader in the Trading Environment. These records indicate which other traders may be used when interworking. As with the Offer_Space object, its functionality is available to all Traders in the system.
- Importer#6 represents the Importing object. This page models an importing object which can submit multiple Queries to Traders in the model.
- Exporter#18 represents the Exporting object. This object is capable of exporting Offers to the Trader and also contains methods which provide the services that it will export to the Trader.
- OGA_Page#16 contains the occurrence graph functions used with the OEOS analysis of the model. It also contains ML code that is used to configured parameters associated with generating the OG.
- Draw_OGA#17 is used as a temporary page for drawing occurrence graphs as required. This is useful when debugging a model since it can be used to backtrack errors to their source.

The Offer_Space#4 and Link_Space#5 pages represent objects which have not been specified in the ODP Trader Standard [16]. Thus, in this model, these objects have been modelled using a

“common sense” approach. That is, they are modelled to provide Traders with the minimum functionality required to perform the Trading Function. This allows testing of the Trading Environment model through simulation runs and OG analysis using Design/CPN.

6.8.1 Global Declaration Node

As with the CPN model of the Trader presented in Chapter 5, the GDN (GDN#1) is used to declare variables and define colour sets and functions. The GDN of the model presented in this chapter is very large (14 pages) and thus, cannot be described in minute detail. However, it is included in Appendix A and has been extensively commented to improve readability and comprehension.

The GDN is structured into the following sections:

- colour set definitions which define the data structures used throughout the model,
- functions used throughout the model which are used to perform complex processing of tokens on inscriptions throughout the model, and
- variables used in inscriptions throughout the model.

6.8.2 Colour Sets

In this chapter’s description of the model, only the colour set definitions which are essential to the understanding of the model will be covered. The reader is invited to reference Appendix A for more detailed information on the model as necessary.

Rather than describing all colour set declarations in one place, colour sets will be described as they are used throughout the model. This allows the reader to associate colour sets with certain functions of the model, thereby improving comprehension.

6.8.3 Functions

As with the colour sets in the model, functions will be discussed as they are encountered within the model, except for general purpose projection functions which are described in the next section.

6.8.3.1 Projection Functions

In order to access information which is *hidden* inside tokens, some *Projection Functions* have been written. They allow the model to move data around as high order colours, whilst still allowing access to data within the token. Some basic functions are used to access the most commonly used data in a `Message` token, all of which follow a basic pattern. The input parameter to the functions is a `Message` token.

```
(** Projection functions that extract elements from Message tokens **)  
fun S(((s,r),dat:Data):Message):Id = s;  
fun R(((s,r),dat:Data):Message):Id = r;
```

```

fun A((addr:Addresses,dat:Data):Message):Addresses = addr;
fun D((addr:Addresses,dat:Data):Message):Data = dat;

fun Op((addr:Addresses,(opn,(op_e,t_id))):Message):Operation = opn;
fun Op_E((addr:Addresses,(opn,(op_e,t_id))):Message):Op_Elem = op_e;
fun Op_D((addr:Addresses,(opn,(op_e,t_id))):Message):Op_Data = (op_e,t_id);
fun T_Id((addr:Addresses,(opn,(op_e,t_id))):Message):Trans_Id = t_id;

```

Using the same approach, the following projection functions operate on instances of other tokens.

```

(** Projection function to extract the Request ID from an Import Request **)
fun Req_Id((serv:Serv_Type,imp_pol:Imp_Policy):Imp_Req):Request_Id =
#request_id(imp_pol);

(** Projection fn. to extract the Import Policy from an Import Request **)
fun I_Pol((serv:Serv_Type,imp_pol:Imp_Policy):Imp_Req):Imp_Policy = imp_pol;

(** Projection functions to extract the Service Interface Identifier and
    Service Type from a matched Offer. Used by the Importer object **)
fun SIID((int_t:Int_Type,prop_l:Prop_List):Serv_Type,s_int_id:Id):Serv_Off)
:Id = s_int_id;

fun SType(((int_t:Int_Type,prop_l:Prop_List):Serv_Type,s_int_id:Serv_Int_Id)
:Serv_Off):Serv_Type = (int_t,prop_l);

```

6.8.4 Variables

A number of variables are declared at the end of the GDN. They have been re-used on multiple pages where possible. Each variable is assigned a colour set and can only be bound to a value within that colour set.

Important variables throughout the model include:

```

(** Used all over the model when processing messages **)
var mess : Message;

(** Sender and Receiver object identifiers **)
var s: Id; (Sender Id)
var r: Id; (Receiver Id)

var op_d : Op_Data;
var op_e : Op_Elem;

(** Transaction identifier **)
var t_id : Trans_Id;

```

Other variables are used throughout the model which are not explicitly explained in the text. The reader is directed to refer to Appendix A, which contains a complete listing of the GDN.

6.9 Trading Environment

The Trader Environment page (Trad_Env#2) is the *highest* page in the model. It is used to model the object instances in the system. There are six objects in figure 6.9, each of which is represented by a substitution transition. A substitution transition is indicated by the presence of a HS within a box located in each transition on the page. The sub-page associated with each substitution transition is shown in a nearby box. When multiple instances of objects are in the

system, each instance refers to an instance of the same sub-page, which has its own marking and hence, its own state (as discussed in section 6.2).

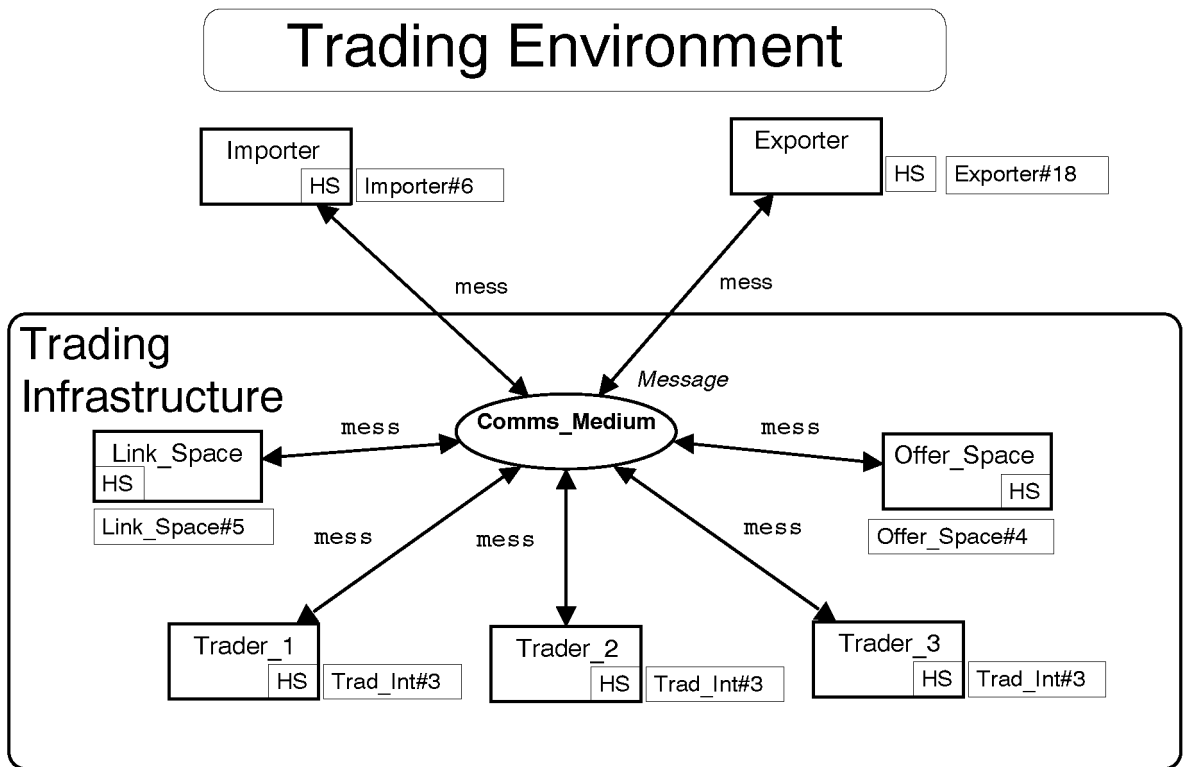


Fig. 6.9: Trading Environment (Trad_Env#2)

Each object in the system has a unique object identifier (required for message addressing) which must be a member of the *Id* colour set defined below:

```
(** Identifiers of all the Objects in the System **)
color Id = with
    (** Trading Objects **)
    importer |
    exporter |
    search   |
    link_space |
    offer_space |
    (** Generic Interface Identifiers **)
    OSI      |
    LSI      |
    (** Lookup Interfaces **)
    lui_1    |
    lui_2    |
    lui_3    |
    lui_4    |
    (** Register Interfaces **)
    regi_1   |
    regi_2   |
    regi_3   |
    regi_4   |
    (** Offer Space Interfaces **)
    t1_osi   |
    t2_osi   |
```

```

        t3_osi      |
        t4_osi      |
(** Link Space Interfaces **)
        t1_lsi      |
        t2_lsi      |
        t3_lsi      |
        t4_lsi      |
(** Service Providers **)
(**- Pizza providers -**)
        pizza_haven |
        pedro_pizza |
        pizza_chef  |
(**- Software providers -**)
        software_supermarket |
        advantek     |
        virgin       |
(**- Used Car providers -**)
        bob_moran    |
        aust_motors  |
        champion     |
        NO_ID;

```

The above definition for *Id* includes identifiers for three Traders. As more Traders are added to the model, they require additional unique identifiers to be added to the *Id* colour set definition.

The Trader has been modelled as a distributed message passing system. The main Colour set at this level of abstraction is *Message*, which defines the structure of tokens passed between objects in the system. It is defined as a tuple of two colours which represent *Addresses* and *Data* fields (as previously discussed in section 6.4.2). The colour sets required for the definition of the *Message* colour set are outlined below:

```

(** Addressing mechanism that requires a sender and receiver object
    identifier **)
color Addresses = product Id*Id;

(** Small Integer ranges from -1 to 4 **)
color Small_Int = int with ~1..4;

(** Unique Transaction identifiers ranging from -1 to 4 **)
color Trans_Id = Small_Int;

(** Operation Data contains the operation element and a transaction
    identifier **)
color Op_Data = product Op_Elem*Trans_Id;

(** Data payload for messages contains the message operation and the
    Operation Data **)
color Data = product Operation*Op_Data;

(** Message is defined as a product of addressing information and
    the data contained within the message **)
color Message = product Addresses*Data;

```

Addresses is a pair of sender and receiver identifiers for the message, and *Data* is a pair containing an *Operation* to be performed and Operation Data (*Op_Data*), which are parameters for the operation. *Op_Data* is also defined as the product of two elements: an Operation Element (*Op_Elem*) and a Transaction identifier (*Trans_Id*).

The *Operation* colour set defines the names of all valid operations (commands) which objects may perform, as shown below:

```
(** Identifiers for all of the possible Operations in the model **)
color Operation = with add_offer      | (** Offer Space Operations **)
                   get_offers       |
                   import            | (** Importer operations **)
                   submit_order      |
                   export            | (** Exporter Operation **)
                   get_links         | (** LinkSpace Operation **)
                   error             | (** General Purpose Opns **)
                   respond           |
                   acknowledge;
```

All Operations may have an associated *Op_Elem* which contains Operation parameters. *Op_Elem* is defined as a Union of different colours which represent parameters required for different Operations.

```
(** An operation element is selected from the union of these elements,
    depending upon the object which the operation element is intended for **)
color Op_Elem = union err:Error+
               imp:Imp_Req+
               exp_return:Small_Int+
               ack:Ack+
               none:Null+
               off:Serv_Off+
               link_l:Link_List+
               serv_t:Serv_Type+
               off_l:Offer_List;
```

Thus, any CPN place typed by the colour set *Op_Elem* may store a token which is a member of the union. There is a need to specify the union identifier of the colour set (such as *link_l* for data of colour set *Link_List*), when tokens are taken from a place. This indicates the colour set of the data and is used extensively throughout the model. In most cases, *Op_Elem* is coupled with a transaction identifier as an *Op_Data* token. This ensures that the model can differentiate between tokens that are used by concurrent threads and is used extensively throughout the model.

Objects communicate using input and output ports which are mapped from the high-level Trader Environment page to the corresponding sub-page which represents the object. Thus, the place *Comms_Medium* has a place in each substitution transition sub-page, where each of these places has the same marking. This is evident in the Importer, Exporter, LinkSpace, OfferSpace and Trader objects.

One way to ensure that threads are consistent throughout the model is to use a guard such as `[T_Id(mess)=t_id]` on transitions. This ensures that the transaction identifier of a message on a transition's input arc is the same as the transaction identifier (*t_id*) associated with other tokens on input arcs to the transition. An alternative is to explicitly use the *t_id* variable in all inscriptions, thereby ensuring that only tokens with the same *t_id* will be consumed when a transition occurs.

6.10 Auxiliary objects

There are a number of objects in the Trading Environment which are required as part of the Trading environment but are not Traders themselves. These objects are considered an auxiliary group of objects and their CPN models are described in this section.

6.10.1 OfferSpace

An OfferSpace object that is capable of servicing multiple requests at a time is modelled on Offer_Space#4, shown in figure 6.10. It provides a `get_offers()` method (represented by the **Apply_Matching** transition) that can be used by Traders to retrieve offers that they have previously stored in the *Offer_Storage* database. The OfferSpace object in the model does not provide a method for adding offers to the offer space since this may be simulated by modifying the initial marking of *Offer_Storage*.

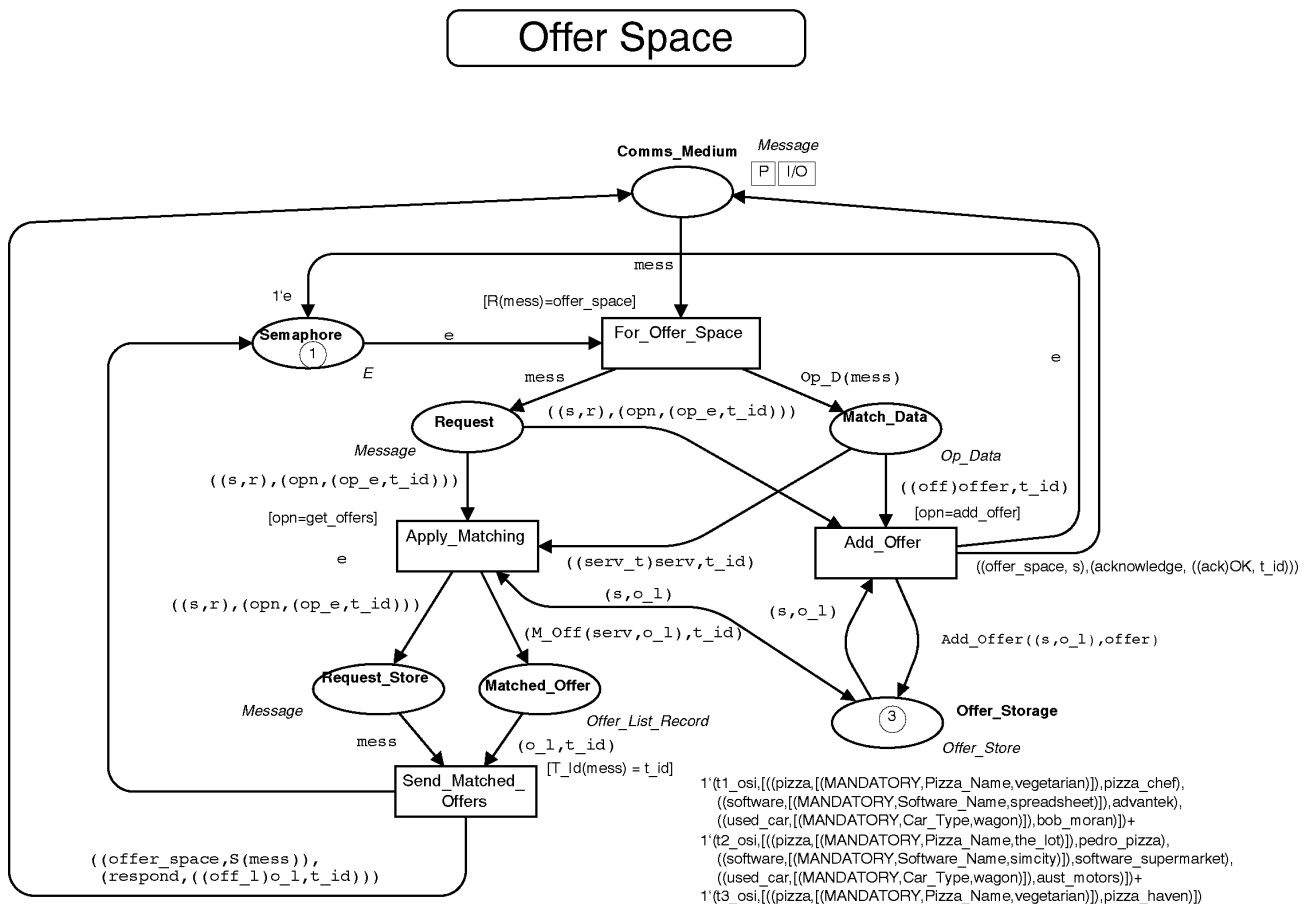


Fig. 6.10: A CPN of the OfferSpace object (Offer_Space#4)

The OfferSpace object stores service offers for later retrieval. A service offer contains a service type (as discussed in section 3.4.2) and a Service Interface Identifier at which the service is available, as further defined below:

```
(** Set of service interface identifiers. Same as the
    set of object identifiers **)
color Serv_Int_Id = Id;
```

A Service Interface Identifier is the same as an object identifier.

```

(** Set of service types **)
color Int_Type = with pizza |
                 software |
                 used_car;

(** Set of possible service property modes **)
color PropertyMode = with NORMAL |
                     READONLY |
                     MANDATORY |
                     MANDATORY_READONLY;

(** Set of service property names **)
color PropName = with Pizza_Name |
                  Software_Name |
                  Engine_Size |
                  Car_Type;

(** Set of service property values **)
color PropValue = with the_lot |
                  vegetarian |
                  simcity |
                  spreadsheet |
                  six_cylinder |
                  three_litre |
                  sports |
                  wagon;

(** Definition of a service property. It is a triple comprising
    the Property Mode, Property name and the property value **)
color Properties = product PropertyMode*PropName*PropValue;

(** List of service properties **)
color Prop_List = list Properties;

(** A list of Offers used by the OfferSpace Object **)
color Offer_List = list Serv_Off;

(** A Service Type contains the Interface Type and a list of
    properties associated with the service available at the interface **)
color Serv_Type = product Int_Type*Prop_List;

(** Service Offer contains the Service Type and a Service Interface Identifier
    to uniquely identify the service and its provider **)
color Serv_Off = product Serv_Type*Serv_Int_Id;

(** Colour defining the System's set of OfferSpace Interfaces **)
color OSInt = subset Id with [t1_osi,t2_osi,t3_osi,t4_osi] declare in;

(** Used by the OfferSpace object to store offers for multiple Traders.
    Used on page Offer_Space#4 **)
color Offer_Store = product OSInt*Offer_List;

```

An initial marking of the OfferSpace object's database (maintained in *Offer_Storage*) is given below:

```

1^(t1_osi,[(pizza,[(MANDATORY,Pizza_Name,vegetarian)]),pizza_chef),
          ((software,[(MANDATORY,Software_Name,spreadsheet)]),advantek),
          ((used_car,[(MANDATORY,Car_Type,wagon)]),bob_moran)]+
1^(t2_osi,[(pizza,[(MANDATORY,Pizza_Name,the_lot)]),pedro_pizza),
          ((software,[(MANDATORY,Software_Name,spreadsheet)]),

```

```

software_supermarket),
((used_car, [(MANDATORY, Car_Type, wagon)], aust_motors))] +
1'(t3_osi, [(pizza, [(MANDATORY, Pizza_Name, vegetarian)], pizza_haven)])

```

The initial marking contains three Offer_Space tokens, each of which is associated with a different OSI instance (t1_osi, t2_osi or t3_osi). Each of the tokens contains a list of Offers, including pizza, used_car and software services. Offers are explained in sections 3.4.3 and 6.10.3)

Messages arrive at the OfferSpace object via *Comms_Medium* when the **For_Offer_Space** transition occurs. This transition may only become enabled if there is a message token addressed to offer_space in *Comms_Medium*. When **Accept_Message** occurs, the Operation Data of mess is extracted into *Match_Data*, and the incoming message is stored in *Request*.

The following tuple contains a number of variables which have corresponding elements within a Message token.

```
((s,r),(opn,(op_e,t_id)))
```

Each variable can be bound to a value from its designated colour set using CPN/ML's ability to perform pattern matching:

s: colour set Id, (sender)

r: colour set Id, (receiver)

opn: colour set Operation, (operation command)

op_e: colour set Op_Elem, (operation element)

t_id: colour set Trans_Id. (transaction identifier)

6.10.1.1 get_offers() Method

Apply_Matching may only become enabled if the incoming message had an Operation field equal to get_offers. When **Apply_Matching** occurs, the binding for s (which is the Id of the Trader that sent the message) is used to select the offers associated with s from *Offer_Storage*. These offers are stored in a list which is bound to o_l. M_Off() is a function used to select offers that match serv from the list of offers in o_l and are put into *Matched_Offers*. **Send_Matched_Offers** is used to send a message back to the Trader that initiated the request, containing the matched offers.

6.10.1.2 add_offer() Method

Add_Offer may only become enabled if the incoming message had an Operation field equal to add_offer. As with the get_offers() method, when **Add_Offer** occurs, the binding for s (which is the Id of the Trader that sent the message) is used to select the offers associated with s from *Offer_Storage*, which are stored in a list that is bound to o_l. Add_Offer() is a function used to add a new Offer to the list of Offers in o_l which is then replaced in *Offer_Storage*. Finally, when **Add_Offer** occurs, an acknowledgment message is sent back to the Trader that initiated the request.

6.10.1.3 Summary

The functionality of this object is very similar to that of the LinkSpace object since both are used to maintain data for a group of Traders. The main difference in structure between the CPNs is due to the syntax demands of CPN/ML which requires that the Operation Data of the message is explicitly extracted into a place that indicates its colour set. This data may then be used by `M_Off()` when applying matching to the list of offered services. It is not possible to perform pattern matching on the data within the message and use it in the `M_Off()` function call because operation data is a union and thus, can be of many possible colour sets. An explicit cast into the expected colour set is not possible. This is a limitation of CPN/ML which has resulted in much more complicated CPNs than desired.

6.10.2 LinkSpace

The Link Space object is modelled on `Link_Space#5`, as shown in figure 6.11. Its function is to maintain a database of links for multiple clients (Traders). It provides clients with a `get_links()` method which allows them to retrieve their own list of links. However, it does not provide methods which allow clients to add or modify links. This means that its state (*Link_Storage*) must be configured when the CPN marking is initialised. If a Trader is to be linked to a new Trader, then the initial marking of *Link_Storage* must be altered accordingly.

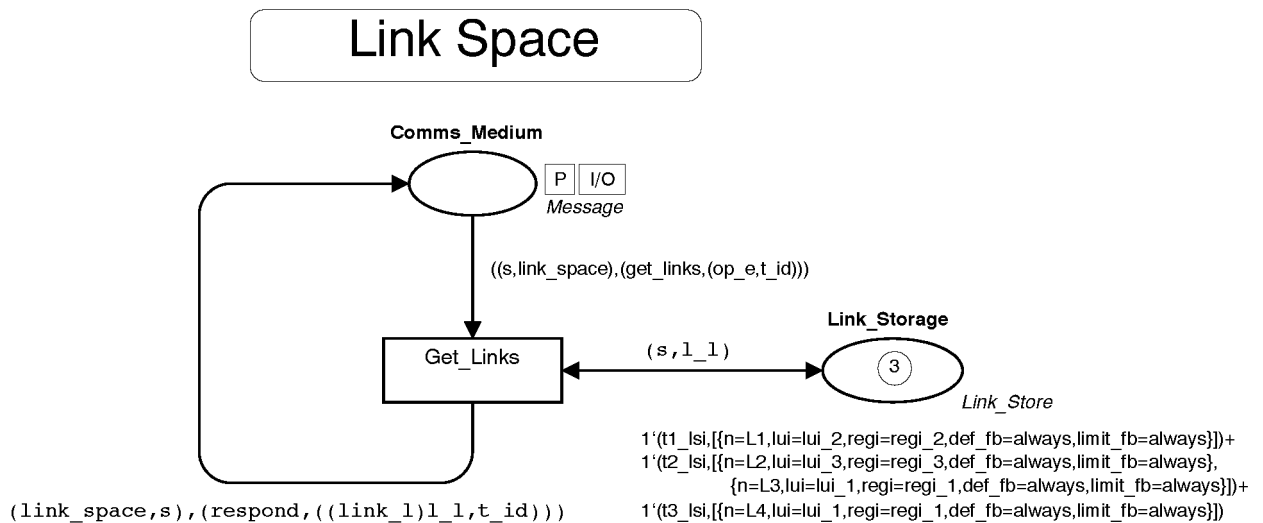


Fig. 6.11: A CPN of the LinkSpace object (`Link_Space#5`)

In the scenario presented in figure 6.11, the initial marking of *Link_Storage* contains a set of *Link_Store* tokens. This means that for each Trader that uses the LinkSpace object, there is a token in *Link_Storage* maintaining a list of links for that Trader. Colour sets associated with Links are defined in the GDN as:

```
(** Names associated with Links **)
color Link_Name = with L1 | L2 | L3 | L4 | L5 | L6;

(** Definition of Link Follow Options **)
color Follow_Option = with no_option | local_only | if_no_local | always;
```

```

(** Colour defining the System's set of Lookup and Register Interfaces **)
color LUInt = subset Id with [lui_1,lui_2,lui_3,lui_4] declare in;
color RegInt = subset Id with [regi_1,regi_2,regi_3,regi_4] declare in;

(** Link Data Structure including :
    Name, Lookup Interface, Register Interface,
    Default and Limiting Follow Behaviour **)
color Link = record n:Link_Name*
                lui:LUInt*
                regi:RegInt*
                def_fb:Follow_Option*
                limit_fb:Follow_Option;

```

Link was defined using a record since records allow easy extraction of a single element without requiring projection functions to be written. It is also more intuitive to use a record structure since that is precisely what it is intended to represent.

```

(** A list of Links **)
color Link_List = list Link;

(** Colour defining the System's set of LinkSpace Interfaces **)
color LSInt = subset Id with [t1_lsi,t2_lsi,t3_lsi,t4_lsi] declare in;

(** Used by the Link Space object to store links for
    multiple Traders. Used on page Link_Space#5 **)
color Link_Store = product LSInt*Link_List;

```

and where *Link_Storage* has the following initial marking:

```

1'(t1_lsi, [{n=L1, lui=lui_2, regi=regi_2, def_fb=always, limit_fb=always}]) +
1'(t2_lsi, [{n=L2, lui=lui_3, regi=regi_3, def_fb=always, limit_fb=always},
            {n=L3, lui=lui_1, regi=regi_1, def_fb=always, limit_fb=always}]) +
1'(t3_lsi, [{n=L4, lui=lui_1, regi=regi_1, def_fb=always, limit_fb=always}])

```

The initial marking contains a token for each of the LSIs (*t1_lsi*, *t2_lsi* and *t3_lsi*), which correspond to Trader 1, 2 and 3 LinkSpace Interfaces respectively. Trader 1 (T1) is linked to Trader 2 (T2) via link L1, T2 is linked to T3 and T1 via links L2 and L3 respectively, and T3 is linked to T1 via link L4. All of these links have a default follow behaviour of *always*, and a limiting follow behaviour of *always*. This trading topology is illustrated in figure 6.12, where an importer (I) uses T1.

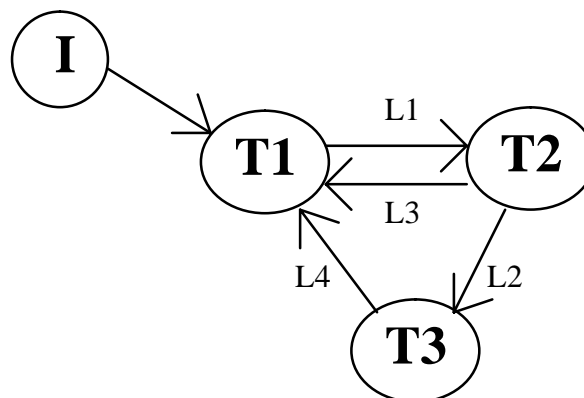


Fig. 6.12: Three linked Traders

As with all other objects in the model, the LinkSpace object only accepts messages which are addressed to it. This is ensured by the receiver field of the inscription associated with the input arc to **Get_Links**, which mandates that the receiver field of a Message token in *Comms_Medium* must have a value of `link_space` before the transition becomes enabled.

Messages can be broken down into their smaller components using the pattern matching capability of CPN/ML, as discussed in section 6.10.1, where

```
((s,r),(opn,(op_e,t_id)))
```

allows components of a Message token to be assigned to different variables.

When **Get_Links** occurs, a message token is taken from the *Comms_Medium* and a list of links (`l_l`) is selected from the link database in *Link_Storage*, using the binding for `s` to select the token associated with `s` from *Link_Storage*. This list of links is returned to the Trader which initiated the request as the payload of a response message which is put into *Comms_Medium*. The response message is addressed to `s`, the sender of the initial message and is sent by `link_space`. Its operation is `respond`, its operation data is the matching list of links (`l_l`) and the original transaction identifier (`t_id`) of the request.

6.10.3 Importer

The Importer object with two Queries is modelled by Importer#6, as shown in figure 6.13. This object's function is to send Queries to Traders and accept the resulting list of matching Offers. A discussion of the colour sets required by this page follows.

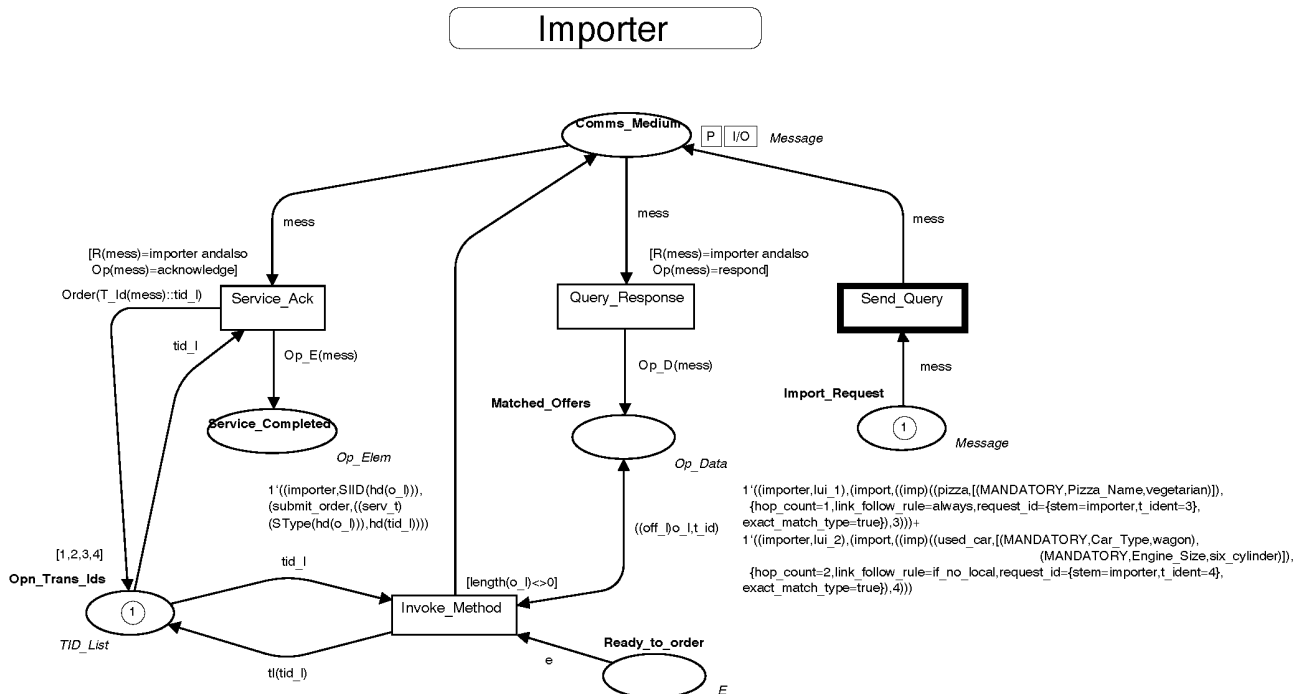


Fig. 6.13: Importer with two Queries

```
(** Stem used in identifying the originator of a Request **)  
color Request_Stem = Id;
```

```

(** Request_Id contains the request stem and also a transaction
    identifier. These elements uniquely identify a request from a
    specific object **)
color Request_Id = record stem:Request_Stem*
                      t_ident:Trans_Id;

```

As discussed in section 3.4.8.2, the Importer should provide a Request_Id parameter when submitting a Query to a Trader to allow detection of duplicate Queries.

```

(** Import Policy specified by the Importing Object **)
color Imp_Policy = record hop_count:Small_Int*
                      link_follow_rule:Follow_Option*
                      request_id:Request_Id*
                      exact_match_type:boolean;

```

Section 3.4.8.2 provides a discussion of the fields which are specified to be part of the Import Policy. However, only four of the fields have been included in the CPN model since the other fields are relevant to behaviour which is not supported by the Trader model (i.e. search, match and return cardinality and the starting Trader name for a Query).

```

(** Import Request contains the Service Type required and
    an associated Import Policy **)
color Imp_Req = product Serv_Type*Imp_Policy;

```

The initial marking of *Import_Request* contains two Query messages which are addressed to different Traders and request different services. A Query message has an Op_Elem of type Imp_Req, which is a tuple containing a Service Type and an Import Policy, as defined above. The Imp_Policy record is based upon the Import Policy requirements identified in section 3.4.8.2.

```

1'((importer,lui_1),(import,((imp)((pizza,[(MANDATORY,Pizza_Name,the_lot)]),
  {hop_count=2,link_follow_rule=always,request_id={stem=importer,t_ident=3},
  exact_match_type=true})),3)))
1'((importer,lui_2),(import,((imp)((used_car,
  [(MANDATORY,Car_Type,wagon),(MANDATORY,Engine_Size,6)]),
  {hop_count=1,link_follow_rule=always,request_id={stem=importer,t_ident=4},
  exact_match_type=true})),4)))

```

The Query with receiver field set to lui_2 is requesting a used_car service. More specifically, the parameters associated with the Query indicate that the Car_Type desired is wagon, and the number of cylinders is 6. Each parameter is MANDATORY, which means that matched offers must provide a value for these fields. Attributes may also be READ_ONLY which means that they cannot be altered by the exporter at a later date. The Query has a hop_count of 1, and Request_Id identifying the Query as transaction 4, originating from the Importer object.

Queries are created in *Import_Request* when the model is initialised and are moved into *Comms_Medium* when *Send_Query* occurs. When a Trader has processed a Query, its response is returned to the Importer via *Comms_Medium*. Since the receiver field of a response is set to importer, and the Operation of mess is respond, **Query_Response** becomes enabled when *Comms_Medium* contains a response token. When it occurs, the response is accepted by the Importer and the Operation Element (Op_E) of the response is extracted and put into

Matched_Offers. This token is a tuple containing the result of the Query and the transaction identifier of the initial Query. The matching Offers are contained in a list and represent the conclusion of the Query transaction. It is important for the response message to retain the initial Query's transaction identifier so that the Importer can match responses with the Queries it previously initiated.

As with the model presented in section 5.4, having obtained a reference to a service provider, the Importer subsequently interacts with the interface which provides the matched service. This is modelled by the **Invoke_Method** transition which can only become enabled when there is a list of matching offers in **Matched_Offers**, and an `e` token in **Ready_to_order**. When **Invoke_Method** occurs, the Importer is no longer **Ready_to_order** and thus, the token from that place is consumed, whilst the token containing a list of matched offers is returned to **Matched_Offers**. In addition, a token with Operation field set to `send_order` is sent to the interface which provides the service, using the head `t_id` from the list of T_Ids contained in **Trans_Ids** to identify the new transaction.

When the service provider (probably the Exporter of the service) sends a response to the service request with `Op(mess)=acknowledge`, **Service_Ack** becomes enabled. When it occurs, the response message is consumed, the message's `t_id` is replaced into the list of available `t_ids` in **Trans_Ids** and the service provider's response is put into **Service_Completed**. This represents the final step in the sequence of interactions between the Exporter, Trader and Importer objects, as modelled in section 5.4.

6.10.4 Exporter

The Exporter object is modelled on Exporter#18 page, as shown in figure 6.14. The initial marking of **Offers_to_Export** contains an Offer which is to be exported by the Exporter to the Trader when **Export_Offer** occurs. The initial marking is as follows:

```
1``((exporter,regi_1),
    (export,((off)((pizza,[MANDATORY,Pizza_Name,the_lot]),exporter),3)))
```

This represents a message from the `exporter` object to `regi_1` exporting a `pizza` service with `Pizza_Name` service attribute set to `the_lot`, service location equal to `exporter` and the export operation's transaction identifier equal to 3.

When the Trader has completed the export operation, it returns a `respond` message to the Exporter which is accepted by the Exporter when **Export_Reply** occurs. This response contains a tuple that associates an offer identifier (of colour `Small_Int` and union identifier `exp_return`) with the transaction identifier (`t_id`) used in the initial Export. This allows the Exporter to maintain a reference to the exported Offer (in **Offer_Identifiers**) so that it may modify or withdraw the offer at a later date if necessary.

The Exporter also provides a transition which occurs when a message containing a `submit_offer` Operation is sent to the Exporter. When **Accept_Order** occurs, the Exporter takes the order it was sent and stores it in **Pending_Order**, which represents an on-line ordering system.

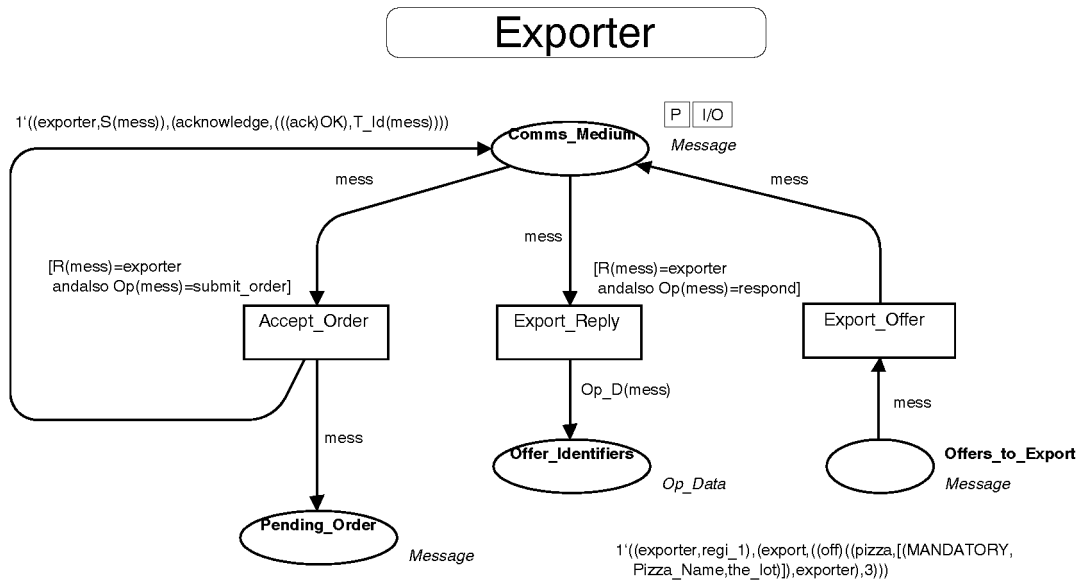


Fig. 6.14: Exporter object with one Offer

6.11 Trader

This section contains a description of the pages which model the Trader in the Trading System. It shows the relationship between the pages and explains how multiple instances and multi-threading are included in the model. It also shows how the Trader model has been designed to allow for further refinement and expansion as necessary.

6.11.1 Trading Interfaces

The Trader is modelled on Trad_Int#3 in figure 6.15 as a group of interfaces which are connected to the *Comms_Medium* and also to functional operations (methods) as shown in figure 6.15. The Trader utilises a number of Interfaces which are grouped into three major categories: Functional, Inter-Object and Abstract. The functional interfaces are provided by the Trader to its clients and provide an interface to the Trading function's services. Inter-object interfaces are used by the Trader when communicating with the link and offer space objects. The Abstract interface provides clients with a means for inspecting Trader attributes.

Trad_Int#3 provides a view of the Trader that illustrates the relationship between the *Comms_Medium*, the Trader's interfaces and methods (or operations). All messages are passed through the *Comms_Medium* and may be accepted by any of the Functional, Inter-object or Abstract interfaces. They include:

Functional Interfaces: Lookup, Register, Proxy, Link and Admin
 Inter-object: Offer_Space_Interface, Link_Space_Interface
 Abstract: Trader Components, Support Attributes, Import Attributes.

The Functional and Inter-object interface groups are represented by a substitution transition which has a corresponding sub-page, providing a more detailed description of the interfaces contained within. The Abstract interface is represented by a transition Abstract which has a guard

which always evaluates to *false*. This is used to provide a “placeholder” transition that can be converted into a substitution transition at a later date if greater modelling detail of the Abstract interface is required.

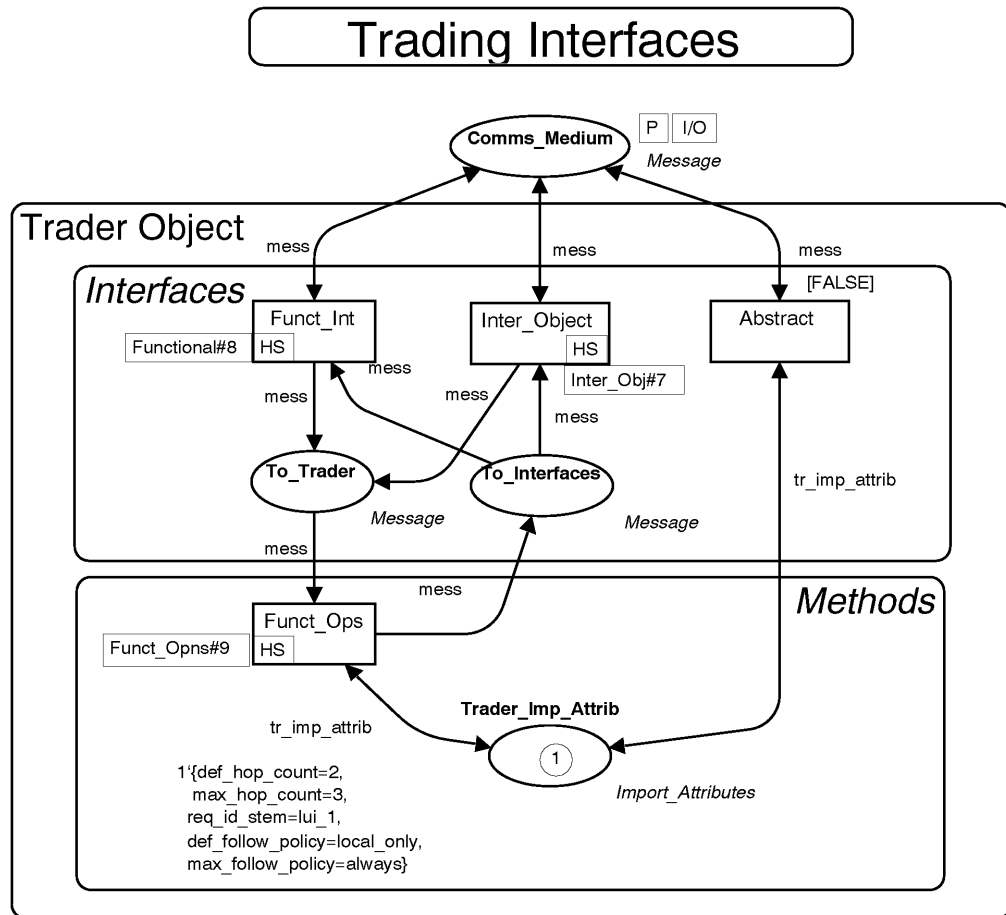


Fig. 6.15: Trading Interfaces (Trad_Int#3)

The Functional and Inter-object interfaces are connected to **Funct_Ops** by two places labelled **To_Trader** and **To_Interfaces**. These places are used to forward message tokens from the interfaces to the Trader’s functional operations which are further modelled in the **Funct_Ops** substitution transition. Thus, both of these interfaces are able to pass messages to the functional operations within the Trader. Additionally, they are able to accept tokens which the Trader places in **To_Interfaces**. This topology makes it simple to add new functions to the Trader, as discussed in section 6.11.5.

The Abstract interface allows clients to read the values of the Trader’s internal policies which are located in **Trader_Imp_Attrib**. As can be seen in figure 6.15, the Functional Operations (contained within the **Funct_Ops** substitution transition) also require access to the Trader’s internal policy values. Thus, **Trader_Imp_Attrib** is connected by a bi-directional arc to **Funct_Ops** so it can use and replace the Trader Policy information.

The **Trader_Imp_Attrib** place is initialised with a token that has the following value:

```
1`{def_hop_count=2,
    max_hop_count=3,
    req_id_stem=lui_1}
```

```
def_follow_policy=local_only,
max_follow_policy=if_no_local}
```

- `def_hop_count` is used when a Query does not specify a `hop_count`.
i.e. it is the default `hop_count` value which in this case, is set to a value of 2.
- `max_hop_count` is used as a limiting `hop_count` for all requests that the Trader services,
- `req_id_stem` is the identifier of the lookup interface which is used as a stem for the request id (see section 6.11.6.1).
- `def_follow_policy` is used when a Query does not specify a `follow_policy`.
i.e. it is the default `follow_policy` which in this case is set to `local_only`.
- `max_follow_policy` is a limiting value for all Queries processed by the Trader.
i.e. when policies are unified, the result can never be greater than the Trader's limiting `follow_policy`.

According to the Trading Standard [16], the Abstract Interface is used to modify the Trader's import attributes. However, in the CPN model, if the Trader's attributes need to be changed (to create a new scenario), then the marking of *Trader_Imp_Attrib* can be altered before the commencement of simulation or OG analysis using Design/CPN's "Change Marking" command. Thus, there is no need to include a detailed representation of the Abstract interface in the CPN model since the Trader's policy values can be changed statically before the simulation commences.

6.11.2 Inter-Object Interfaces

Inter-Object interfaces are used by the Trader for communicating with other objects in the Trading Environment, such as for example, the LinkSpace and OfferSpace objects. In the RM-ODP Object model, objects may only interact after establishing a binding between their respective interfaces as described in section 3.2.2. This is modelled by providing the Trader with an instance of the OSI (OfferSpace Interface) and LSI (LinkSpace Interface) which represent such bindings.

From a CPN modelling perspective, the OSI and LSI Interfaces provide a conduit for messages that are bound for external objects which must be sent via the *Comms_Medium*. Without such interfaces, there would be no way for the Trader to send messages to other objects, except for re-using the Functional Interfaces which it presents to external objects. This is undesirable since the Trader would then be using the standardised interfaces for non-standardised purposes (such as communicating with the OfferSpace and LinkSpace objects).

The substitution transition **Inter-Object** in figure 6.15 is assigned to `Inter_Obj#7` which itself contains two substitution transitions. **OSI** is assigned to `OSI#10` and **LSI** is assigned to `LSI#11`.

Inter-Object Interfaces

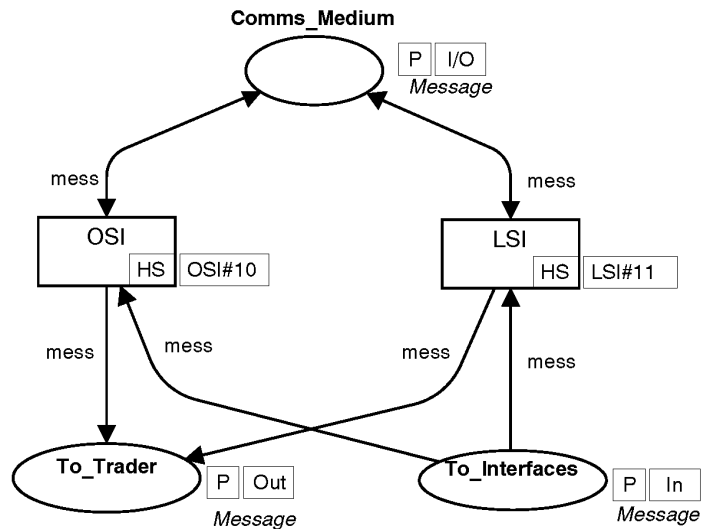


Fig. 6.16: Inter-Object Interfaces (Inter_Obj#7)

Messages from the OfferSpace and LinkSpace objects destined for the Trader pass through either of the interfaces and are put into **To_Trader**. Similarly, messages from the Trader which are destined for the OfferSpace or LinkSpace objects are read from **To_Interfaces** by the OSI or LSI.

By creating an Inter-Object Interfaces page, it becomes simple to add more inter-object bindings to the model since it only requires creating a new substitution transition which is connected via arcs to the **To_Trader** and **To_Interfaces** places. For example, if the Trader needed to communicate with an e-mail object, it would do so using an **Email** interface which would be added to the Inter_Obj#7 page .

6.11.2.1 OfferSpace Interface

The OfferSpace Interface (OSI) shown in figure 6.17 is primarily used for routing messages between the Trader and the OfferSpace object. It's main function is to convert messages to/from a generic instance-neutral identifier (such as OSI or LSI) to a message which contains addressing information specific to the OSI instance that is sending/accepting the message. This is important since it allows lower level pages to create messages that are to be sent via interfaces without having to know the interface's unique object identifier, only the interface type's generic identifier.

For example, the Query operation on page#13 must be able to send messages to the OfferSpace object via an OSI instance. Since Query#13 is re-used by multiple instances of Traders, it is not possible to hard-code an OSI object identifier into the Sender field of a **Message** token. Instead, a generic object identifier such as OSI is used so that when the message is sent to the OSI connected to the Trader, its unique object identifier is substituted into the Message in place of the generic identifier.

Offer Space Interface

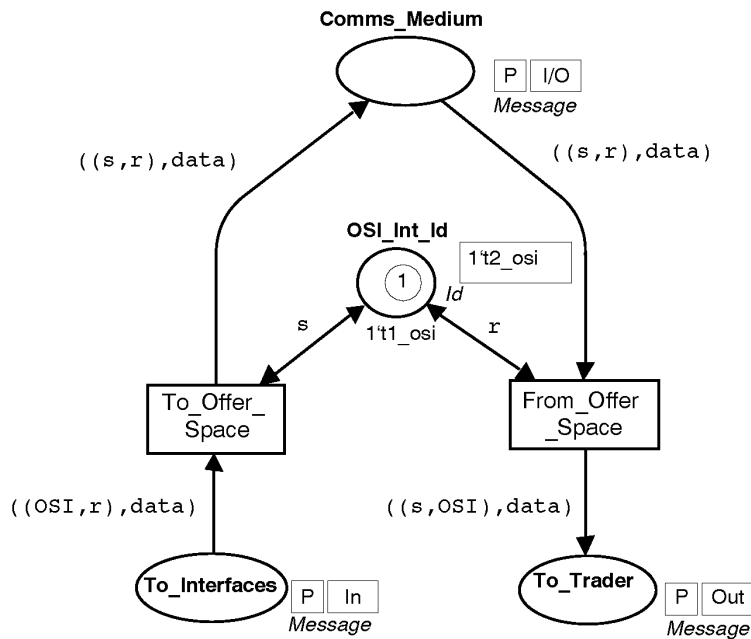


Fig. 6.17: Offer Space Interface (OSI#10)

Thus, the Trader is able to create messages which go through the OSI or LSI without knowing the interface’s unique object identifier. It simply refers to the OSI or LSI generic object identifier, and the interfaces fill in their unique object identifier before sending the message.

The OSI sends messages which it obtains from *To_Interfaces* to the *Comms_Medium* when *To_OfferSpace* occurs. It substitutes its unique object instance identifier (contained within *OSI_Int_Id*) for the generic OSI identifier in the incoming message. Similarly, it accepts messages addressed to its unique object identifier from the *Comms_Medium* when *From_OfferSpace* occurs and substitutes the generic object identifier into the Receiver field of the message when it is forwarded to the Trader in *To_Trader*.

6.11.2.2 LinkSpace Interface

The LinkSpace Interface (LSI) is shown in figure 6.18 and is functionally identical to the OSI, except that it communicates with a LinkSpace object rather than an OfferSpace object. As with the OSI, the LSI allows the Trader to communicate with the LinkSpace object using a dedicated interface (representing an RM–ODP binding).

6.11.3 Functional Interfaces

The Functional Interfaces are used by client objects when they invoke methods on the Trader. They are the interfaces that the Trader presents to the outside world and provide client objects with a well defined interface for accessing Trader functionality.

Figure 6.19 shows the relationship between the *Comms_Medium*, the functional interfaces and the Trader. This topology is similar to that used in the OSI and LSI, connecting the Trader to the

Link Space Interface

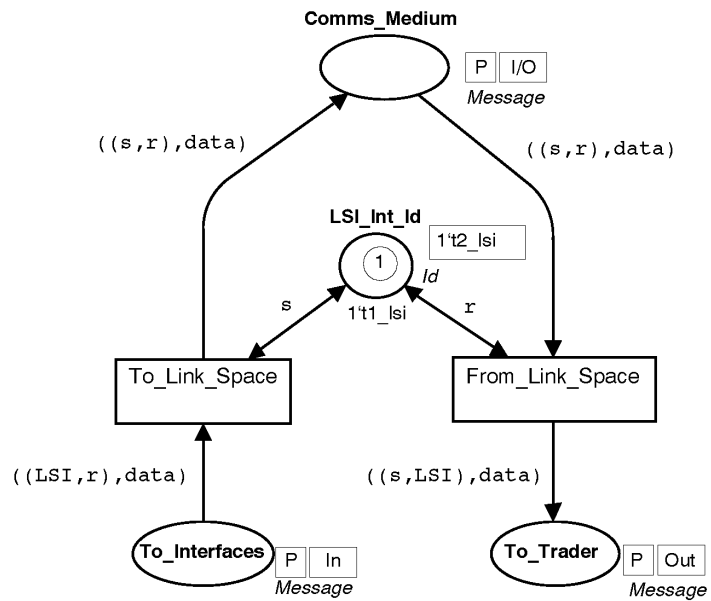


Fig. 6.18: Link Space Interface (LSI#11)

Functional Interfaces

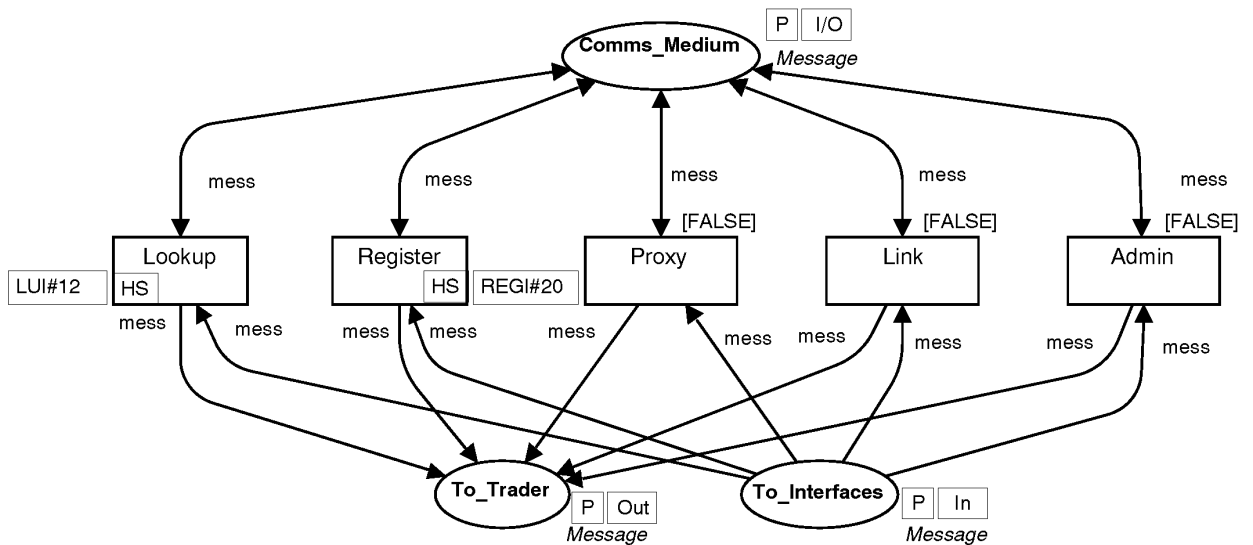


Fig. 6.19: Functional Interfaces (Functional#8)

Comms_Medium. The interfaces are represented by transitions, where the Lookup and Register Interfaces are a substitution transition (associated with LUI#12 and RegI#20 respectively). The Proxy, Link and Admin interfaces are all inscribed with a guard which always evaluates to FALSE, thereby ensuring that they will never become enabled. At a later date, these transitions can be associated with a substitution transition which adds new functionality to the model. For the purposes of analysis, only the Lookup and Register interfaces were included in the model.

6.11.3.1 Lookup Interface

The Lookup Interface page (LUI#12) is shown in figure 6.20. Messages from other objects that are destined for a specific instance of the Lookup Interface (LUI) are accepted by **From_LUI**. This transition uses the token in *LUI_Id* to identify messages in *Comms_Medium* that are addressed to it. For **From_LUI** to become enabled, the receiver field of the message (*r*) must have the same value as the unique LUI identifier contained in *LUI_Id*.

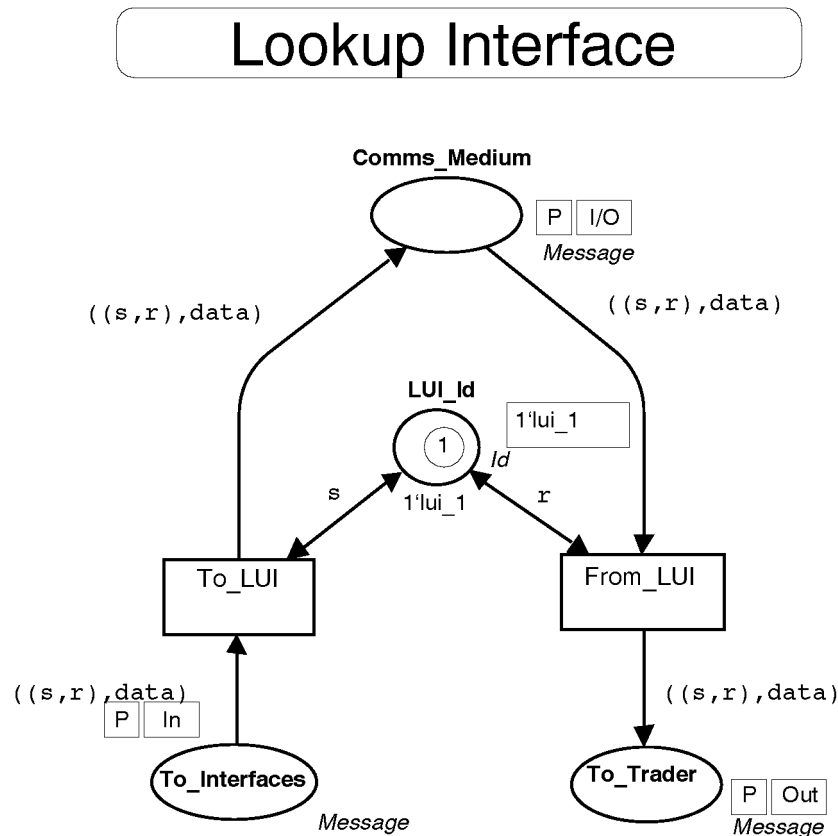


Fig. 6.20: Lookup Interface (LUI#12)

When the model is initialised, the marking of *LUI_Id* must be modified to contain a unique object identifier (i.e. *lui_1*, *lui_2* or *lui_3*). Modifying the marking of this place is essential because otherwise, all instances of the interface have the same object identifier which is therefore, not unique. It is important to ensure that there is only one instance of the interface with a given identifier, otherwise it is impossible to correctly model a message passing system. Unique object identifiers should be in the marking of each instance of *LUI_Id* prior to any simulation or OG analysis. When **From_LUI** occurs, it also forwards *mess* to the Trader for processing in *To Trader*.

Messages from the Trader destined for other objects appear in *To_LUI*. They may be of two types:

- a response to a completed Query operation,
- a forwarded Query destined for a linked Trader.

Thus, the LUI acts as a gateway for messages that are addressed to it, and for messages that the Trader creates when interworking with other Traders. It is one of the published interfaces used by clients in order to utilise the Trader’s services.

6.11.3.2 Register Interface

The CPN model of the Trader includes the Register interface functionality which allows clients to add offers (export services) to the Trader’s offer space. The Register interface is modelled in the same way as the Lookup Interface of LUI#12. When **From_REGI** occurs, the Register Interface accepts messages addressed to the specific Register Interface instance (contained within **REGI_Id**) and forwards them to its associated Trader for processing on the **Funct_Opns#9** page. Similarly, it accepts messages from the Trader which are to be sent to external objects via the Register Interface and puts them into the **Comms_Medium** when **To_REGI** occurs.

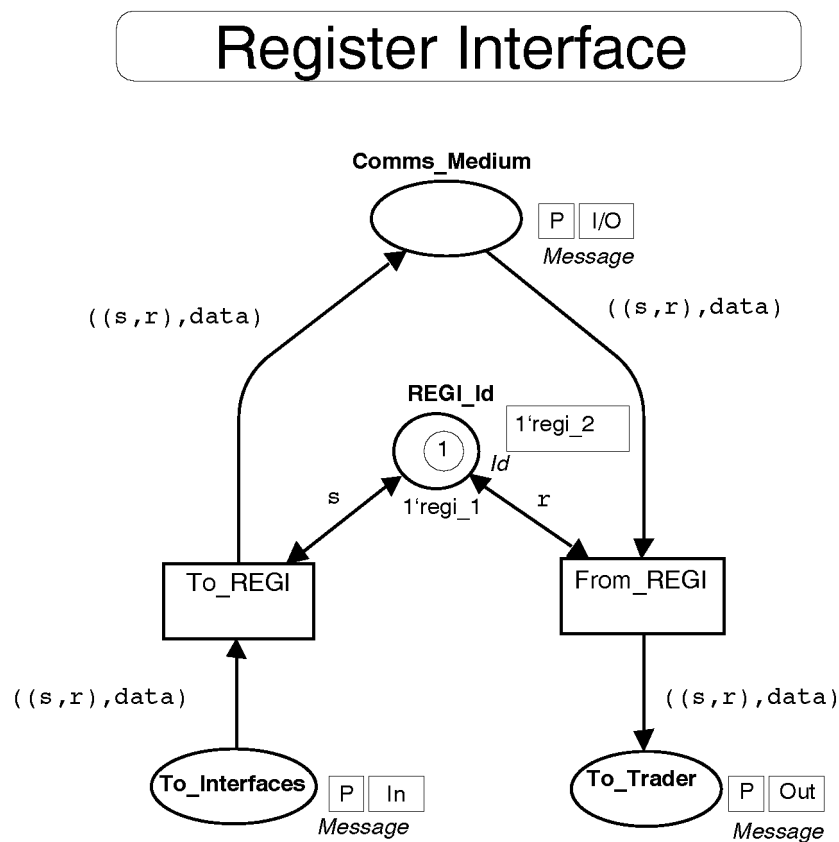


Fig. 6.21: Register Interface (REGI#19)

The Register Interface was included in the CPN model of the Trader because all functionality (methods) provided by the Trader must occur using interfaces. Since each Register interface is a stand-alone object instance, each instance requires a unique object identifier as modelled by **REGI_Id**.

6.11.3.3 Proxy, Link and Admin Interfaces

As described in section 3.4.4.1, the Proxy interface allows exporters to offer services which are provided at dynamically changing interfaces. An exported Proxy offer does not contain a reference to the interface at which the service can be obtained. Instead, it contains a reference to

an interface which the Trader uses for a second Query operation that returns a reference to the interface which provides the service.

This mechanism makes it possible for a service to be offered by an exporter without statically identifying the interface at which the service is provided. Exporters are then able to dynamically alter the interface which provides a service without having to modify services already exported to a Trader.

The Proxy interface was not included in the Trader model since its functionality is not necessary for basic interworking Trader operation. Analysis of this interface is left as an exercise for future work, as stated in section 9.2.

The Link Interface provides the Trader with functionality which can be used by a Trading Administrator to manipulate the Trader's links. Links are managed by the LinkSpace object which can be statically administered using Design/CPN to directly alter its marking. Since one of the modelling assumptions discussed in section 6.3 was the static nature of links, the ability to manipulate Trading topologies dynamically was not required by the model. As with the Proxy interface, investigation of this interface is left as an exercise for future work.

The Admin interface of the Trader allows a Trading Administrator to alter the Trader's internal Import policy values. These values can be read by clients via the Abstract interface, as discussed in section 3.4.4.2. As there is no Trading Administrator object in the Trader model, there is no need to provide an interface for dynamic modification of the Trader's policy information. Thus, the Admin interface was not included in the Trader model since the Trader's policy details were able to be changed statically by altering the CPN marking.

For each of the Proxy, Admin and Link interfaces, a "dummy" transition was included in the Functional Interfaces page (Functional#8) which is used to indicate the interface's location within the model. However, each of the Interfaces has an associated guard expression which always evaluates to FALSE as discussed in section 6.11.3.

6.11.4 Abstract Interface

The Abstract interface provides objects with a facility that allows them to determine the Trader's import policy values which are discussed in section 3.4.4.2. A detailed model of the Abstract interface was not included in the model since its functionality was not required for the Analysis performed in Chapter 7. However, a placeholder transition was included to indicate the Interface's position in the Trader and to make it easy for it to be added to the model at a later date if necessary.

6.11.5 Functional Operations

The Funct_Opns#9 page (shown in figure 6.22) models the Trader's functional operations (methods) which it provides to clients. The page is assigned to the **Funct_Ops** substitution transition located on Trad_Int#3 (discussed in section 6.11.1). The page contains the

Query_Opn and **Export_Opn** substitution transitions which are associated with Query#13 and Export#19 pages respectively.

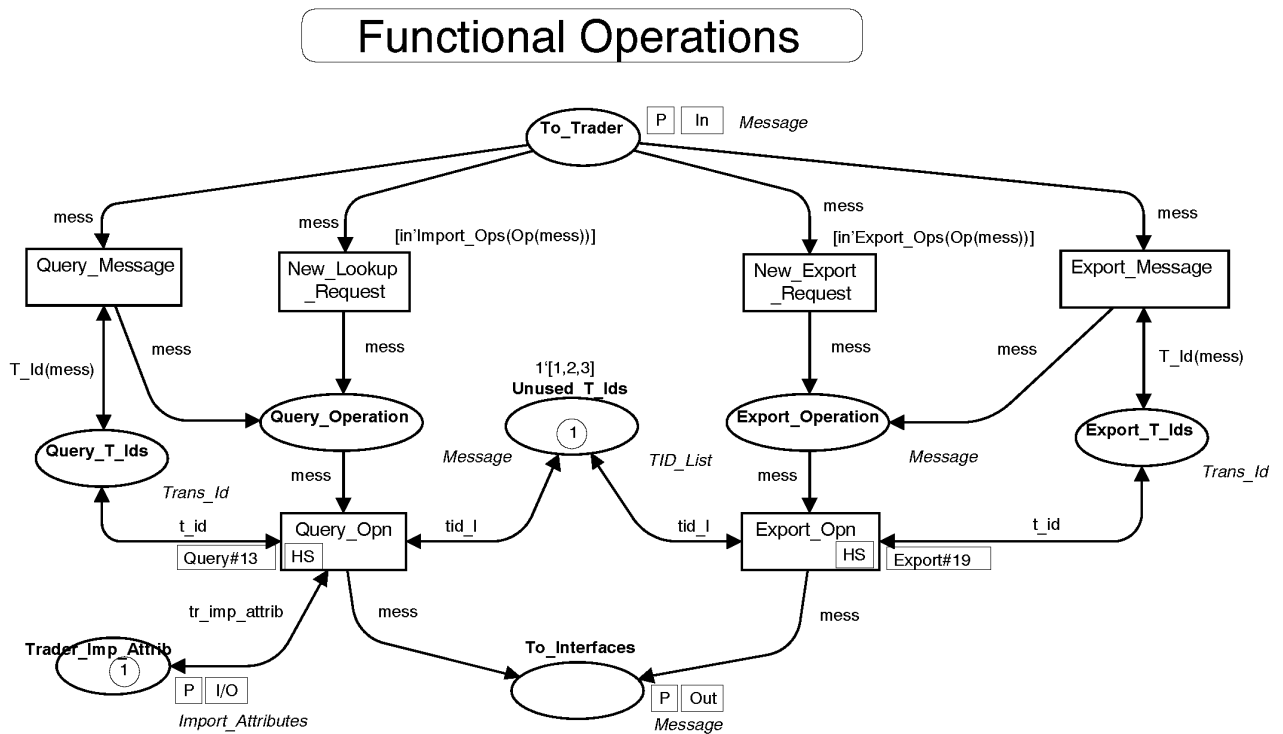


Fig. 6.22: Functional Operations (Funct_Opns#9)

Messages are forwarded to this page via the higher-order Functional Interface page (Functional#8) which contains the Lookup (LUI#12) and Register (RegI#20) Interfaces. Thus, all messages destined for either of the Lookup or Register interfaces are channelled toward the Funct_Opns#9 page to be processed and arrive in *To Trader*.

If a new Query operation is received, **New_Lookup_Request** becomes enabled. **Query_Message** cannot become enabled since initially, there are no tokens in *Query_T_Ids*. When **New_Lookup_Request** occurs, the message containing the new Query operation is put into *Query_Operation*. It is then ready to be processed by the **Query_Opn** sub-page (Query#13) which is described in section 6.11.6. The Query#13 page also uses *Trader_Imp_Attrib* in its processing.

Similarly, if a new Export operation is received from *To Trader*, **New_Export_Request** becomes enabled whilst **Export_Message** cannot due to the fact that *Export_T_Ids* is initially empty. When **New_Export_Request** occurs, the new Export Operation is put into *Export_Operation*, ready to be processed by **Export_Opn**'s sub-page (Export#19) which is described in section 6.11.7.

Unused_T_Ids contains a list of unused Transaction Identifiers which can be used by the Query and Export operations to uniquely identify tokens related to a specific transaction being processed. It is initialised with a list of 3 identifiers as shown below.

1'[1,2,3]

When a `t_id` is being used by the Query operation, it is put into *Query_T_Ids*. This allows the `Funct_Opns#9` page to detect when tokens associated with an in-process transaction are forwarded through to the Functional Operations via *To_Trader*. The same logic applies for the Export Operation and *Export_T_Ids*.

When the **Query_Opn** and **Export_Opn** sub-pages have finished processing, they can create message tokens for sending to external objects via the interfaces. These tokens are put into *To_Interfaces* and exit this page. They also return used transaction identifiers to the list located in *Unused_T_Ids*.

6.11.6 Query Operation

The Trader's Query operation is modelled on the `Query#13` page which is shown in figure 6.23. It is the most significant page in the Trader model, detailing the steps that a Trader performs when it attempts to find matching offers for a Query. The Trader Standard [16] specifies the steps which occur whilst performing a Query Operation from the Computational viewpoint.

This page uses *Query_Operation*, *To_Interfaces*, *Trader_Imp_Attrib*, *Query_T_Ids* and *Unused_T_Ids* as port places for token I/O. When a message arrives for the Query operation, the guard on the Check Parameters transition only allows messages whose operation field is equal to `query` to proceed.

From left to right, the vertical streams of token flow have the following functionality:

- The initial Import Request is checked to see if its parameters are correct. This has been modelled through the use of non-determinism. A boolean variable `result` of colour `Test` (which may take only two values : `FAIL` or `PASS`) is bound to one of its values arbitrarily. The effect is that the transition will pass or fail the test non-deterministically, eliminating the need to write a function to perform the test. At this level of abstraction in modelling the Trader, we are interested only in the behaviour under certain conditions, not the implementation to determine whether parameters are correct or not. When the security check is passed non-deterministically, a `t_id` token is removed from the pool of unused identifiers and inserted into the import request. At the same time, the import request's `t_id` is saved along with the new `t_id` in *Request_T_Ids*, so it may be restored when processing is complete. When the message passes both the Parameter and Security checks, it enters the **Process_Req#14** sub-page. As described in section 6.11.6.1, this page checks to see if the Query is a duplicate of one that has already been processed and creates messages for the OfferSpace and LinkSpace objects in *To_Interfaces* as necessary.
- If the LinkSpace object was queried, it responds with the Trader's links allowing **Accept_Trader_Links** to become enabled. When it occurs, the links are extracted from the message and passed into the `Link_Import#15` sub-page which is described in section

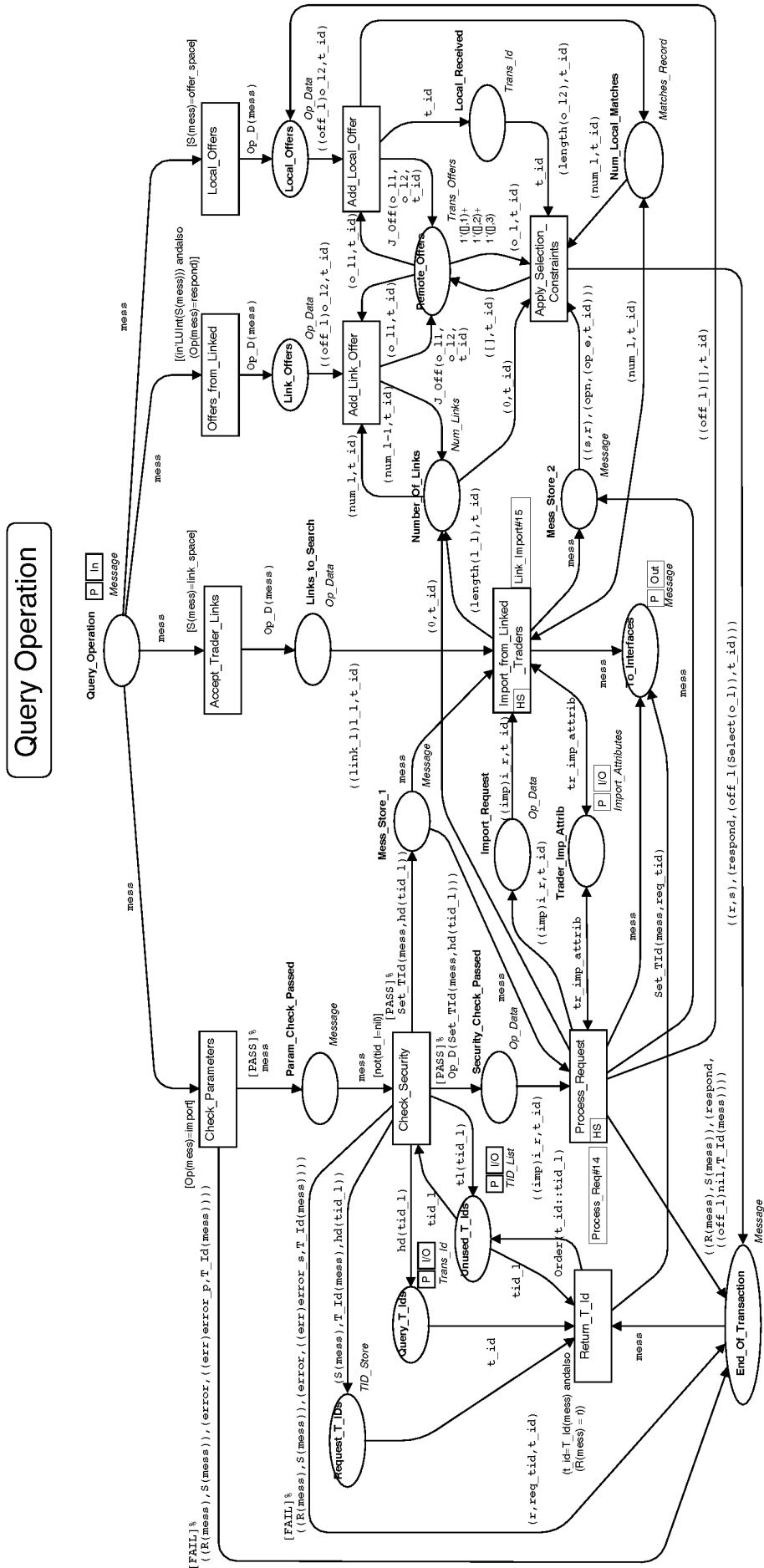


Fig. 6.23: Query Method (Query#13)

6.11.6.2. It creates messages to be sent to linked Traders via *To_Interfaces* as dictated by the unified Trader, Link and Query policies. A counting token in *Number_of_Links*, which maintains the number of links remaining to be searched, is updated based upon the number of Queries which were forwarded to Linked Traders. The initial message is also moved to *Mess_Store_2* for later use.

- When a linked Traders replies, **Offers_from_Linked** becomes enabled. When it occurs, a list of matching offers is extracted from the message and put into *Link_Offers*. When **Add_Link_Offer** occurs, the new offers are added to an initially empty list in *Remote_Offers*, and the counting token in *Number_of_Links* (used to indicate how many linked Trader responses are still outstanding) is decremented by 1. When the Trader receives a response from the OfferSpace object regarding its own offer space Query, **Local_Offers** becomes enabled. When it occurs, the matching local Offers are put into *Local_Offers*. Local Offers are added to the initially empty list contained within *Remote_Offers* when **Add_Local_Offer** occurs. **Apply_Selection_Constraints** cannot become enabled until the Trader has received a reply for its search of the OfferSpace, and the number of outstanding link requests has reached zero (contained in *Number_of_Links*). Thus, it will wait until all responses have been received and when it occurs, will apply selection constraints to the list of matching offers using the `Select ()` function in order to return the *best* offer(s) to the Importer. When processing is complete, **Return_T_Id** occurs which returns the `t_id` used whilst servicing the request to the pool of *Unused_T_Ids* for subsequent re-use.

This page is a top-level page for the logic associated with managing and processing a Query and as such, models the Trader's most fundamental function – finding matching Offers for Queries.

6.11.6.1 Process Request

This page (Process_Req#14) is a sub-page of **Query#13** (shown in figure 6.24) and is instantiated once for each instance of its super page. It models detection and handling of duplicate Query requests. The input to the page is from *Security_Check_Passed*, which contains `Op_Data` of requests that have already passed parameter and security checks on the **Query#13** page.

The *IWT_Request_Store* place is used to maintain a record of all interworking requests that the Trader has serviced. The Trading Standard [16] states that only *recent* requests should be stored. No definition for recent is given which leaves the specification ambiguous. It also states that only **Request_Ids** are stored which was shown in section 6.5.3 to be insufficient to ensure deterministic traversal of the entire offer space whilst interworking. It was decided to maintain a record of all requests in *IWT_Request_Store* rather than only recent requests, although it would

be possible to alter the model so that a finite number of requests are recorded. The colour set definitions required for *IWT_Request_Store* are shown below.

```
(** List of request identifiers used by the Trader to record Queries
    that have been serviced **)
color Req_Id_List = list Request_Id;

(** List of import policies **)
color Imp_Policy_List = list Imp_Policy;

(** Function that returns a boolean value regarding whether the r_id
    parameter already exists within the r_list import policy list.
    Used on page Process_Req#14 **)
fun Req_Id_Exists(r_id:Request_Id,r_list:Imp_Policy_List):bool =
  if(null(r_list)) then
    false
  else if(#request_id(hd(r_list))=r_id) then
    true
  else
    Req_Id_Exists(r_id,tl(r_list));
```

IWT_Request_Store (Interworking Trader Request Store) contains a list of Import Policies, which among other elements, contains a Request_Id (as defined in section 6.10.3). *IWT_Request_Store* is initialised to contain an empty list. If a new Query is received, then **Is_Unique** becomes enabled since the Req_Id_Exists() function returns FALSE in the guard (i.e., the Request_Id does not already exist in the list of Import Policies).

```
fun OrderIP(iplist:Imp_Policy_List):Imp_Policy_List=
  if (length(iplist)=1) then
    iplist
  else
    let
      val head = #t_ident(#request_id(hd(iplist)))
      val tail = tl(iplist)
    in
      if (head < #t_ident(#request_id(hd(tail)))) then
        hd(iplist)::OrderIP(tail)
      else
        hd(tail)::OrderIP(hd(iplist)::tl(tail))
    end;
```

When **Is_Unique** occurs, the Import Policy of the new Query (*i_p*) is added to the head of the list of existing Import Policies (*seen*) and then ordered using OrderIP(). This function returns an ordered Import Policy list which is returned to *IWT_Request_Store*. It is important to use a function to re-order lists throughout the model since a list containing the same elements but in a different order is considered to be a different value by the OG and thus, increases the state space accordingly. A similar approach was used in Query#13 (see section 6.11.6) when returning used *t_ids* to the list in *Unused_T_Ids*.

Alternatively, if the new Query has already been processed, then **Is_Duplicate** becomes enabled since Req_Id_Exists() returns a non-zero value. When **Is_Duplicate** occurs, a token is created by Dup_Match() in *Duplicate_Message*, containing the new Query's import request and the import policy of the Query which has already been processed.


```

(** Function that searches a list of Import Policies and returns the element
    that matches the import policy parameter.
    Assumes there is a duplicate to match. **)
fun Dup_Match((st,i_pol):Imp_Req,ip_list:Imp_Policy_List,t_id:Trans_Id):Dup_Data =
  if (#request_id(hd(ip_list))=#request_id(i_pol)) then
    (hd(ip_list),(imp)(st,i_pol),t_id)
  else
    Dup_Match((st,i_pol),tl(ip_list),t_id);

```

When **Is_Duplicate** occurs, `Update()` is used on the list of seen Import Policies and the Import Request being processed (`i_r`). This updates the seen list so that the existing Import Policy is replaced by the new Import Policy if it has a greater `hop_count` than the existing Import Policy.

```

(** This function assumes that a duplicate has been found, and therefore r_list is a
    non-empty list **)
fun Update((st,ip):Imp_Req,r_list:Imp_Policy_List):Imp_Policy_List =
  if (#request_id(hd(r_list))=#request_id(ip)) then
    if (#hop_count(hd(r_list))) > (#hop_count(ip)) then
      r_list
    else
      ip::tl(r_list)
  else
    hd(r_list)::Update((st,ip),tl(r_list));

```

If the `hop_count` of the new Query (contained within `ip`) is less than that of the old Query (contained within `ip_1`), then there is no need for the Query to be forwarded to linked Traders, therefore **No_Link_Search_Reqd** becomes enabled. This is because there is no possibility of reaching more Traders than the Query which has already been sent to Linked Traders, as previously discussed in section 6.5.2. At this point, the Query processing is complete since no local searching is required and no linked Traders are queried. Thus, the Trader creates a response message returning an empty list of matching Offers which is put into **End_Of_Transaction**.

If the `hop_count` of the new Query (contained within `ip`) is greater than that of the old Query (contained within `ip_1`), then **Link_Search_Reqd** becomes enabled, indicating a need to forward the Query to linked Traders. When this occurs, the token in **Processing_Duplicate** which corresponds to the transaction identifier being utilised is marked TRUE, indicating that a duplicate is being processed. Also, the original Query (received through **Security_Check_Passed** at the start of processing) is forwarded to **Process_Message** for further processing.

When **Unify_Policy** occurs, the Trader and Query policies are merged (as discussed in section 3.4.8.4) by `merge_policy_options()` and put into **Reduced_Policy**. The Import_Request is moved into **Import_Request_Store**.

```

(** Function that calculates the merged policy based upon the Trader,
    Link and Import Policies **)
fun merge_policy_options(i_p:Imp_Policy, trad:Import_Attributes,
t_id:Trans_Id):Follow_Record=
  if(#link_follow_rule(i_p)<>no_option) then
    (smallest_option(#max_follow_policy(trad),

```

```

        #link_follow_rule(i_p),always),t_id)
else
    (smallest_option(#max_follow_policy(trad),
        #def_follow_policy(trad),always),t_id);

```

If the `hop_count` of the Import Policy equals zero, or the merged policy equals `local_only`, then there is no need to forward the Query to linked Traders. There is, however, a need for a local search. If the Query is also a duplicate, then a local search of the OfferSpace is not necessary and thus, **Local_Search_Only** creates a response message that returns a `nil` list of matching offers to the object that initiated the Query. If the Query is unique, then when **Local_Search_Only** occurs, a message is created for the OfferSpace object requesting it to `get_offers`.

If the `hop_count` of the import policy is greater than zero and the `merged_policy` is not equal to `local_only`, then there is a need for both a link and local search to be performed. Thus, **Link_andor_Local_Search** becomes enabled. If the Query being processed is a duplicate, then there is no need to perform a local search, but the Trader's links still need to be obtained from the LinkSpace object. When the merged policy equals `always`, only a link search is required. This means that the Trader sends a message to the LinkSpace object requesting it to `get_links`, and puts an empty list of offers in **Local_Offers** since no local OfferSpace search is required.

If the merged policy equals `if_no_local`, then the Trader must perform a local search of the OfferSpace to determine whether the Query has a local match. In this situation, the Trader issues a message to the OfferSpace object requesting it to `get_offers` and also sends a message to the LinkSpace object requesting it to `get_links`.

6.11.6.2 Import from Links

Link_Import#15 (shown in figure 6.25) is a sub-page of Query_Opn (Query#13) and is assigned to the **Import_from_Linked_Traders** substitution transition. It is instantiated once for each Trader. The page models the actions that a Trader performs when it initiates interworking with linked Traders. The first transition in the page is **Process_Links** which requires the following information from input places:

- the Import Request which was extracted from a Query sent by an Importer or another Trader. A token that contains this information is consumed from **Import_Request**.
- the Trader's Import Attributes which indicate the Trader's policy with regard to Import Queries. This information is obtained from **Trader_Imp_Attrib** and returned to the same place after **Process_Links** has occurred.
- a list of links for the Trader which is taken from **Links_to_Search**.
- the original message that was received from either an Importer or another Trader. A token containing this information is taken from **Mess_Store_1**.

- `Resolve_Local_ip()` – creates a message for each link that contains the resolved import policy for that link which is to be used for local decision making regarding Query propagation,
- `Create_PO_Mess()` – creates messages that are ready to be sent to linked Traders,
- `Bidirectional_Link()` – detects when two Traders are connected to each other.

For a complete CPN/ML listing of these functions, refer to Appendix A.

On this page, the Trader processes its links and forwards requests to linked Traders where appropriate. Forwarded requests may have a different follow policy than the initial policy which was used to determine whether the link should be followed. This is why two policies are calculated: one for local decision making, another for forwarding with Queries to linked Traders.

When **Process_Links** occurs, the original Query message, the links that are to be searched, the Trader's Import Attributes and the Import request extracted from the Query message are used to create the following:

- a set of messages addressed to each of the linked Traders. These messages contain the local import policy and are put into *Link_Messages* when `Resolve_Local_ip()` is evaluated. These tokens are used for local decisions making regarding whether to forward a Query along a specific link or not.
- the number of links that should be followed by the Trader is put in *Number_Of_Links*. This value depends upon the result of `Bidirectional_Link()` which returns a boolean that indicates whether the Trader that sent the original Query message is connected to this Trader via a bi-directional link. If this is the case, then the number of links to be traversed is one less than the total number of links that the Trader has. This is because a Trader never forwards a Query back to the Trader from which it came, as proposed in section 6.5 regarding the Revised Interworking Protocol.
- a set of messages addressed to each of the linked Traders. These messages contain the pass on import policy and are put into *Link_Pass_On_Messages* when `Create_PO_Mess()` is evaluated.
- the original message is put into *Mess_Store_2* for use later in the CPN.

When **Process_Links** has occurred, either **Select_Always** or **Select_If_No_Local** can become enabled, depending upon the `link_follow_rule` of the message. If it is equal to `always`, then a message is taken from *Link_Messages* and put into *Link_Searches_To_Reduce* when **Select_Always** occurs. If the `link_follow_rule` is equal to `if_no_local`, then a message is selected from *Link_Messages* and moved into *Requests_INLocal*.

Send_If_No_Local will become enabled when *Num_Local_Matches* contains a record for the current Query that is being processed. If the Trader succeeds in finding a local match for the Query, then *num_matches* is bound to a non-zero value. This means that the Trader should not forward any requests to linked Traders. Thus, when **Send_If_No_Local** occurs, a message which is bound to *pass_mess* and is ready to be passed on to a linked Trader is instead consumed from *Link_Pass_On_Messages*. The token in *Number_Of_Links* is replaced with its value reduced by one. This transition will continue to become enabled and occur for all messages that are intended to be sent to linked Traders if the local OfferSpace search failed to locate a matching Offer.

However, if *num_matches*=0, this indicates that the Trader has been unable to match the Query locally and is required to forward the Query to its linked Traders where possible. Thus, when **Send_If_No_Local** occurs, the Query is put into *Link_Searches_To_Reduce*. The final transition on this page is **Import_From_Link** which puts the messages that contain the pass-on policy into *To_Interfaces* which is an output port for this page. This transition is used to remove tokens from *Link_Searches_to_Reduce* when the messages in *Link_Pass_On_Messages* are sent to linked Traders. Thus, message tokens in *Link_Searches_To_Reduce* act as a flag, indicating that Queries in *Link_Pass_On_Message* are to be sent to linked Traders.

This model is made significantly complex due the fact that the Trader requires a unified local import policy which it uses to make decisions on which links can be used and **also** requires a link *pass_on_policy* which is used in the messages which are propagated to linked Traders. Two different policies are required because the Trader supports default follow policy values which are used when no *follow_policy* value is supplied with a Query.

Another factor which makes this page more complex is support for the *if_no_local* policy requires messages to be destroyed if they are not required to be sent to linked Traders (i.e. when a local match is obtained).

6.11.7 Export Operation

The Trader's Export Operation is modelled on Export#19 which is shown in figure 6.26. This page is responsible for processing Export operations when they are sent to the Trader via the Register Interface (REGI#20). It is a sub-page of Funct_Opns#9 and is the dual of Query#13 which processed Query operations.

As with Query#13, a pool of available *t_ids* is maintained in *Unused_Tids*. When a new Export operation is received via **Export_Operation**, the message's current *t_id* is saved in *T_Id_pairs* along with the new *t_id* when **Export_Operation** occurs. This is to ensure that the transaction utilises a unique *t_id* within the Trader. As a result of **Export_Operation** occurring, the *Op_E()* of the message is extracted and sent in a new message to the OfferSpace object instructing it to *add_offer*.

When the OfferSpace object replies to the *add_offer* message, a message token is put into **Export_Operation**. When **From_Offer_Space** occurs, the response message's *Op_D()*

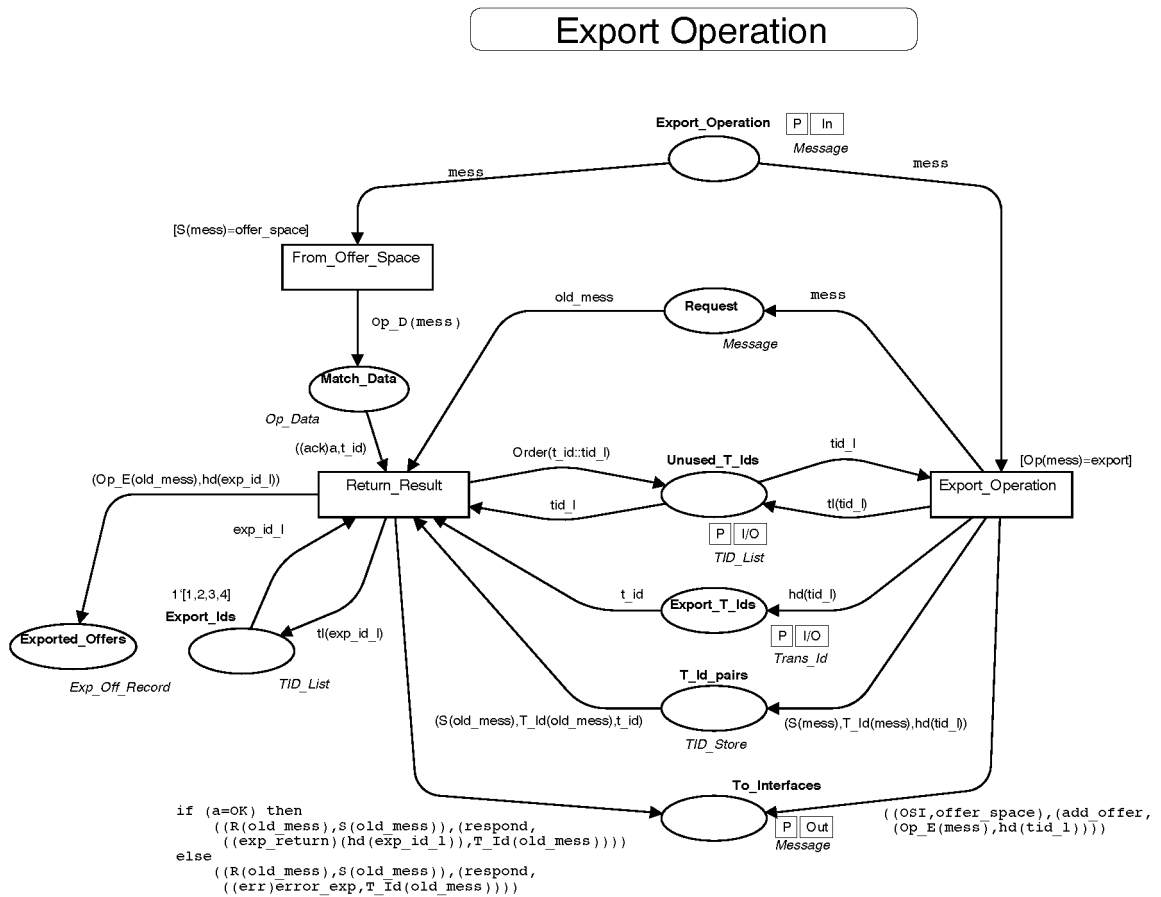


Fig. 6.26: Export Operation (Export#19)

extracted and put into *Match_Data*. *Export_Ids* contains a list of t_ids which are used to uniquely associate an identifier with each Exported Offer. When **Return_Result** occurs, depending upon the OfferSpace object's result, either a unique *Export_id* or an error indicating an export error is returned to the Exporter. A copy of the *Export_Id* and the associated offer are maintained by the Trader in *Exported_Offers* which allows withdrawal and modification of offers to be added to the model at a later date if necessary.

6.12 Refining the Model

The model presented in this chapter has evolved over the space of three years to reach its current form. This section details some of the issues raised during the evolutionary modelling of interworking Traders and their environment.

6.12.1 Stages of Modelling

As the Trading standard evolved, it was necessary to re-model the Trader to reflect changes in the Trading standard. Initial background work on the Trader was based upon the textual description of the Trader included in Appendix A (Trading Tutorial) of the Draft ODP Trading Function document released in 1993 [9]. Modelling of the Trader was based upon the 1994 Draft ODP Trading Function document [10]. This document included a Tutorial on the Trader in

Annex A, and a Message Sequence Chart (MSC) description of the Trader's Operation in Appendix B. These documents were used extensively in the first CPN model of the Trader, which combined the text-based description of the Trader with the graphical power of the MSCs.

In Annex B, a Search Policy object was defined that was able to interrupt the search when the Search Policy dictated such as, for example, due to a timeout. This object was removed in the 1996 Trading Standard [15], since the concept of a `hop_count` was introduced to ensure that the propagation of requests is finite and controllable. Initial modelling of the Trader using CPNs and the 1994 standard resulted in two publications [26, 27].

Using the 1995 release of the Trading Standard [14] the model was revised to include the Trading Service Interface (TSI) which was more precisely defined in the 1995 Standard. A TSI effectively provides the functionality of the Lookup and Register interfaces in one and defines a separate client and server role for Interworking. The standard did not adequately specify how Traders were to interwork, so an Interworking technique which required Traders to append their unique object identifiers to a list of *seen* traders within a Query was developed, modelled and tested. Based upon the revised model, a number of papers were published [29, 31, 32].

In 1996, the RM-ODP Trading standard was brought into technical alignment with the CORBA Object Trading Service specification [45]. This resulted in a radically different Trading architecture being adopted involving:

- removal of TSI's from the specification,
- creation of new Interfaces (Lookup, Register, Admin, Proxy and Link),
- creation of a new Interworking Protocol involving a `hop_count` and associated `Request_Ids`.

Much of the logic involved in Queries in Query#13 was re-used, however a number of new pages were required (i.e. `Process_Req#14` and `Link_Import#15`). In addition, a completely new architecture utilising a hierarchy of interfaces was developed which allowed the model to accommodate interfaces to new objects and new methods as required. Finally, the model was altered to allow multiple threading to be incorporated into each object, thereby greatly increasing the level of concurrency within the model.

The 97 Trading Standard [16] involved minor textual changes and did not impact on the model developed using the 1996 Trading Standard.

6.12.2 Object-based Modelling

Initially, the system was modelled as a group of entities that communicated with each other through a fixed communications medium. These objects were not explicitly modelled to utilise interfaces or provide methods for other objects to use. Thus, they did not properly model RM-ODP objects since interfaces are a fundamental component of the RM-ODP object model. The addition of interfaces allowed the functionality of objects (methods) to be separated from the mechanics of object message passing (as discussed in sections 6.4.1 and 6.4.2).

Having modelled inter-object message passing, it was possible to make changes to object functionality (methods) without requiring modification of associated interfaces, except where colour sets were modified as discussed in section 6.12.3.

6.12.3 Modifying Colour Set Definitions

As the model evolved to reflect the changing Standard, it became necessary to alter colour sets. The structure of the `Message` colour set remained constant throughout the lifetime of the model, but radical changes in the model resulted in new colour sets, functions and variables being added at each iteration. The color sets were intended to be flexible since many were created using cartesian product notation which is easy to extend (simply add a new element to the tuple). However, this also resulted in significant inscription changes since inscriptions which assumed a specific token structure were made invalid. Utilising projection functions wherever possible was also beneficial since it reduced clutter in the models and also increased “portability” since token structure was not explicitly stated in inscriptions.

6.13 Comments on High-level Petri Nets and Design/CPN for Modelling OOBDS

In this section, an examination of some positive and negative points regarding High-level Petri Nets and the Design/CPN tool are presented.

6.13.1 Petri Nets

High-level Petri Nets are useful for modelling OOBDS since they provide support for modelling:

- concurrency in a natural and intuitive manner, and
- non-deterministic behaviour.

They may also be used to model a system at multiple levels of abstraction which is useful since it allows different viewpoints of the same system to be examined.

However, High-level Petri Nets have the following drawbacks at present:

- lack of a finalised International Standard [88] which would facilitate a unified approach to the development of tool inter-operability and High-level Petri Net semantics, and
- not widely recognised in industry as a valuable tool for Software Engineering.

6.13.2 Design/CPN

6.13.2.1 Positive Points

- *Hierarchical*: allows modularisation and re-use of pages when modelled in a generic manner.
- *Simulator*: the tool provides a simulator which is very valuable for testing models.
- *online help* and FAQs are available via the Design/CPN website [83] and often, direct feedback is provided by the tool's developers when assistance is sought.
- *ongoing development* of the tool assists in improving the tool's stability (bug-fixes) and the addition new features such as libraries.

6.13.2.2 Negative Points

- *Unstable*: It is possible for the tool to crash, resulting in lost work. The tool relies on a binary file format for saving models which can result in complete loss of a model if its file becomes corrupted. However, the tool does provide a roll-back capability since it creates backup files of models.
- *Requires Expertise in ML*: For the uninitiated, CPN/ML (based on the ML language) syntax can be non-trivial and may require significant practice to become a proficient user.
- *User Interface*: the tool's user interface requires updating to include for example, a tool bar and context-specific right mouse button pop-up menus for smoother editing. Also, the Linux X-Windows version requires the user to initiate screen refreshes due to residual graphics on the display.

6.14 Summary

In this chapter, a CPN model of the computational viewpoint of the RM-ODP Trader has been presented. The model makes use of hierarchical CPN constructs in Design/CPN to provide a modular view of the Trading environment. In the model, Traders utilise interfaces and a communications medium when interacting between objects. The model allows multiple Traders to be instantiated, where each Trader is capable of servicing multiple requests concurrently. This provides a basis for modelling the Trader interworking protocol.

The effects of evolutionary modelling of a changing standard was discussed, along with an assessment of High-level Petri Nets and the Design/CPN tool for modelling the OOBDS. The model presented in this Chapter will be analysed in detail in Chapter 7.

Chapter 7

Analysis of the Trader

In this chapter, the CPN model of the Trader and its environment presented in Chapter 6 is analysed. Firstly, the Trader is analysed as a standalone entity that is capable of providing services to clients (Query and Export). Next, it is shown that concurrent threads of execution within a Trader instance are independent. This demonstrates the Trader's ability to service multiple independent concurrent requests.

Having analysed the Trader's functionality as a standalone entity, the improved Trader interworking protocol suggested in section 6.5 is analysed. Correct operation of multiple Trader instances that exist independently and can service concurrent requests is verified. Finally, Traders are linked in a number of scenarios which verify the stopping conditions for Query request propagation.

7.1 Methodology for detecting errors in the model

Analysis of the Trader is performed in two stages, the first of which is simulation and the second, Occurrence Graph Analysis (OGA). Occurrence Graph Analysis with Equivalences can also be used to reduce the size of the OG in certain scenarios. Each of these techniques may be used to detect errors in the CPN model.

7.1.1 Examples of modelling errors

Errors can be traced to a number of sources such as transition guards, arc inscriptions associated with transitions, or SML code contained within arc expressions. An error associated with a guard usually manifests itself by the incorrect enabling of a transition. In some cases, this results in premature or incorrect firing of transitions, or transitions that are never enabled due to an overly strict guard which never evaluates to TRUE.

Arc inscription errors usually resulted in tokens with incorrect values being used or created when transitions occur. If the arc inscription contained a variable that did not have a deterministic binding, then the simulator was able to bind it to any legal value in the variable's colour set. This resulted in non-deterministic data values being introduced into the model, thereby corrupting the organised flow of information in the model. Such errors were sometimes difficult to locate until OGA techniques were applied as described later in this section.

The most difficult errors to identify in the model were those associated with SML code executed as part of a transition firing. In such situations, if the SML code creates erroneous output it cannot be detected by Design/CPN's static syntax checking. Thus, a syntactically correct model would result in run-time errors during CPN execution. In order to locate these bugs, step-by-step execution of the model was required to pinpoint the transition associated with the error. Having identified that SML code was in error, it was helpful to execute the SML code in question in an auxiliary box using sample data. This technique was also used for the rapid creation of SML code functions since it allows functions to be written and tested without requiring a complete re-build of the GDN.

7.1.2 Simulation

Using the simulator, the model was executed step-by-step making it possible to locate errors as transitions occurred during the token game. When performing simulations, there are two possible modes of operation: manual or automatic. Using manual simulation, the user is able to select the next transition to occur and also the bindings used for its enabling. This was a useful debugging technique since it provided immediate feedback on the dynamic behaviour of the model.

Automatic mode is a much faster way to perform simulations since the simulator selects transitions and their bindings randomly, thereby ensuring that the simulation run is not affected by the modeller's bias with respect to the ordering of transition occurrences. By careful observation of the model's behaviour and the generation of a simulation report, it is possible to determine transitions that are in error.

7.1.3 Occurrence Graph Analysis

When the model successfully completed scenarios in the simulator, the next step is to perform OG Analysis, effectively performing all possible simulation runs at one time. When analysing the Trader model (which possibly contained modelling errors that result in huge OGs), the technique adopted for generating OGs was a combination of breadth-first and subsequent depth creation of the OG.

By altering the OG generation branching factor via Design/CPN's `OGSet.BranchingOptions` function, it was possible to limit to 1, the number of bindings and transition elements considered when generating the next node in the graph.

In practice, this meant that for an initial period of time, the OGA was directed to generate the OG breadth first. This provided a large number of markings that were reached from the initial

marking and could be subsequently investigated through depth-based generation of the OG. The application of a breadth-based followed by depth-based generation of the OG allows it to narrow in on a final terminal marking which should be reachable from all nodes in the OG.

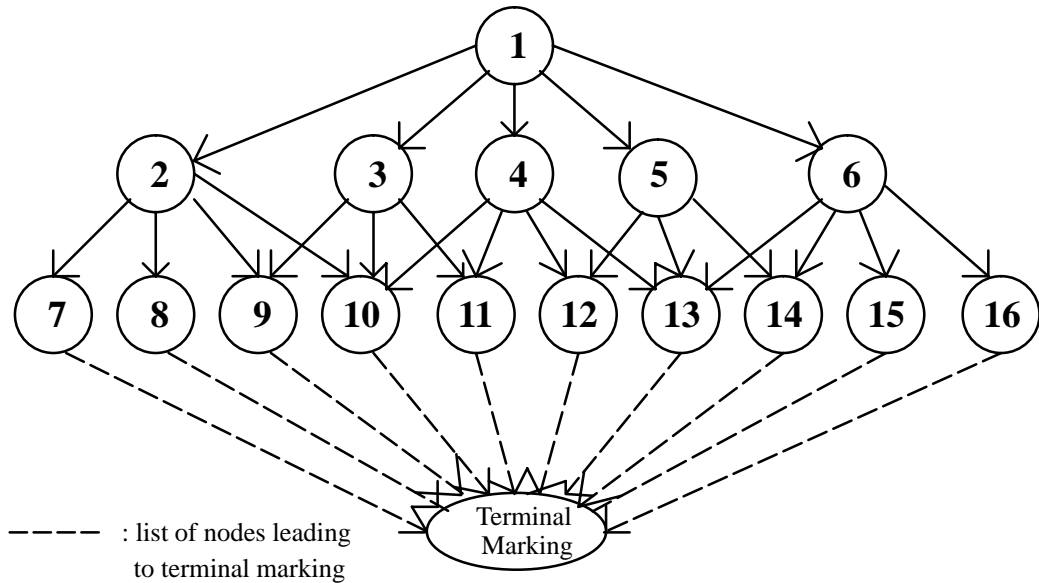


Fig. 7.1: Calculating the OG using Breadth then Depth

With models that are expected to terminate in the same state after all simulation runs, the OG contains a single dead node (terminal marking). This is because all of the paths that lead from the first marking of the model (node 1 in the OG) eventually reach the dead node. This node is also a home marking [66] since it can be reached by all other reachable markings. If the OG contains a single dead node and its marking is correct (based upon the expected behaviour of the scenario), then the scenario has been verified.

Since the aim of the OG analysis of the model is to detect all of the dead nodes in the OG (although a single dead node is expected), this technique increases confidence in the model since dead nodes are calculated much earlier. Even though not all nodes have been calculated, the fact that all paths that have been calculated lead to the same dead node is a promising sign. If there is more than one dead node in the OG after using this technique, then it is safe to assume that the model is flawed. It is then possible to investigate the bindings and transitions that resulted in entering multiple dead nodes, which leads to the cause of the modelling error.

When analysing the Trader, it became evident that modelling errors related to distinguishing between tokens within threads allowed state information of threads to become confused. This manifested itself when tokens belonging to a given thread were used by a different thread. In this situation, both threads became corrupted and the model ceased to function correctly.

Simulation runs were less capable of finding these errors since, in most cases, they did not test the case where concurrent requests are at the same stage of processing within the model, the source of most errors. OG analysis of the model was much quicker and more effective at detecting concurrent thread errors.

7.1.4 Occurrence Graphs with Equivalence Classes

As demonstrated in section 5.4.5.3, it is possible to use OGs with Equivalence classes to reduce the size of an OG. The same technique can be applied to the model described in Chapter 6, where equivalence classes are used to map out the `t_ids` used by Traders when servicing multiple requests concurrently. This is useful since it can greatly reduce the size of the OG because `t_ids` are assigned to Queries on a first-come first-served basis, but the arrival order of Queries is non-deterministic. This results in an OG which includes the possibility of different `t_ids` for each Query which enlarges the size of the OG without providing extra analytical benefit. This is because the value of a Query's `t_id` must be unique within the Trader, but its value is unimportant.

In the model of the Trader presented in Chapter 6, a unique transaction identifier is associated with each Query as it is processed. When processing multiple Queries, it is possible for them to arrive in any order (2! possibilities with 2 Queries, and 3! with 3 Queries). This means that there are multiple paths through the OG which represent the same Trading behaviour except for the use of a different value for the Query transaction identifier.

Having transaction identifiers allows the Trader to differentiate between tokens that are used by multiple threads of execution, but the threads operate in exactly the same way irrespective of the unique transaction identifier's value.

OGs with Equivalence Classes can be used when generating an OG to detect markings that are considered equivalent and to represent them using a single node in the OG, thereby reducing the size of the OG.

Projection functions (`EquipBE()` and `EquipMark()`) which map out the `t_id` from tokens (such as those of colour set Message and Op_Data) are included in Appendix A. Using these functions, the OEOS tool was used to create reduced Equivalent OGs when the scenario included concurrent Query servicing by a Trader.

In the case of a modelling error associated with concurrent threads in the model, the OG generation resulted in more terminal markings than expected. These are easily obtained using the standard CPN/ML library function `ListDeadMarkings()`.

7.2 Permutations of `merge_policy_options()`

When unifying the three policies (Trader, Import and Link) at `Unify_Policy` on `Process_Req#14`, (see sections 6.11.6.1 and 3.4.8.4), the three policies are passed to `merge_policy_options()`. In this section, we verify that the order in which these parameters are passed to the function does not effect the function's return value. This is important since it means that all permutations of the three policy values need not be investigated in separate scenarios. The first step in the analysis is to investigate `smallest_option()` which is used by `merge_policy_options()`.

There are three possible follow policy values that can be returned by the `smallest_option()` function. They are from strongest to weakest:

- `always`
- `if_no_local`
- `local_only`

The functionality of the Trader when `smallest_option()` evaluates to each of these values must be verified. This is accomplished by verifying that the return values of the `smallest_option()` function are as specified by the rules given in the Trading standard [16] section 8.2.7.6. The function's SML source code is shown below.

```
(** Function that finds the smallest follow option based upon the rules in the
    Trading Standard (1996) section 8.2.7.6 Link Follow Behaviour **)
fun
smallest_option(o1:Follow_Option,o2:Follow_Option,o3:Follow_Option):Follow_Option =
  if ((o1=local_only) orelse (o2=local_only) orelse (o3=local_only))
  then local_only
  else if ((o1=if_no_local) orelse (o2=if_no_local) orelse (o3=if_no_local))
  then if_no_local
  else
    always;
```

By examining the CPN/ML code for the function, it can be seen that the order in which the values are passed as parameters (`o1`, `o2` and `o3`) does not affect the result of the function call. When using the function with only two parameters, the third parameter to the function call should be `always`, which does not influence the function's return value.

```
since it is known that: (always > if_no_local > local_only), then

smallest_option(always, if_no_local, local_only) = local_only
smallest_option(always, always, if_no_local) = if_no_local
smallest_option(always, always, always) = always
```

This means that if at least one of the policies equals `local_only`, then the function will return `local_only`.

In `Process_Req#14`, there are only two policies which are to be unified, since only the Import and Trader policies are available at that point. In this situation, the model uses the `merge_policy_options()` function (which uses `smallest_option()`) as defined below:

```
(** Function that calculates the merged policy based upon the Trader,
    and Import Policies **)
fun merge_policy_options(i_p:Imp_Policy, trad:Import_Attributes,
t_id:Trans_Id):Follow_Record=
  if (#link_follow_rule(i_p)<>no_option) then
    (smallest_option(#max_follow_policy(trad),#link_follow_rule(i_p),
    always),t_id)
  else (smallest_option(#max_follow_policy(trad),#def_follow_policy(trad),
    always),t_id);
```

When the link follow rule of the Query Import policy is not set, then `smallest_option()` is invoked using the Trader's default follow policy.

Having verified the function's behaviour, the model requires analysis for each of the three possible return values for `smallest_option()`, rather than for all possible permutations of input parameters to the function. This greatly reduces the number of scenarios that need to be considered (permutations of the parameters) since they have been taken into account during the function's verification.

7.3 Initialisation of the Trading Environment

Before any simulation or analysis can take place, the Trading Environment must be initialised by declaring initial markings for objects in the system. For all Trading scenarios, the following initialisations are required:

- the Trader requires its trading policy in *Trad_Imp_Attrib* on Trad_Int#3 to be initialised, as explained in section 6.11.1,
- the OfferSpace object requires a set of offers to be placed in its database (*Offer_Storage* on Offer_Space#4) for each Trader being verified, as explained in section 6.10.1,
- the LinkSpace object requires a record of links for each Trader in the Trading Environment (stored in *Link_Storage* on Link_Space#5), as explained in section 6.10.2.
- a Query request must be generated by the Importer in *Import_Request* on Importer#6. This Query specifies the service type and service parameters which need to be matched by the Trader.
- for each instance of the interfaces (LUI, REGI, OSI and LSI) a unique object identifier must be assigned. This is performed by using the change marking item from the **Sim** menu in Design/CPN. As the instance identifiers are initialised to the value of the first instance (i.e., LUI_1 or OSI_1), they must be altered for each instance of the Trader in the highest level of the model in Trad_Env#2.

These initialisations are required so that:

- all Traders have their own Trading policy,
- there are offers in the OfferSpace object,
- there are Links in the LinkSpace object which can be pursued when Traders interwork,
- there is at least one Query available for processing by the Trader,
- all interfaces in the system have a unique instance identifier.

Depending upon the scenario, the initialisation of the system will be different. So, for a scenario in which three Traders are to interwork, the LinkSpace object must be initialised with links required for the scenario. All object instances within the three Traders must also have unique identifiers.

7.4 Approach to Analysis of the Trading Environment

Analysis of the CPN model of the Trading Environment aims to show the following:

- 1 correct operation of a Trader which does not interwork (*standalone*) processing a single Query.
- 2 correct operation of a standalone Trader with *multiple concurrent* Queries (internal multi-threading).
- 3 correct operation of *multiple* Traders operating concurrently and *independently*.
- 4 identify and ensure correct behaviour for all *stopping conditions* required for Query propagation.
- 5 analysis of *arbitrarily complex* Trading Environment configurations.

Some of these points are depicted in figure 7.2.

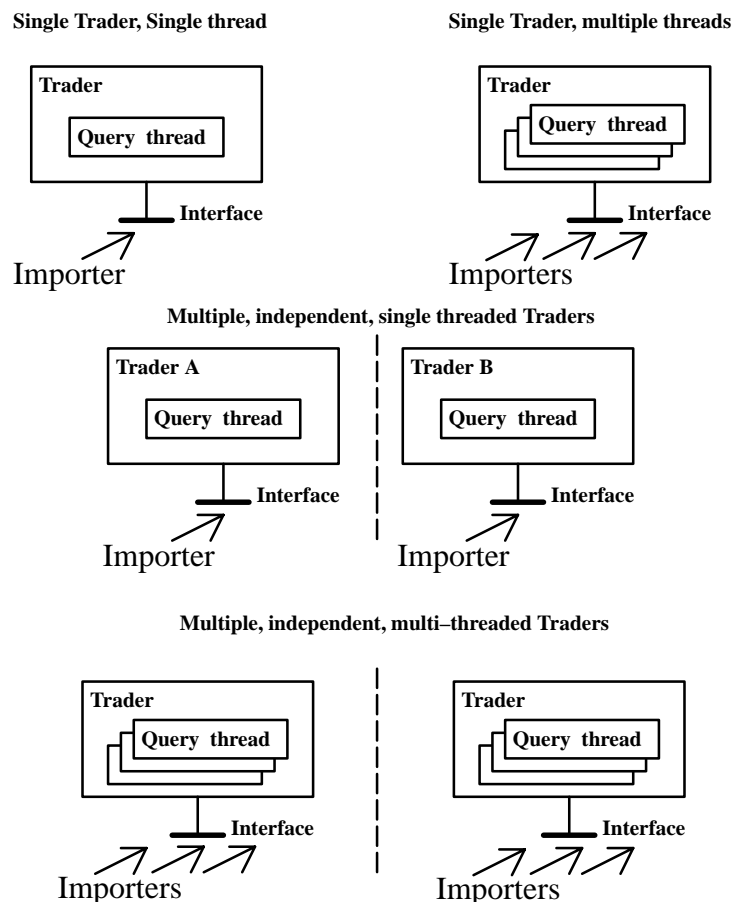


Fig. 7.2: Analysis goals

7.5 Analysis of a Trader Configured as a Standalone

A fundamental step in the analysis of interworking Traders is the analysis of a Trader which is configured to be a standalone Trader. This means that the Trader does not interact with other Traders when processing a Query. The analysis must include all permutations of relevant initial conditions and the servicing of multiple requests by concurrently executing threads of control. In order to verify the correct operation of the Trader's Query operation as a standalone entity, the model must include at least one instance of the following objects: Trader, Importer, OfferSpace and LinkSpace. For analysis of the Export operation, an Exporter must also be included in the model.

7.5.1 Single Non-threaded Trader Query

A single non-threaded Trader is the simplest of scenarios in which the Trader has a single Query to service and does not pass the Query to other linked Traders. This scenario is shown in figure 7.3, where an Importer is represented by a dashed circle containing an **I**, and the Trader is represented by a circle containing a **T** and an instance identifier (**1** or **2**). When analysing a standalone Trader, it is possible that the Query is not propagated to linked Traders due to policy limitations rather than a lack of links to follow. This is shown in figure 7.3, where a dashed link from **T1** to **T2** exists but is not pursued by the Query (shown as a dashed arrow). Note that interaction with an Exporter is considered in section 7.5.3.

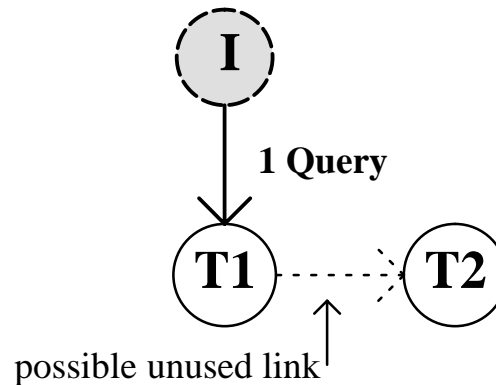


Fig. 7.3: Trader servicing a single Query

7.5.1.1 Initialisation

In order to create a standalone Trader, two approaches may be taken. The first involves isolating the Trader using policies whilst the other involves ensuring that the Trader has no links to follow, even if it attempts to do so.

When adopting the first approach, three policy settings may be altered. They are:

- the `link_follow_policy` of all of the Trader's Links,
- the Trader's Import Policy,
- the Import Policy of the Query itself.

The second option involves initialising the LinkSpace object so that it does not contain links for the Trader being analysed. This option allows the Trader to `get_links` from the LinkSpace object, but the request results in an empty list of links.

When analysing scenarios in which `merged_policy_options()=if_no_local`, some scenarios require the Importer object to be initialised with a Query that results in a successful match from the OfferSpace object, whilst other scenarios require it to be initialised with a Query that does not result in a match.

7.5.1.2 OG Analysis

OG analysis of the Trader can be used to verify that the Trader successfully performs a Query operation and always finishes the Query with exactly the same marking. This marking should indicate that:

- a response to the Query containing matching offers associated with all Traders visited by the Query was received by the Importer,
- all objects are in a *clean* state (no unexpected tokens left over in an object's CPN).

All possible permutations of the stopping conditions need to be considered when verifying the scenario in figure 7.3. This means that each of the possible return values for `merge_policy_options()` must be used with:

- `hop_count=0` and `hop_count<>0` and
- `links_available=0` and `links_available<>0`

to ensure that the model behaves correctly under all conditions. Table 3 shows the results of OG verification for each of the scenario permutations when the number of available links=0 (i.e. the LinkSpace object returns an empty list of links). Two unique behaviours were identified in the scenarios, resulting in two classes of OG (A and B).

Available links=0		merged follow_option			
		local_only	if_no_local and local_match=true	if_no_local and local_match=false	always
Query hop_count	= 0	A	A	A	A
	<> 0	A	B	B	B

Table 3 – Combinations of `hop_count` and `follow_option` for `links=0`

A: Nodes=20, Arcs=21 Full OG. Simulation = 20 steps

B: Nodes=75, Arcs=122 Full OG. Simulation = 26 steps

Table 3 shows that the model displays identical behaviour when the unified policy=`local_only` and when the `hop_count=0`. This behaviour is denoted as type A. For combinations which resulted in a type A OG, the Trader did not issue a request to the LinkSpace object to `get_links`, (since **Local Search Only** occurs on `Process_Req#14` as described in section 6.11.6.1) resulting in a small OG (20 nodes). This is expected since there is no need for the Trader to `get_links` in either case since the Query does not need to be propagated.

In contrast, combinations resulting in type B OGs did contain a reference to the LinkSpace object to `get_links` since **Link_andor_Local_Search** occurs on `Process_Req#14` (as described in section 6.11.6.1). This is expected since when the `merged_policy_options()` is `if_no_local` or `always`, the Trader needs to get links from the LinkSpace object, regardless of whether the links are subsequently used or not. This depends upon the results obtained from a local search of the OfferSpace and the link's follow policy.

It follows that combinations resulting in type A OGs are not affected by changing the number of links associated with a Trader because the links are never obtained. Thus, only combinations resulting in type B OGs need to be investigated further when the number of links available is non-zero, as shown in Table 4.

Some combinations with non-zero links result in the Query being propagated along the link (a link search) which is beyond the scope of analysis for a single Trader. These combinations are marked **link search** and are considered later in section 7.7.

Available links <> 0		merged		follow_option	
		local_only	if_no_local and local_match=true	if_no_local and local_match=false	always
Query hop_count	= 0	N/A	N/A	N/A	N/A
	<> 0	N/A	C	link search	link search

Table 4 – Combinations of hop_count and follow_option for links <> 0

C: Nodes=85, Arcs=140 Full OG. Simulation = 28 steps

When the `merged_policy_options()` is `if_no_local`, the `hop_count <> 0`, a local match to the Query is found and there are links to follow, the Trader exhibits a third behaviour denoted C. This behaviour is observed because the Trader obtains links from the LinkSpace object and performs a local offer space search which results in a matching offer, thereby signalling that the search is not to be propagated further.

The Trader processes `if_no_local` requests at **Select_If_No_Local** and **Send_If_No_Local** on (Link_Import#15, figure 6.25), an extra two transition occurrences when compared with type B behaviour. The Trader aborts searching linked Traders at **Send_If_No_Local** when it determines (from *Num_Local_Matches*) that the local search resulted in a successful match. Since behaviour C exhibits two more transition occurrences than behaviour B, it has a larger OG.

By using CPN/ML's `ListDeadMarkings()` function call, it is possible to verify for each OG that it contains a single terminal marking. This means that the model reaches a single conclusive state whose marking may be checked to ensure that it is the expected terminal state of the model. The marking of this final state can be obtained by instructing Design/CPN to draw the dead node using the menu and then double clicking on it.

The Trader is highly configurable which resulted in numerous scenarios requiring investigation. As a result of analysis, it has been shown that for all possible standalone scenarios,

the resulting OGs contain a single terminal marking which represents the successful conclusion of the Trading scenario and the absence of deadlocks.

At this point, the Trader’s behaviour has been verified for all possible permutations of its controlling parameters and this result can now be used as a building block (or axiom) for further verification.

7.5.2 Single Trader with Concurrent Request Servicing

There is a need to verify that multiple concurrently executing threads within the Trader do not interact in an undesired manner. Since there is a high degree of internal concurrency within the Trader itself, applying Occurrence Graphs with Equivalences reduces the size of the model’s OG by mapping out the τ_id field inside tokens. This field is used by the model to distinguish tokens belonging to different threads. The identifier associated with a thread depends upon the order in which it is processed. This is a non-deterministic property of the model and is mapped out by the Equivalence relation used in the OEOS tool [85].

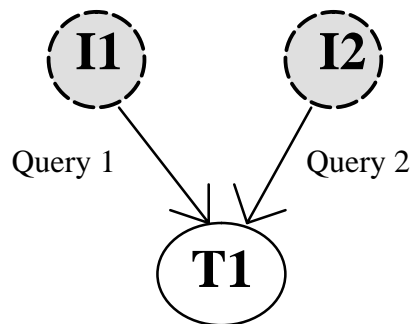


Fig. 7.4: The Trader services multiple Queries concurrently

In figure 7.4, the Trader is sent two Queries (Query 1 and Query2) by two Importers (**I1** and **I2**). These requests are generally for different services, although this is not critical for the verification of independent concurrent service processing. It is also possible that the two Queries originate from the same Importer (e.g. **I1**), rather than from two Importers.

In section 7.5.1, three fundamental standalone Trader behaviours were identified (A, B and C). When verifying that the Trader can service multiple Queries concurrently, it is important to test all possible combinations of Trader behaviours. Having identified the three fundamental Trading behaviours, only combinations of behaviour A,B and C must be investigated. The following combinations have been identified:

```

Query1 = A, Query2 = A,
Query1 = A, Query2 = B,
Query1 = A, Query2 = C,
Query1 = B, Query2 = B,
Query1 = C, Query2 = C

```

Thus, when a Trader services a type A Query, it displays behaviour of a type A OG. Similarly, servicing a type C Query will result in a type C OG. Table 5 shows the number of steps required for simulation runs using different combinations of concurrent Queries that are processed by a

standalone Trader. Due to symmetry, some of the scenarios are redundant since they are already included in the table.

In order to display type B behaviour, the Trader must not have any links available, whereas type C behaviour requires the Trader to have at least one link available in the LinkSpace object. Thus, it is not possible for a Trader to service a type B Query at the same time as a type C Query since they are mutually exclusive (Mut-Ex).

Concurrent Behaviours		Query 1		
		A	B	C
Query 2	A	40	Redundant	Redundant
	B	46	52	Mut-Ex
	C	48	Mut-Ex	56

Table 5 – Simulation Steps for Single Trader Concurrent Queries

The values in Table 5 are as expected, based upon the number of steps required to simulate single type A, B and C behaviours (calculated in section 7.5.1). For example,

concurrent A and A = 20+20 = 40 steps
 concurrent A and C = 20+28 = 48 steps
 concurrent B and B = 26+26 = 52 steps

Table 6 shows statistics for OGs created using the standard OG features of Design/CPN which do not utilise Equivalence classes.

Concurrent Behaviours		Query 1		
		A	B	C
Query 2	A	Nodes=680 Arcs=1284 Secs=11	Redundant	Redundant
	B	Nodes=2630 Arcs=6756 Secs=74	Nodes=10124 Arcs=32820 Secs=507	Mut-Ex
	C	Nodes=2992 Arcs=7740 Secs=89	Mut-Ex	Nodes=13104 Arcs=42956 Secs=653

Table 6 – OG size for Single Trader Concurrent Queries

Table 7 shows the results of creating OGs with Equivalence Classes for each possible combination of concurrent Query types.

The result for two concurrent A Queries is expected since for each in Query A there are 20 different possible states for Query A', resulting in 400 different states. The terminal marking of this example combines two of the states into one, resulting in 399 states using Equivalence Classes. OEOS analysis resulted in significantly smaller OGs whilst preserving detection of the single terminal marking in each case.

For each OG described in Tables 6 and 7, there is a single terminal marking which corresponds to successful termination of the concurrent Queries. These markings are included in Appendix F.

Concurrent Behaviours		Query 1		
		A	B	C
Query 2	A	Nodes=399 Arcs=773 Secs=5	Redundant	Redundant
	B	Nodes=1547 Arcs=3954 Secs=26	Nodes=4274 Arcs=13163 Secs=97	Mut-Ex
	C	Nodes=1753 Arcs=4516 Secs=29	Mut-Ex	Nodes=4096 Arcs=12323 Secs=91

Table 7 – OEOS size for Single Trader Concurrent Queries

7.5.3 Single Trader with Importer–Exporter interaction

This scenario is an extension of the Single threaded Trader Query verification outlined in section 7.5.2. After the Exporter performs a successful Export, the Importer queries the Trader for an Offer which matches that exported by the Exporter. When the Importer receives a response to its Query, it extracts the first Offer from the list of matching offers and invokes the service on the service provider (Exporter). This is the same sequence of interactions modelled in section 5.4, except the model of Chapter 6 is a much more detailed model of the Trader’s internal logic.

The Exporter is initialised such that *Offers_to_Export* contains an offer which will be exported to the Trader when **Export_Offer** occurs, as explained in section 6.10.4.

Referring to figure 6.13, section 6.10.3, it can be seen that before the Importer can invoke the service it obtained from the Trader using a Query operation, an ϵ token must be present in *Ready_to_order* to allow **Invoke_Method** to become enabled. When this is the case, an extra three transitions occur (The Importer’s **Invoke_Method**, the Exporter’s **Accept_Order** and the Importer’s **Service_Ack**) before the model reaches a terminal marking (node 376 shown in figure 7.5) which includes:

- an Export_id token in the Exporter’s *Offer_Identifiers*,
- a submit-order Message from the Importer in the Exporter’s *Pending_Orders*,
- an acknowledgment Message in the Importer’s *Service_Completed*,

The other terminal marking (node 366 shown in figure 7.6) represents the situation when the Importer does not obtain a matching offer because its Query was processed by the Trader before the Exporter’s Offer was added to the OfferSpace’s list of Offers. In this situation an empty list of matching Offers is located in the Importer’s *Matched_Offers* place. OGs obtained using standard OG tools and OGs with Equivalence Classes are shown in Table 8 and are included in Appendix F.

```

376
Process_Req'IWT_Request_Store 1:
1'[{hop_count = 0,link_follow_rule = always,request_id = {stem = importer,t_ident =
3},exact_match_type = TRUE}]

Export'Exported_Offers 1:
1'(off(((pizza,[ (MANDATORY,Pizza_Name,the_lot)]),exporter)),1)
Export'Export_Ids 1: 1'[2,3,4]

Exporter'Offer_Identifiers 1: 1'(exp_return(1),3)
Exporter'Pending_Order 1:
1'((importer,exporter),(submit_order,(serv_t((pizza,[ (MANDATORY,Pizza_Name,the_lot)
)),1)))

Importer'Service_Completed 1: 1'ack(OK)
Importer'Matched_Offers 1:
1'(off_l(((pizza,[ (MANDATORY,Pizza_Name,the_lot)]),exporter))),3)

Offer_Space'Offer_Storage 1:
1'(t1_osi,(((pizza,[ (MANDATORY,Pizza_Name,the_lot)]),exporter),
((pizza,[ (MANDATORY,Pizza_Name,vegetarian)]),pizza_chef),
((software,[ (MANDATORY,Software_Name,spreadsheet)]),advantek),
((used_car,[ (MANDATORY,Car_Type,wagon)]),bob_moran)))+
1'(t2_osi,(((pizza,[ (MANDATORY,Pizza_Name,the_lot)]),pedro_pizza),
((software,[ (MANDATORY,Software_Name,simcity)]),software_supermarket),
((used_car,[ (MANDATORY,Car_Type,wagon)]),aust_motors)))+
1'(t3_osi,(((pizza,[ (MANDATORY,Pizza_Name,vegetarian)]),pizza_haven)))

Trad_Env'Comms_Medium 1: empty

```

Fig. 7.5: Reduced marking of node 376 from OEOS analysis of Importer–Exporter interaction

OG	OEOS
Nodes=609	Nodes=376
Arcs=1132	Arcs=695
Seconds=19	Seconds=4
FULL	FULL

Table 8 – OGs for Concurrent Importer/Exporter Trader Interaction

Each of the OGs resulted in two terminal markings, as discussed earlier in this section. For the OEOS analysis, the terminal markings were 366 and 367. When the Importer invokes a service request on the Exporter, the simulation takes 36 steps. When the Importer does not obtain a matching offer, the simulation takes 33 steps.

7.5.4 Summary

In this section, the Trader’s most basic functionality has been verified, where a single Query and concurrent Queries are processed by a standalone Trader. All of the possible stopping criteria were used in the verification and it was found in section 7.5.1.2 that the Trader’s behaviour was identical when hop_count=0 and merge_policy_option()=local_only (type A OG).

```

366

Process_Req'IWT_Request_Store 1:
1'[{hop_count = 0,link_follow_rule = always,request_id = {stem = importer,t_ident =
3},exact_match_type = TRUE}]

Export'Exported_Offers 1:
1'(off(((pizza,[ (MANDATORY,Pizza_Name,the_lot)]),exporter)),1)
Export'Export_Ids 1: 1'[2,3,4]

Exporter'Offer_Identifiers 1: 1'(exp_return(1),3)
Exporter'Pending_Order 1: empty
Exporter'Offers_to_Export 1: empty

Importer'Service_Completed 1: empty
Importer'Matched_Offers 1: 1'(off_l([],3)

Offer_Space'Offer_Storage 1:
1'(t1_osi,[((pizza,[ (MANDATORY,Pizza_Name,the_lot)]),exporter),
((pizza,[ (MANDATORY,Pizza_Name,vegetarian)]),pizza_chef),
((software,[ (MANDATORY,Software_Name,spreadsheet)]),advantek),
((used_car,[ (MANDATORY,Car_Type,wagon)]),bob_moran)])+
1'(t2_osi,[((pizza,[ (MANDATORY,Pizza_Name,the_lot)]),pedro_pizza),
((software,[ (MANDATORY,Software_Name,simcity)]),software_supermarket),
((used_car,[ (MANDATORY,Car_Type,wagon)]),aust_motors)])+
1'(t3_osi,[((pizza,[ (MANDATORY,Pizza_Name,vegetarian)]),pizza_haven)])

Trad_Env'Comms_Medium 1: empty

```

Fig. 7.6: Reduced marking of node 366 from OEOS analysis of Importer–Exporter interaction

The three unique behaviour types may be summarised as:

- A:** The Trader does not attempt to get its links,
- B:** The Trader gets its links but finds that there are no links to pursue,
- C:** The Trader has a unified policy of `if_no_local`, obtains valid links from the `LinkSpace` object but does not send the `Query` to linked Traders because it also obtains a local `OfferSpace` match to the `Query`.

Having identified the three basic Trading behaviours, in section 7.5.2, a single Trader was configured such that it concurrently serviced all possible combinations of pairs of the three basic behaviours. This showed that the Trader is capable of servicing all types of `Queries` concurrently without exhibiting unexpected deadlock or other undesirable behaviour.

In section 7.5.3, the Trading Environment was configured such that an Importer would submit a `Query` to a single Trader requesting an offer which was not in the `OfferSpace` whilst at the same time, the Exporter was exporting a matching offer via the Trader. This resulted in two terminal markings as expected and is another example of concurrent Trader operation combined with analysis of the Trader’s export functionality.

Results of the analysis presented in this section are used during further analysis into the behaviour of multiple independent Traders in the following sections.

7.6 Verification of Multiple Independent Traders

In section 7.5, the basic standalone Trader functionality was verified for all permutations of parameters (policies, links and offers), thereby testing all standalone Trader stopping conditions for single and concurrent Query processing. In this section, the previous result is used to verify the correct operation of multiple Traders operating independently but concurrently in the same system.

7.6.1 Multiple Autonomous Objects

It is important to verify that the model can contain multiple autonomous entities which do not “interfere” with each other. An abstract model of concurrent independent Traders is shown in figure 7.7 which shows two Traders (**T1** and **T2**) servicing 2 Queries independently.

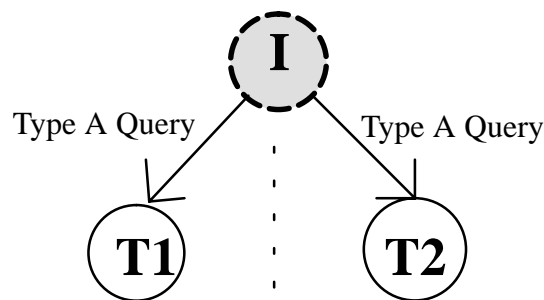


Fig. 7.7: 2 Traders and 2 independent Queries

7.6.1.1 Initialisation

The Trad_Env#2 page was modified to contain two Traders. This was accomplished by creating a new substitution transition connected to *Comms_Medium* and connecting it to the Trad_Int#3 page. Each of the interfaces was initialised as for other scenarios, with the initial marking of each changed to the appropriate unique identifier for each instance.

The LinkSpace object was initialised so that both Trader 1 (**T1**) and Trader 2 (**T2**) were not linked to other Traders. The Importer was initialised to contain two Queries, one addressed to T1 and the other to T2. The OfferSpace object was initialised to contain matching offers for both of the Queries to be matched by the Traders.

7.6.1.2 OG Analysis

Since there is no internal thread concurrency within the Traders, there is no benefit to be obtained by using Occurrence Graphs with Equivalences (OEOS). Using the OGA, it was found that there were no deadlocks since a single dead node was observed in the OG. The marking of *Matching_Offers* in the single terminal marking corresponds to the successful conclusion of concurrent Queries. The single terminal marking in the OG indicates that for all possible behaviours, the model always terminates in the same desirable state.

Concurrent Behaviours		Trader 1 Query		
		A	B	C
Trader 2 Query	A	Nodes=437 Arcs=828 Secs=2	Redundant	Redundant
	B	Nodes=1547 Arcs=3954 Secs=37	Nodes=5429 Arcs=17376 Secs=142	Mut-Ex
	C	Nodes=1753 Arcs=4516 Secs=43	Mut-Ex	Nodes=6969 Arcs=22584 Secs=161

Table 9 – OG size for Multiple Traders

In section 7.5.1, it was shown that when issued as the sole Query to a standalone Trader, OGs for servicing of Type A, B and C Queries all contained a single terminal marking. In this section, it has been shown that combinations of Query types can be serviced concurrently by two independent Traders.

Thus, it has been shown that the model is capable of allowing multiple instances of Traders to operate independently and concurrently. This result may appear intuitive, but it ensures that the CPN model of the Trader does not allow any hidden interactions between Traders when there are multiple Trader instances operating concurrently.

7.6.2 Multiple Autonomous Threaded Objects

The final step in the basic verification of the Trader is to verify that the model can accommodate multiple Trader instances that are autonomous, operate concurrently and independently and can service multiple requests concurrently. This will be accomplished using the results from previous sections.

In section 7.5.2, it was verified that the Trader model was capable of successfully servicing multiple Queries concurrently. Section 7.6.1 showed that the model is capable of containing multiple independent Traders. Using these two results, it can be deduced that the model allows multiple independent Traders to operate concurrently (from 7.6.1), where each Trader is able to service multiple Queries concurrently. This is shown in figure 7.8.

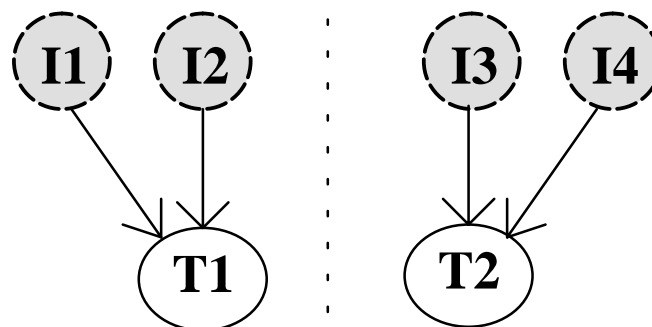


Fig. 7.8: 2 Traders and 4 independent Queries

This result is significant because it allows complicated Trading scenarios to be broken down into more manageable independent concurrent activities which can be verified individually.

Since it has been shown that multiple concurrent Traders do not interact in unexpected ways, it is possible to break down complicated scenarios into a number of simple scenarios which operate concurrently but independently. This result will be used when verifying arbitrarily complex interworking Trader scenarios in section 7.9.

7.7 Verification of Interworking Traders

To verify the Interworking of Traders, it is necessary to create a series of scenarios in which multiple traders interact. These scenarios will verify a set of Query stopping conditions which can be used to verify that forwarded Queries always terminate. The analysis also ensures that the model does not contain livelocks or deadlocks.

It is possible to envisage extremely complex trading topologies, containing many Traders that interwork via numerous links and which are able to issue multiple Queries with arbitrary initial `hop_count` values. To calculate the entire state space of such a system would be infeasible and thus, a different approach must be adopted.

The analysis in this section builds upon the results of the previous sections and aims to verify the interworking of m Traders, each of which is capable of servicing n requests concurrently, where n and m are positive integers.

7.7.1 Recursive Definition of Interworking Traders

When a Query is propagated to linked Traders, the forwarded Query is a slightly modified version of the initial Query since it has a smaller `hop_count` and possibly, a different `follow_policy`. A propagated Query is serviced by a different Trader using the same type of interface, but the Query has slightly different parameters to those of the initial Query. This resembles a recursive situation in which a function calls itself many times with different parameters until it reaches a point at which it no longer calls itself. At that point, it retraces its path through the recursive function calls until it reaches the initial call and returns its result.

According to Garland [141], a recursive problem must meet two requirements if it is to solve a problem successfully. They are:

- any invocation of a recursive sub-program from within its own definition must solve a problem simpler than the one used when the program was first invoked,
- there must be some instances in which the recursive sub-program does not invoke itself recursively (stopping conditions).

A recursive approach to algorithms is common-place in functional programming languages such as SML, because it allows a small function to re-call itself until the problem has been solved. Many of the functions used in the CPN model described in Chapter 6 use recursion. It is considered that Traders do the same thing when they interwork, since a Query is continually broken up into smaller, more easily processed elements which are forwarded to linked Traders.

All such Queries should eventually reach a point at which it is no longer propagated and the Query trail is re-traced back to the Trader which received the initial Query.

The analysis of the Trader in this section exploits the recursive behaviour displayed by the Trader and combines it with a number of stopping conditions to verify that forwarded Queries always terminate. In addition, it will be shown that when an offer is propagated between Traders, its path through the Trading graph is deterministic.

7.7.2 Reducing the size of the Problem

The Trader reduces the size of the problem (forwarding of Queries to linked Traders) in two ways. Firstly, it reduces by 1 the `hop_count` in a Query's Import Policy when it is forwarded to linked Traders. It is also possible for the `follow_policy` of a Query to be reduced in strength as it is propagated, either by the Trader itself, or the link which the Query is following (i.e. from `if_no_local` to `local_only`). Using either of these two parameters, it is possible to limit the number of links the Query follows, thereby reducing the size of the problem.

7.7.3 Interworking Query Stopping Conditions

Based upon modelling the Trader presented in Chapter 6, the following six Query stopping conditions have been identified. Each of the conditions have been modelled in an associated scenario as shown in Table 10. Note that *valid* links are defined in section 6.5.1.1.

Stopping Condition —— Scenario	<code>hop_count = 0</code>	total available links = 0	merged policy = <code>local_only</code>	merged policy = <code>if_no_local</code> but gets a local match	total available links > 0 but valid links = 0	Query re-visits a Trader with a smaller <code>hop_count</code>
1	Y	N	N	N	N	N
2	N	Y	N	N	N	N
3	N	N	Y	N	N	N
4	N	N	N	Y	N	N
5	N	N	N	N	Y	N
6	N	N	N	N	N	Y
Combined	Y	Y	Y	Y	Y	Y

Table 10 – Scenario Stopping conditions

The scenarios are used to verify that a Query will stop being propagated when a specific stopping condition is met. The scenarios have been designed to ensure that only one of the stopping conditions is tested in each scenario except for the combined scenario which includes **all** of the stopping conditions (although some stopping conditions are tested more than once). Analysis of these stopping conditions will be used later in section 7.9 to analyse a more complex Trading example.

7.7.3.1 hop_count=0 (Scenario 1)

This scenario verifies that a Query stops following links and returns a result (possibly an empty list of Offers) when its `hop_count=0`. This is a very simple stopping condition which is shown in figure 7.9. In the CPN model of the Trader, this stopping condition is considered on the `Process_Request#14` page (figure 6.24).

When a non-duplicate Query is processed with `hop_count=0` (by **T3**), the **Local_Search_Only** transition occurs since the search is not intended to extend to linked Traders. This transition is also enabled if `merge_policy_options()` evaluates to `local_only`, which is the case for the terminating scenario described in section 7.7.3.3.

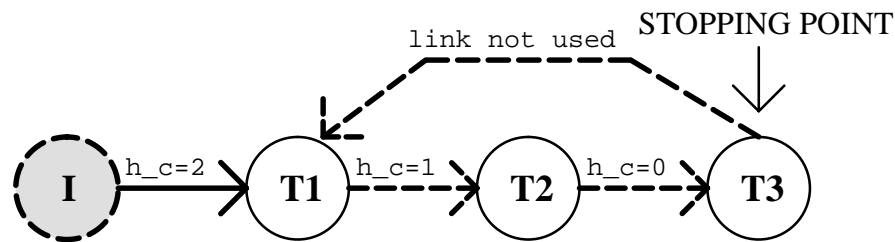


Fig. 7.9: Query stops when `hop_count=0`

When **Local_Search_Only** occurs, three tokens are created. One of the tokens is a copy of the initial Query which is placed in `Mess_Store_2` for later use. Since the search has been truncated to being a `local_search_only`, a token is placed in `Number_Of_Links` with the value `(0, t_id)`, thereby indicating that there are 0 links for the current `t_id`. Finally, a message is created which is addressed to the Offer Space Interface (OSI) which requests it to `get_offers` from the OfferSpace object.

If the request is a duplicate, then **No_Link_Search_Reqd** becomes enabled since a local search of the offer space is not necessary, and Queries are not propagated when `hop_count=0`. Thus, a message containing an empty list of matching offers and addressed to the object which sent the Query is created and put into **End_Of_Transaction**.

For a Query with `hop_count=0`, the **Link_andor_Local_Search** transition cannot become enabled in conflict with **Local_Search_Only** since its guard requires the `hop_count` of the current request's Import Policy to be greater than 0 (i.e. `#hop_count(ip)>0`).

The Query is sent by the Importer (**I**) with `hop_count=2` and is processed by Trader 1 (**T1**) which also forwards the Query to Trader 2 (**T2**) with `hop_count=1`. It processes the Query and forwards it to Trader 3 (**T3**) with `hop_count=0`. **T3** cannot forward the Query and thus, processes it locally and then returns its results to **T2** and so on until the combined result of all the searching is returned to **I**.

The scenario verifies that even with a merged policy option of `always` and `valid` links to follow such as the link from **T3** to **T1**, (see sections 6.5.1.1 and 7.7.3.5 for a definition of `valid`), the Query terminates after being serviced by **T3**. Table 11 contains statistics related to the OG of this scenario which is shown in figure 7.10.

OG
Nodes=2628
Arcs=6807
Seconds=25
FULL

Table 11 – OG size for Stopping Scenario 1

Figure 7.10 contains markings which indicate the following:

- the OfferSpace object's database (*Offer_Storage*) and therefore, all of the offers associated with the three Traders,
- the LinkSpace object's database (*Link_Storage*) and therefore, all of the links associated with the three Traders,
- each Trader's interworking Query storage (*IWT_Request_Store*) which indicates how many Queries each Trader has processed,
- the matched offers which were returned to the Importer in *Matched_Offers* and
- any Messages still in transit on the *Comms_Medium*.

Since there are three Traders instantiated in this example, the place markings for each Trader instance are indicated by a 1, 2 or 3 next to the `page_name`' `place_name` text on the left.

```

2628
Offer_Space'Offer_Storage 1:
1'(t1_osi,[(pizza,[MANDATORY,Pizza_Name,vegetarian]),pizza_chef),
((software,[MANDATORY,Software_Name,spreadsheet]),advantek),
((used_car,[MANDATORY,Car_Type,wagon]),bob_moran)])+
1'(t2_osi,[(pizza,[MANDATORY,Pizza_Name,the_lot]),pedro_pizza),
((software,[MANDATORY,Software_Name,simcity]),software_supermarket),
((used_car,[MANDATORY,Car_Type,wagon]),aust_motors)])+
1'(t3_osi,[(pizza,[MANDATORY,Pizza_Name,vegetarian]),pizza_haven]])

Link_Space'Link_Storage 1:
1'(t1_lsi,[{n = L1, lui = lui_2, regi = regi_2, def_fb = always, limit_fb = always}])+
1'(t2_lsi,[{n = L2, lui = lui_3, regi = regi_3, def_fb = always, limit_fb = always}])+
1'(t3_lsi,[{n = L3, lui = lui_1, regi = regi_1, def_fb = always, limit_fb = always}])

Process_Req'IWT_Request_Store 1: 1'[{hop_count = 2, link_follow_rule =
always, request_id = {stem = importer, t_ident = 3}, exact_match_type = TRUE}]
Process_Req'IWT_Request_Store 2: 1'[{hop_count = 1, link_follow_rule =
always, request_id = {stem = importer, t_ident = 3}, exact_match_type = TRUE}]
Process_Req'IWT_Request_Store 3: 1'[{hop_count = 0, link_follow_rule =
always, request_id = {stem = importer, t_ident = 3}, exact_match_type = TRUE}]

Importer'Matched_Offers 1:
1'(off_l1([(pizza,[MANDATORY,Pizza_Name,vegetarian]),pizza_chef),
((pizza,[MANDATORY,Pizza_Name,vegetarian]),pizza_haven)]),3)

Trad_Env'Comms_Medium 1: empty

```

Fig. 7.10: Reduced marking of node 2628 from OG analysis of scenario 1

It can be seen that the Query is propagated to **T3** but is not propagated onwards to **T1** since **T1** serviced the Query when its `hop_count=2`. It is also possible to see that matching vegetarian pizza offers are obtained from **T1** (`pizza_chef`) and **T3** (`pizza_haven`).

7.7.3.2 Total available links=0 (Scenario 2)

In this scenario the total number of links available to the Trader for forwarding of Queries is zero, as shown in the scenario model of figure 7.11. The two Traders are configured such that the `limiting_follow_rule=always` in all cases, and the Query `hop_count` starts with a non-zero value (arbitrarily chosen to be 2). However, the `LinkSpace` object is initialised so that it returns an empty list of links for **T2** to follow. Thus, even though the Query `hop_count` allows more hops to be made, the Query terminates after being processed by both of the Traders. This stopping scenario is handled on the `Import_from_Links#15` page (discussed in section 6.11.6.2 and shown in figure 6.25) and is independent of the Query's non-zero `hop_count` value since Query `hop_count` is processed on `Process_Req#14` (discussed in section 6.11.6.1 and shown in figure 6.24).

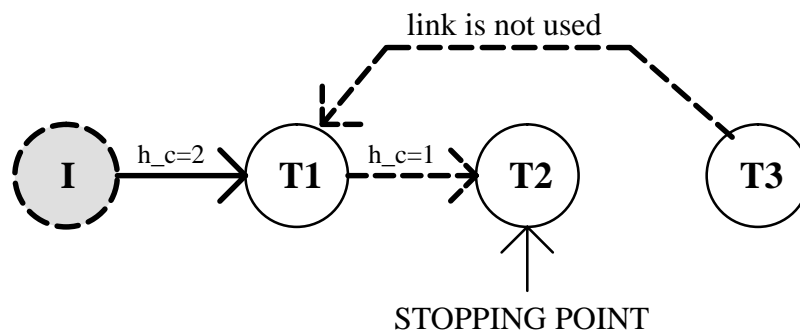


Fig. 7.11: Query stops when there are no links to follow

A check on the OG (statistics shown in Table 12) using `ListDeadMarkings()` resulted in a single node (758) being identified as having no successors. The marking of this node (shown in figure 7.12) corresponds with a matching vegetarian pizza offer is obtained by **T1** and returned to the Importer (**I**). It also shows that the Query stopped at **T2** since **T2**'s `IWT_Request_Store` marking shows that it processed a Query with `hop_count=1` whilst **T3**'s shows that it did not service any Queries at all.

OG
Nodes=758
Arcs=1789
Seconds=6
FULL

Table 12 – OG size for Stopping Scenario 2

```

758
Offer_Space'Offer_Storage 1:
1'(t1_osi,[(pizza,[MANDATORY,Pizza_Name,vegetarian]),pizza_chef),
((software,[MANDATORY,Software_Name,spreadsheet]),advantek),
((used_car,[MANDATORY,Car_Type,wagon]),bob_moran)])+
1'(t2_osi,[(pizza,[MANDATORY,Pizza_Name,the_lot]),pedro_pizza),
((software,[MANDATORY,Software_Name,simcity]),software_supermarket),
((used_car,[MANDATORY,Car_Type,wagon]),aust_motors)])+
1'(t3_osi,[(pizza,[MANDATORY,Pizza_Name,vegetarian]),pizza_haven])

Link_Space'Link_Storage 1:
1'(t1_lsi,[{n = L1,lui = lui_2,regi = regi_2,def_fb = always,limit_fb = always}])+
1'(t2_lsi,[])+
1'(t3_lsi,[{n = L3,lui = lui_1,regi = regi_1,def_fb = always,limit_fb = always}])

Process_Req'IWT_Request_Store 1: 1'[{hop_count = 2,link_follow_rule =
always,request_id = {stem = importer,t_ident = 3},exact_match_type = TRUE}]
Process_Req'IWT_Request_Store 2: 1'[{hop_count = 1,link_follow_rule =
always,request_id = {stem = importer,t_ident = 3},exact_match_type = TRUE}]
Process_Req'IWT_Request_Store 3: 1'[]

Importer'Matched_Offers 1:
1'(off_l([(pizza,[MANDATORY,Pizza_Name,vegetarian]),pizza_chef]),3)

Trad_Env'Comms_Medium 1: empty

```

Fig. 7.12: Reduced marking of node 758 from OG analysis of scenario 2

This scenario shows that as expected, a Query always stops (i.e. at **T2**) when it runs out of links to follow, even when it has a non-zero hop_count and a unified follow_policy=always.

7.7.3.3 Unified Policy follow behaviour=local_only (Scenario 3)

In this scenario, the Query stops after being processed by Trader 1 (**T1**) even though there are valid links to be followed and the Query's hop_count is non-zero. The Query is processed in a manner that is similar to that presented in section 7.7.3.1, where the **Local_Search_Only** transition in Process_Request#14 occurs during processing of the Query. The model performs a local search of the OfferSpace object since the request is unique. A link search is never performed since merge_policy_options() evaluates to local_only.

A scenario that verifies correct termination of Queries when the merge_policy_options()=local_only is shown in figure 7.13. There are three Traders in the scenario and three links connecting them (**T1** to **T2**, **T2** to **T3** and **T3** to **T1**). The Importer's Query has an initial hop_count=4 (h_c=2) and **T1** has one link that can be followed. In order to have a unified_policy=local_only, one of the three policies (trader, import or link) must be equal to local_only. In this scenario, an arbitrary decision was made to make **T2**'s max_follow_policy=local_only, which could reflect an administrative decision to stop **T2** from following links.

An automatic simulation run required 51 steps until termination of the scenario, resulting in successful conclusion of the Query operation.

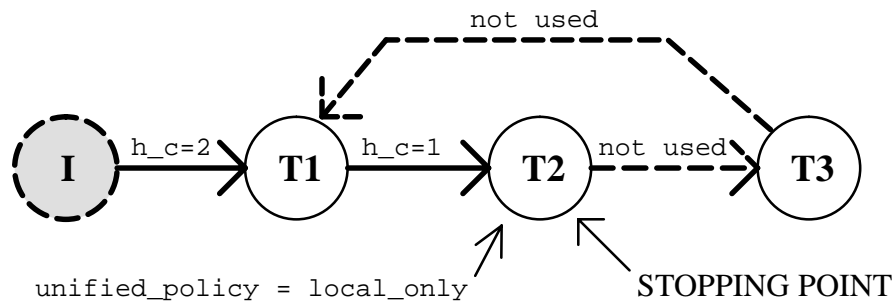


Fig. 7.13: Query stops when unified_policy=local_only

OG
Nodes=296
Arcs=535
Seconds=2
FULL

Table 13 – OG size for Stopping Scenario 3

As with previous scenarios, the OG graph contained a single terminal marking (shown in figure 7.15) which indicates a matching pizza the_lot offer from **T2** was returned to **I**. It also shows that the Query was not forwarded by **T2** to **T3** even though the hop_count=1, and a legal link to **T3** could have been followed. Instead, due to **T2**'s max_follow_policy value (local_only), the merged_policy_option() equalled local_only, thereby terminating propagation of the Query to linked Traders.

7.7.3.4 Merged Policy=if_no_local and local OfferSpace search is successful (Scenario 4)

The scenario shown in Figure 7.14 aims to verify that a Query stops being propagated at **T2** when the Trader has a merged policy of if_no_local and a local match is obtained, even though there are legal links to pursue (from **T2** to **T3**) and the Query's hop_count is non-zero. This scenario is similar to the scenario described in section 7.7.3.3 except that **T2**'s max_follow_policy=if_no_local and thus, merge_policy_options() returns if_no_local rather than local_only.

An automatic simulation run of this scenario took 59 steps with the hop_count of the Query initialised to 2. OG analysis of the scenario resulted in Table 14.

As with the other scenarios investigated in this section, a single terminal marking was obtained (shown in figure 7.16) which represents successful termination of the Query. It shows that **T2**'s max_follow_policy=if_no_local, the Query never reaches **T3** and that a successful match to the Query for vegetarian pizza is obtained by **T1**.

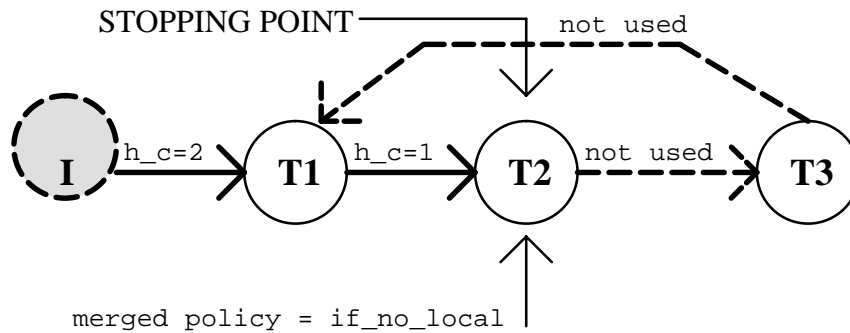


Fig. 7.14: Query stops with a local match and merged follow_policy=if_no_local

```

296
Offer_Space'Offer_Storage 1:
1'(t1_osi,[(pizza,[MANDATORY,Pizza_Name,vegetarian]),pizza_chef),
((software,[MANDATORY,Software_Name,spreadsheet]),advantek),
((used_car,[MANDATORY,Car_Type,wagon]),bob_moran)])+
1'(t2_osi,[(pizza,[MANDATORY,Pizza_Name,the_lot]),pedro_pizza),
((software,[MANDATORY,Software_Name,simcity]),software_supermarket),
((used_car,[MANDATORY,Car_Type,wagon]),aust_motors)])+
1'(t3_osi,[(pizza,[MANDATORY,Pizza_Name,vegetarian]),pizza_haven])

Link_Space'Link_Storage 1:
1'(t1_lsi,[{n = L1,lui = lui_2,regi = regi_2,def_fb = always,limit_fb = always}])+
1'(t2_lsi,[{n = L2,lui = lui_3,regi = regi_3,def_fb = always,limit_fb = always}])+
1'(t3_lsi,[{n = L3,lui = lui_1,regi = regi_1,def_fb = always,limit_fb = always}])

Funct_Opns'Trader_Imp_Attrib 1:
1'{def_hop_count = 2,max_hop_count = 3, def_follow_policy = local_only,
max_follow_policy = always, req_id_stem = lui_1}
Funct_Opns'Trader_Imp_Attrib 2:
1'{def_hop_count = 2,max_hop_count = 3, def_follow_policy = local_only,
max_follow_policy = local_only, req_id_stem = lui_1}
Funct_Opns'Trader_Imp_Attrib 3:
1'{def_hop_count = 2,max_hop_count = 3, def_follow_policy = local_only,
max_follow_policy = always, req_id_stem = lui_1}

Process_Req'IWT_Request_Store 1: 1'[{hop_count = 2,link_follow_rule =
always,request_id = {stem = importer,t_ident = 3},exact_match_type = TRUE}]
Process_Req'IWT_Request_Store 2: 1'[{hop_count = 1,link_follow_rule =
always,request_id = {stem = importer,t_ident = 3},exact_match_type = TRUE}]
Process_Req'IWT_Request_Store 3: 1'[]

Importer'Matched_Offers 1:
1'(off_l([(pizza,[MANDATORY,Pizza_Name,the_lot]),pedro_pizza]),3)

Trad_Env'Comms_Medium 1: empty

```

Fig. 7.15: Reduced marking of node 296 from OG analysis of scenario 3

OG
Nodes=844
Arcs=2015
Seconds=7
FULL

Table 14 – OG size for Stopping Scenario 4

```

Funct_Opns'Trader_Imp_Attrib 1: 1`{def_hop_count = 2,max_hop_count =
3,def_follow_policy = local_only,max_follow_policy = always, req_id_stem = lui_1}
Funct_Opns'Trader_Imp_Attrib 2: 1`{def_hop_count = 2,max_hop_count = 3,
def_follow_policy = local_only, max_follow_policy = if_no_local, req_id_stem =
lui_1}
Funct_Opns'Trader_Imp_Attrib 3: 1`{def_hop_count = 2,max_hop_count = 3,
def_follow_policy = local_only, max_follow_policy = always, req_id_stem = lui_1}

Offer_Space'Offer_Storage 1:
1`(t1_osi,[((pizza,[MANDATORY,Pizza_Name,vegetarian]),pizza_chef),
((software,[MANDATORY,Software_Name,spreadsheet]),advantek),
((used_car,[MANDATORY,Car_Type,wagon]),bob_moran)))+
1`(t2_osi,[((pizza,[MANDATORY,Pizza_Name,the_lot]),pedro_pizza),
((software,[MANDATORY,Software_Name,simcity]),software_supermarket),
((used_car,[MANDATORY,Car_Type,wagon]),aust_motors)))+
1`(t3_osi,[((pizza,[MANDATORY,Pizza_Name,vegetarian]),pizza_haven)])

Process_Req'IWT_Request_Store 1: 1`[{hop_count = 2,link_follow_rule =
always,request_id = {stem = importer,t_ident = 3}, exact_match_type = TRUE}]
Process_Req'IWT_Request_Store 2: 1`[{hop_count = 1,link_follow_rule =
always,request_id = {stem = importer,t_ident = 3}, exact_match_type = TRUE}]
Process_Req'IWT_Request_Store 3: 1`[]

Link_Space'Link_Storage 1:
1`(t1_lsi,[{n = L1,lui = lui_2,regi = regi_2,def_fb = always,limit_fb = always}])+
1`(t2_lsi,[{n = L2,lui = lui_3,regi = regi_3,def_fb = always,limit_fb = always}])+
1`(t3_lsi,[{n = L3,lui = lui_1,regi = regi_1,def_fb = always,limit_fb = always}])

Importer'Matched_Offers 1:
1`(off_l([((pizza,[MANDATORY,Pizza_Name,the_lot]),pedro_pizza)),3)

Trad_Env'Comms_Medium 1: empty

```

Fig. 7.16: Reduced marking of node 844 from OG analysis of scenario 4

7.7.3.5 Valid links to follow=0 (Scenario 5)

This scenario (shown in figure 7.17) illustrates one of the modifications to the Trading standard's Interworking protocol [16] outlined in section 6.5.1. It is a specialised case of when there are no links to follow since the only link that exists is not considered *valid* according to the modified interworking protocol. This is because the incoming Query operation originated from **T1** and there is no benefit for **T2** to re-send the Query back to **T1**. This is true for all bi-directional links between Traders.

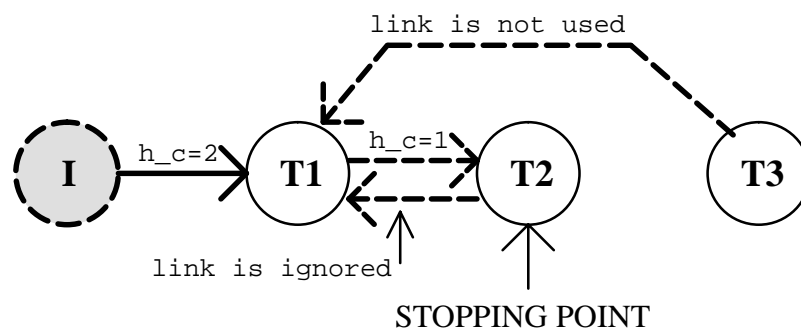


Fig. 7.17: Query stops when there are no valid links to follow

In the scenario, the Query for vegetarian pizza has a hop_count=2 and according to the unmodified Trading standard [16], the Query should be propagated by **T2** back to **T1**. This is

a redundant forwarding since the Query has already been processed by **T1** with a greater value of the `hop_count` parameter than the Query which would be re-sent to it. There are no situations in which a greater Trading scope will be achieved, assuming a static Trading environment (links do not change often over time). An automatic simulation of this scenario required 57 steps.

This protocol optimisation results in a significant reduction in the wasted processing by Traders and messaging bandwidth when interworking, since unproductive Trader queries and messaging is reduced. Statistics for the scenario's OG are given in Table 15 which also contains a single terminal marking at node 758.

OG
Nodes=758
Arcs=1789
Seconds=6
FULL

Table 15 – OG size for Stopping Scenario 5

The single terminal marking for this scenario is shown in figure 7.19. It provides markings which indicate that:

- the Query for vegetarian pizza did not reach **T3**,
- there were no outstanding messages in the *Comms_Medium*,
- **T2** was unable to obtain a match to the Query
- a matching offer was obtained by **T1** and returned to **I**.

7.7.3.6 Revisit Trader with smaller hop_count (Scenario 6)

In the scenario shown in Figure 7.18, a Query request is initiated by the Importer with a `hop_count`=4, and all policies are such that the Query is propagated successfully to all links. Since **T3** is connected via a link to **T1** when the Query's `hop_count`=1, the Query could possibly continue to be re-processed by **T1**, and subsequently passed on to **T2** at which point the `hop_count` would reach 0.

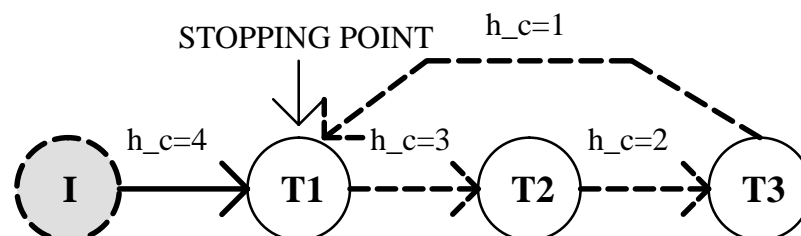


Fig. 7.18: Query stops when it re-visits a Trader

Both of these propagation actions are redundant since the Query operation has already been processed by **T1** and **T2**. Therefore, propagation of the Query is terminated at **T1** because it is a

duplicate Query operation whose hop_count is less than the previously processed Query operation.

```

758
Offer_Space'Offer_Storage 1:
1'(t1_osi,[(pizza,[MANDATORY,Pizza_Name,vegetarian]),pizza_chef),
((software,[MANDATORY,Software_Name,spreadsheet]),advantek),
((used_car,[MANDATORY,Car_Type,wagon]),bob_moran))+
1'(t2_osi,[(pizza,[MANDATORY,Pizza_Name,the_lot]),pedro_pizza),
((software,[MANDATORY,Software_Name,simcity]),software_supermarket),
((used_car,[MANDATORY,Car_Type,wagon]),aust_motors))+
1'(t3_osi,[(pizza,[MANDATORY,Pizza_Name,vegetarian]),pizza_haven])

Link_Space'Link_Storage 1:
1'(t1_lsi,[{n = L1,lui = lui_2,regi = regi_2,def_fb = always,limit_fb = always}])+
1'(t2_lsi,[{n = L2,lui = lui_1,regi = regi_1,def_fb = always,limit_fb = always}])+
1'(t3_lsi,[{n = L3,lui = lui_1,regi = regi_1,def_fb = always,limit_fb = always}])

Process_Req'IWT_Request_Store 1: 1'[{hop_count = 2,link_follow_rule =
always,request_id = {stem = importer,t_ident = 3},exact_match_type = TRUE}]
Process_Req'IWT_Request_Store 2: 1'[{hop_count = 1,link_follow_rule =
always,request_id = {stem = importer,t_ident = 3},exact_match_type = TRUE}]
Process_Req'IWT_Request_Store 3: 1'[]

Importer'Matched_Offers 1:
1'(off_l([(pizza,[MANDATORY,Pizza_Name,vegetarian]),pizza_chef]),3)

Trad_Env'Comms_Medium 1: empty

```

Fig. 7.19: Reduced marking of node 758 from OG analysis of scenario 5

An OG of this scenario was calculated, the results of which are shown in Table 16. This OG contained a single terminal marking, corresponding to the successful conclusion of the Query operation.

OG
Nodes=16005
Arcs=54337
Seconds=182
FULL

Table 16 – OG size for Stopping Scenario 6

7.8 Deterministic Traversal of the Offer Space

This scenario is shown in figure 7.20 where **T1** receives the same Query from two sources and where each Query has a different value for hop_count. The order in which the Queries are processed is not deterministic and thus, either of the two Queries may be serviced first. As discussed in section 6.5.3, without also maintaining the Query's hop_count parameter in the history of *recent* interworking Queries, it is not possible for the Query from **T4** to deterministically reach **T2**. This policy results in two terminal markings which correspond to:

- when the **T3** Query is serviced first and **T4** Query is rejected as being a duplicate, thereby returning an empty list of matching Offers, and
- when the **T4** Query is serviced first and manages to propagate to **T2** where it obtains a matching Offer.

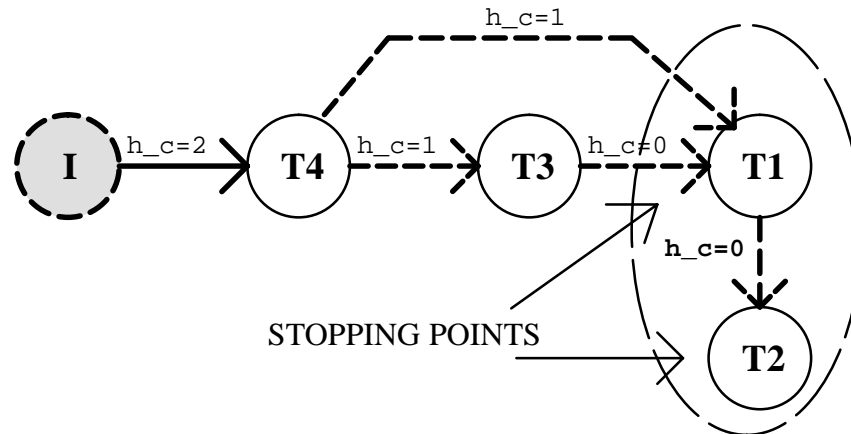


Fig. 7.20: T1 gets duplicates with different hop_count values

Figure 7.21 simulates the scenario of figure 7.20 and focuses on **T1** and **T2**'s behaviour, as indicated by the dashed ellipse in figure 7.20. Using initial markings, it is possible to simulate the effect of the Query having been serviced and propagated by **T4** and **T3**. This reduces the size of the OG since it is not necessary to include the Query servicing of **T4** and **T3** in the analysis.

A simulation of the scenario in figure 7.21 required 67 steps. An OG for the modified protocol which maintains the Query hop_count was calculated and its statistics are shown in Table 17. The single terminal marking showed an empty list of Offers being returned to **T3** and a list containing a single matching Offer being returned to **T4**. This was expected since the OfferSpace object was initialised so that **T2** would obtain a matching offer to the Query whereas **T1** would not.

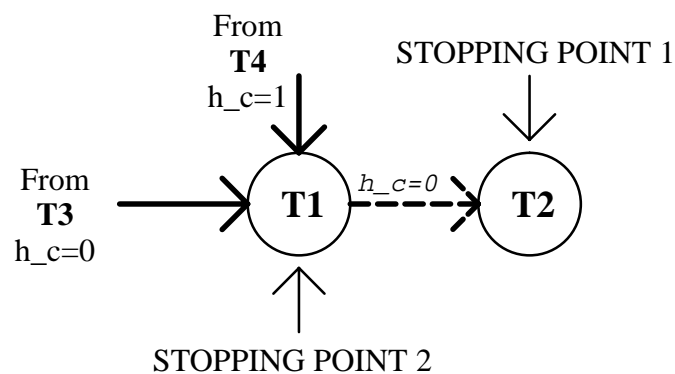


Fig. 7.21: Non-deterministic Query propagation

OG
Nodes=9636
Arcs=26258
Seconds=514
FULL

Table 17 – OG size for Deterministic Traversal

7.9 Combined Trading Topology (All Stopping Conditions)

In this scenario, all of the stopping conditions outlined in section 7.7.3 are demonstrated using a single scenario. The size of this scenario’s OG would be prohibitively large due to the high degree of inter-object concurrency and large number of active Traders. In Figure 7.22, the termination of a Query as a result of each of the six stopping conditions is indicated by an arrow and a corresponding number.

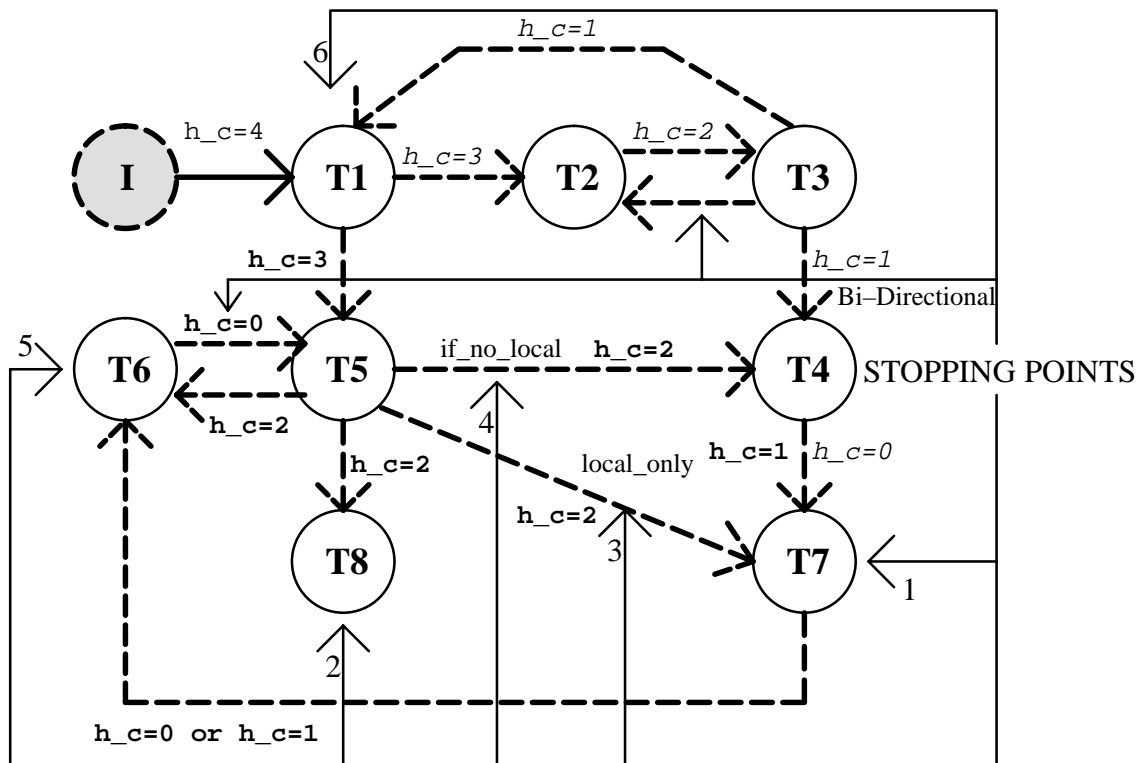


Fig. 7.22: Combined Scenario

When a Query from the Importer is processed by T1, it is forwarded to both T2 and T5. Each of these Queries continue to propagate until all (sub)propagated Queries enter one of the stopping conditions indicated in section 7.7.3. The path that a Query takes through the linked Traders will be denoted a *trace* and each of these conditions (which are represented at least once in the Combined Scenario) are discussed below.

7.9.1 Stopping condition 1

The trace shown in figure 7.23 shows a Query that propagates from the Importer **I** through **T1**, **T2**, **T3**, **T4** and then stops at **T7** because its `hop_count=0`. This is comparable to the stopping condition explained in section 7.7.3.1 except that in this example, the Query visits 5 Traders rather than 3. The propagation of Queries is linear and it has been shown that Traders can successfully propagate Queries for 3 links. Consider a partitioning of the Query in figure 7.23 into two elements: the external environment and scenario 1.

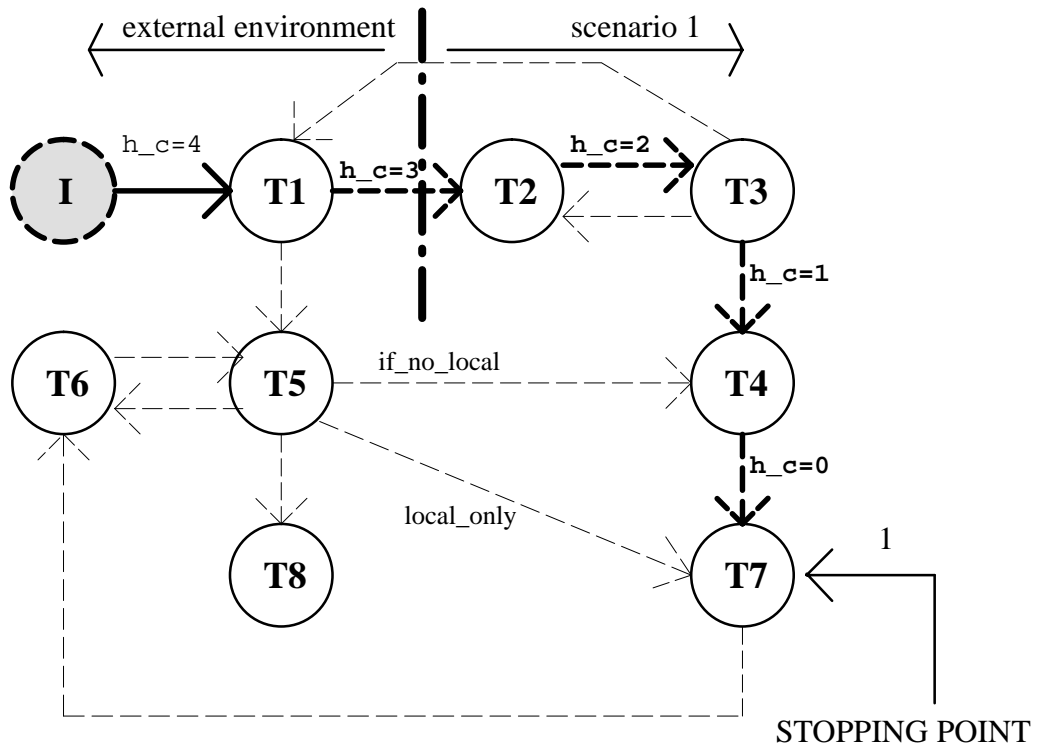


Fig. 7.23: Query trace 1

The external environment must:

- decrement `hop_counts` by 1 and
- accept results from linked Traders and return them to the entity which invoked the Query.

This behaviour has already been demonstrated in the scenario of section 7.7.3.1 between **T2**, **T3** and **T4**. Thus, the behaviour of the external environment is equivalent to a sub-set of a previously verified scenario. The trace presented in figure 7.23 can be projected onto the scenario analysed in section 7.7.3.1, where **T2=I**, **T3=T1**, **T4=T2** and **T7=T3**. Thus, this trace can be considered equivalent to the serial concatenation of two analysed scenarios, both of which are guaranteed to be deadlock-free and provide correct results to Queries.

This result indicates that stopping condition 1 is not dependent upon the Query's initial `hop_count` value and thus, can be generalised to apply to Queries with arbitrary initial `hop_count` values (i.e. initial `hop_count=n`).

7.9.2 Stopping condition 2

The trace shown in figure 7.24 is the same as that analysed in section 7.7.3.2 except that the Query terminates with `hop_count=2`, rather than with `hop_count=1`. The Query terminates because it runs out of links to follow which is independent of `hop_count`'s value (as discussed in section 7.7.3.2). As with stopping condition 1, independence from `hop_count`'s value indicates that stopping condition 2 is also valid for Queries that start with an arbitrary `hop_count` value.

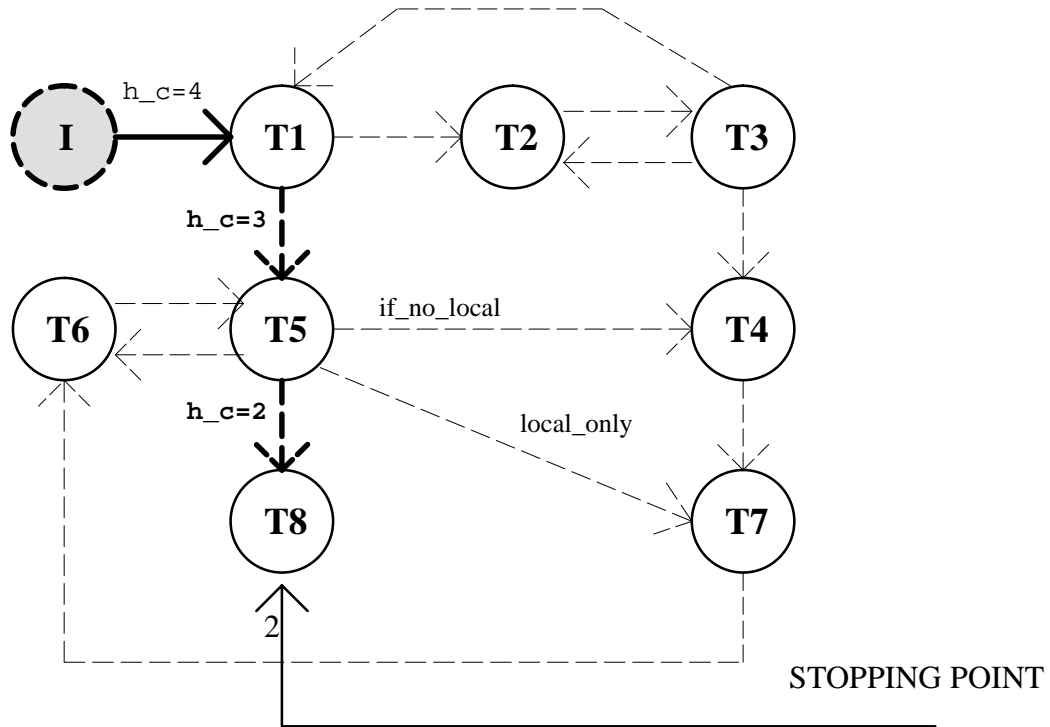


Fig. 7.24: Query trace 2

7.9.3 Stopping condition 3

The trace shown in figure 7.25 is similar to that analysed in section 7.7.3.3, except that the Query terminates with `hop_count=3` rather than `hop_count=1`. In either case, the `hop_count` value is non-zero, as discussed in section 7.7.3.3. The following mapping (from scenario to trace) can be used when comparing the scenario of section 7.7.3.3 with the trace of figure 7.25. `I=I`, `T1=T1`, `T2=T5`, `T3=T7`. As with stopping conditions 1 and 2, stopping condition 3 has been shown to be independent of a Query's initial `hop_count` value.

7.9.4 Stopping condition 4

The trace shown in figure 7.26 is similar to that analysed in section 7.7.3.4, except that the Query terminates with `hop_count=3` rather than `hop_count=1`. Termination of the Query in this scenario is independent of `hop_count`, as discussed in section 7.7.3.4. This indicates that stopping condition 4 is also applicable for Queries with arbitrary values of initial `hop_count`.

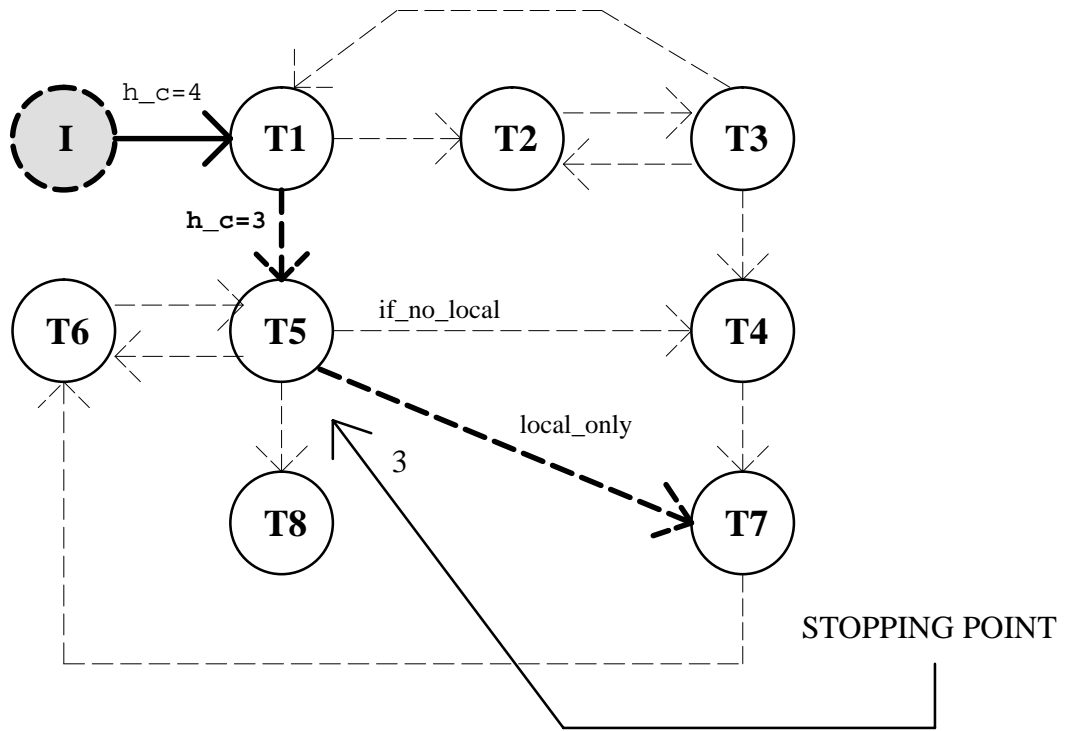


Fig. 7.25: Query trace 3

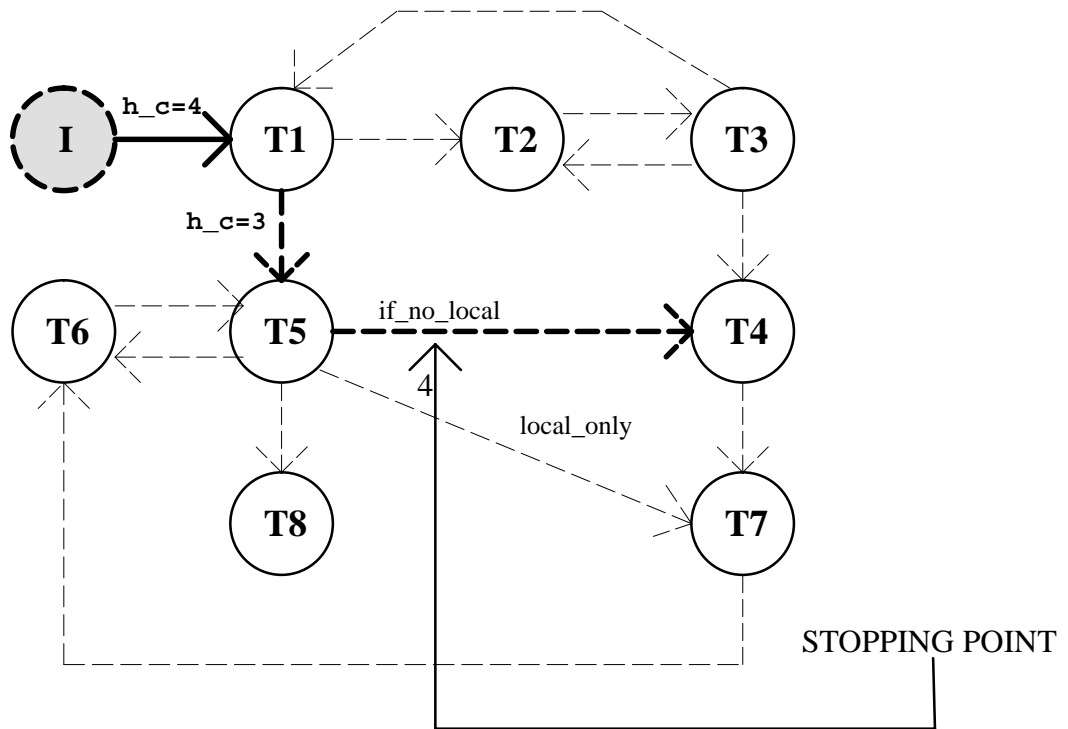


Fig. 7.26: Query trace 4

7.9.5 Stopping condition 5

The trace of figure 7.27 is similar to that analysed in section 7.7.3.5 except that the Query terminates with $hop_count=2$ rather than with $hop_count=1$. Termination of the Query in this scenario is independent of hop_count , as discussed in section 7.7.3.5. This indicates that stopping condition 5 is also applicable for Queries with arbitrary values of initial hop_count .

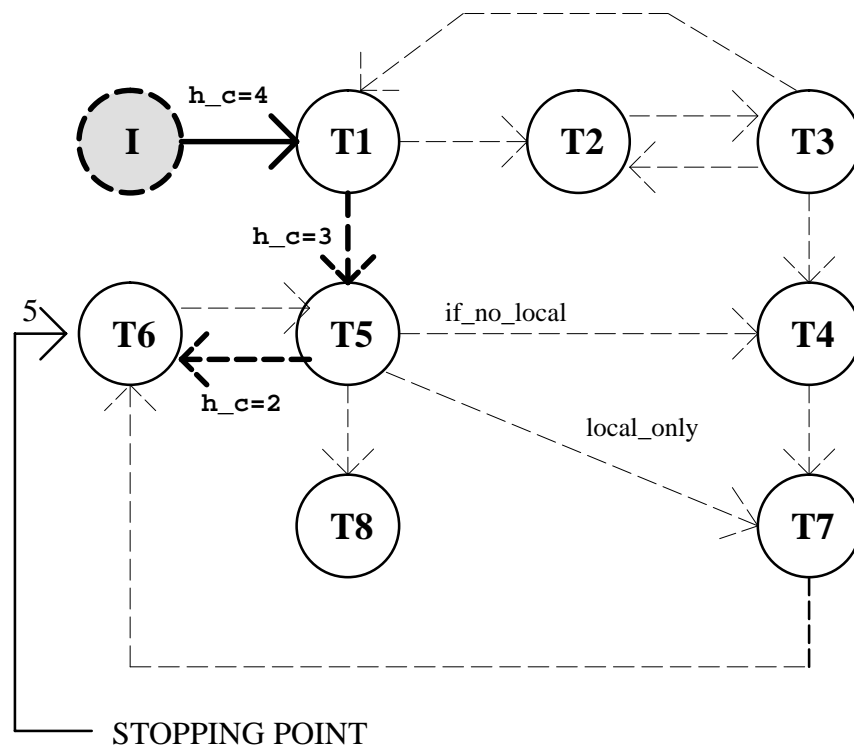


Fig. 7.27: Query trace 5

7.9.6 Stopping condition 6

The trace of figure 7.28 is identical to that analysed in section 7.7.3.6 which has been shown to behave correctly. Stopping condition 6 depends on a Query’s `hop_count` being less when a Trader is re-visited by a Query but it does not depend upon the initial value of `hop_count`. This indicates that stopping condition 6 is also applicable for Queries with arbitrary values of initial `hop_count`.

7.9.7 Conclusion

By applying the recursive Trading approach of section 7.7.1, it is possible to state that this complicated scenario is deadlock-free and is guaranteed to terminate with a single terminal marking if its OG was generated. This is because each of the linked Queries that are created as the initial Query propagates from **I** to **T1** and onwards have been verified individually in preceding sections. The fact that multiple concurrent Trader processing has also been shown to be **independent** (section 7.6.2) allows us to decompose the scenario into multiple concurrent smaller scenarios which have already been verified. This means that a complicated scenario such as that shown in figure 7.22 can be analysed as a series of concurrent traces which do not interact (see figure) and thus, can be analysed individually. This avoids the state space explosion problem associated with creating an OG for multiple concurrent Query propagation.

Consider an extension of this section’s analysis to a Query with arbitrary initial `hop_count` value and an arbitrary Trader interconnection topology. Each of the stopping conditions has been shown to be independent of a Query’s initial value for `hop_count`. As the Query is propagated along links, it will “break up” into a number of independent parallel Query “fragments” which

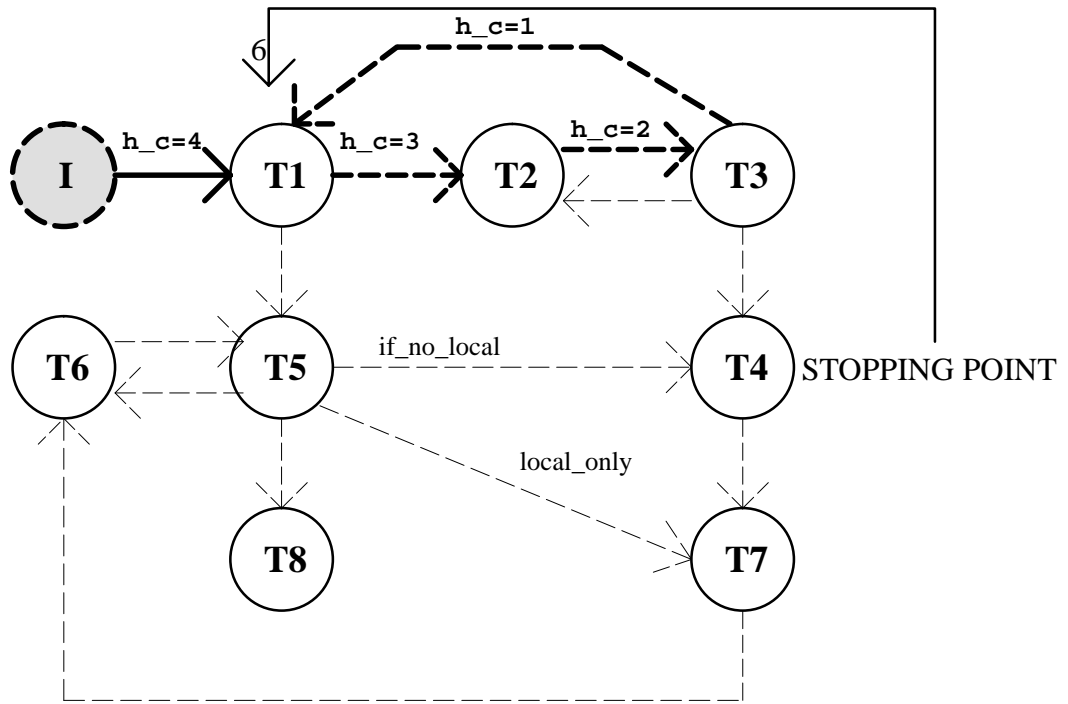


Fig. 7.28: Query trace 6

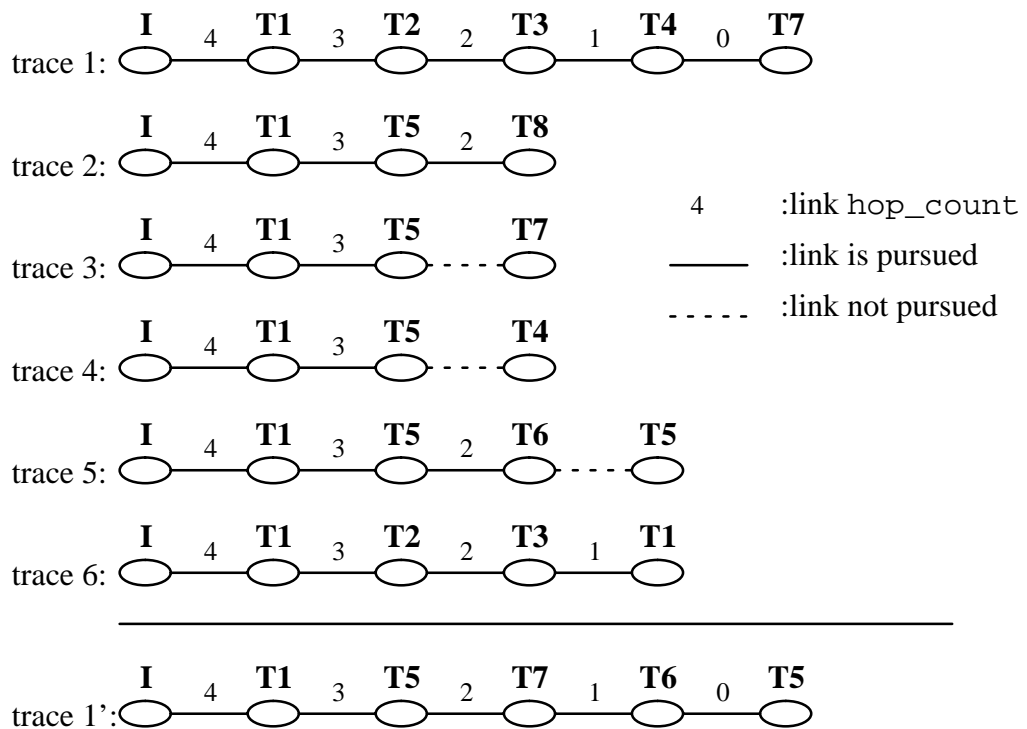


Fig. 7.29: Concurrent independent traces

have their hop_count value decremented as they traverse each link. Eventually, each of the independent parallel fragments will reach a stopping condition that has been analysed (and shown to behave correctly) in section 7.7.3. The fragments are no longer propagated and instead, each fragment returns its matching offers to the Trader which invoked the Query. Eventually, all Queries will return to the “initial” Trader and be collated into a set of offers that is returned to the Importer.

7.9.8 OG Analysis of Combined Scenario

The CPN model of the Trading Environment was configured as depicted in the scenario diagram of Figure 7.22. Although the creation of a complete OG for the scenario was not possible, analysis was performed using Simulation and the creation of a partial OG. The simulation run resulted in a list of Offers being returned to the Importer as expected and took 303 steps to complete. The partial OG was created by allowing the OG to be created normally (breadth-wise) for 10 seconds and then forcing it to create the OG depth-wise until complete. This resulted in a partial OG (statistics shown in table 18), which did not include all possible scenario behaviours but did narrow in on a single terminal marking (593) as expected.

OG
Nodes=5848
Arcs=7485
Seconds=519
PARTIAL Single Dead node=593

Table 18 – OG size for Combined Scenario after 519 seconds

This process was repeated a number of times and the same single terminal node (593) was evident (figure 7.30). This is a good result since it improves confidence in the assertion that the combined scenario has been verified using the previous scenario verification results.

7.10 Using High-level Petri Nets and Design/CPN for Analysis of OOBDSs

7.10.1 High-level Petri Nets

High-level Petri Nets are based on a well defined semantics and are therefore appropriate for formal analysis of OOBDS using tools. Tools are important since it is unlikely that High-level Petri Nets could be easily applied to modelling systems if all analysis was required to be done by hand.

Their application to analysis of large and complex systems can be limited by the significant resource requirements associated with generating a model's entire state space. This problem common to all Labelled Transition Systems and can be addressed by utilising state space reduction techniques as discussed in section 5.2.

The philosophy of calculating a model's entire state space relies on a brute force method that is intuitive to beginners.

7.10.2 Design/CPN

Design/CPN provides the modeller with many tools for analysis. The tool's integrated Simulator and OG generator are quite powerful and easy to use. As with using the tool for modelling, some familiarity with CPN/ML is necessary which could be a hindrance for beginners. Design/CPN's disk and memory requirements are quite large (8 MBytes of disk space required for a classical "Hello World" example).

The OEOS tool requires non-trivial contextual knowledge of the CPN model to be effectively applied in reducing the OG size. It would be useful if the tool was more user-friendly by aiding in the creation of Equivalence specifications.

Another issue is the ability to seamlessly import/export models across platforms. With Design/CPN 3.0, this is not a simple task when moving files from a Linux computer to a Solaris computer and vice-versa. Such functionality is important when the modeller wants to be able to model on one computer and perform analysis on a more heavily resourced computer.

7.11 Summary

In this chapter, the CPN model has been analysed using simulation and Occurrence Graph state space techniques. The aim of the analysis was to test the correctness of the model and therefore the computational viewpoint of the Trading standard.

Correct operation of the Trader was verified when servicing a single and multiple Queries concurrently as a standalone entity. It was also shown that the model behaves correctly when there are multiple instances of objects which operate concurrently and independently.

Using these results as a basis, it was demonstrated that arbitrarily complex Trader connection topologies can be verified since all topologies can be decomposed into a number of parallel Query propagation scenarios which were verified in this chapter.

The suggested improvements to the Trader's interworking protocol of section 6.5, involving removal of redundant Query propagation and ensuring deterministic traversal of the Trading offer space, were also analysed and found to be correct. A discussion regarding High-level Petri Nets and the Design/CPN tool with respect to the Analysis presented in this section was also presented.

593

```
Process_Req'IWT_Request_Store 1: 1'[{hop_count = 4,link_follow_rule =
always,request_id = {stem = importer,t_ident = 3},exact_match_type = TRUE}]
Process_Req'IWT_Request_Store 2: 1'[{hop_count = 2,link_follow_rule =
always,request_id = {stem = importer,t_ident = 3},exact_match_type = TRUE}]
Process_Req'IWT_Request_Store 3: 1'[{hop_count = 1,link_follow_rule =
always,request_id = {stem = importer,t_ident = 3},exact_match_type = TRUE}]
Process_Req'IWT_Request_Store 4: 1'[{hop_count = 1,link_follow_rule =
always,request_id = {stem = importer,t_ident = 3},exact_match_type = TRUE}]
Process_Req'IWT_Request_Store 5: 1'[{hop_count = 2,link_follow_rule =
always,request_id = {stem = importer,t_ident = 3},exact_match_type = TRUE}]
Process_Req'IWT_Request_Store 6: 1'[{hop_count = 1,link_follow_rule =
always,request_id = {stem = importer,t_ident = 3},exact_match_type = TRUE}]
Process_Req'IWT_Request_Store 7: 1'[{hop_count = 1,link_follow_rule =
always,request_id = {stem = importer,t_ident = 3},exact_match_type = TRUE}]
Process_Req'IWT_Request_Store 8: 1'[{hop_count = 1,link_follow_rule =
always,request_id = {stem = importer,t_ident = 3},exact_match_type = TRUE}]

Importer'Matched_Offers 1:
1'(off_l([(pizza,[ (MANDATORY,Pizza_Name,vegetarian)]),pizza_chef),
((pizza,[ (MANDATORY,Pizza_Name,vegetarian)]),pizza_chef),
((pizza,[ (MANDATORY,Pizza_Name,vegetarian)]),pizza_haven),
((pizza,[ (MANDATORY,Pizza_Name,vegetarian)]),pizza_haven),
((pizza,[ (MANDATORY,Pizza_Name,vegetarian)]),pizza_haven)]),3)

Offer_Space'Offer_Storage 1:
1'(t1_osi,([(pizza,[ (MANDATORY,Pizza_Name,vegetarian)]),pizza_chef),
((software,[ (MANDATORY,Software_Name,spreadsheet)]),advantek),
((used_car,[ (MANDATORY,Car_Type,wagon)]),bob_moran)])+
1'(t2_osi,([(pizza,[ (MANDATORY,Pizza_Name,the_lot)]),pedro_pizza),
((software,[ (MANDATORY,Software_Name,simcity)]),software_supermarket),
((used_car,[ (MANDATORY,Car_Type,wagon)]),aust_motors)])+
1'(t3_osi,([(pizza,[ (MANDATORY,Pizza_Name,vegetarian)]),pizza_haven)])+
1'(t4_osi,([(pizza,[ (MANDATORY,Pizza_Name,vegetarian)]),pizza_chef),
((software,[ (MANDATORY,Software_Name,spreadsheet)]),advantek),
((used_car,[ (MANDATORY,Car_Type,wagon)]),bob_moran)])+
1'(t5_osi,([(pizza,[ (MANDATORY,Pizza_Name,the_lot)]),pedro_pizza),
((software,[ (MANDATORY,Software_Name,simcity)]),software_supermarket),
((used_car,[ (MANDATORY,Car_Type,wagon)]),aust_motors)])+
1'(t6_osi,([(pizza,[ (MANDATORY,Pizza_Name,vegetarian)]),pizza_haven)])+
1'(t7_osi,([(pizza,[ (MANDATORY,Pizza_Name,the_lot)]),pedro_pizza),
((software,[ (MANDATORY,Software_Name,simcity)]),software_supermarket),
((used_car,[ (MANDATORY,Car_Type,wagon)]),aust_motors)])+
1'(t8_osi,([(pizza,[ (MANDATORY,Pizza_Name,vegetarian)]),pizza_haven]))

Link_Space'Link_Storage 1:
1'(t1_lsi,[{n = L1,lui = lui_2,regi = regi_2,def_fb = always,limit_fb = always},{n =
L2,lui = lui_5,regi = regi_5,def_fb = always,limit_fb = always}])+)
1'(t2_lsi,[{n = L3,lui = lui_3,regi = regi_3,def_fb = always,limit_fb = always}])+)
1'(t3_lsi,[{n = L4,lui = lui_2,regi = regi_2,def_fb = always,limit_fb = always},{n =
L5,lui = lui_1,regi = regi_1,def_fb = always,limit_fb = always},{n = L6,lui =
lui_4,regi = regi_4,def_fb = always,limit_fb = always}])+)
1'(t4_lsi,[{n = L7,lui = lui_7,regi = regi_7,def_fb = always,limit_fb = always}])+)
1'(t5_lsi,[{n = L8,lui = lui_4,regi = regi_4,def_fb = always,limit_fb = always},{n =
L9,lui = lui_6,regi = regi_6,def_fb = always,limit_fb = always},{n = L10,lui =
lui_7,regi = regi_7,def_fb = always,limit_fb = always},{n = L11,lui = lui_8,regi =
regi_8,def_fb = always,limit_fb = always}])+)
1'(t6_lsi,[{n = L12,lui = lui_5,regi = regi_5,def_fb = always,limit_fb = always}])+)
1'(t7_lsi,[{n = L13,lui = lui_6,regi = regi_6,def_fb = always,limit_fb = always}])+)
1'(t8_lsi,[])

Trad_Env'Comms_Medium 1: empty
```

Fig. 7.30: Reduced marking of node 593 from OG analysis of the combined scenario

Chapter 8

Prototyping the Trader

This chapter describes an implementation of the Trader which is used as a *concept demonstrator*. The implementation is derived from the CPN model in Chapter 6. The technologies utilised in the implementation are discussed, along with the program structure and the application domain of the prototype.

8.1 Motivation

With the emergence of Community Networking as discussed in Chapter 2, it is expected that there will be a need for a Trader that is easily accessible and operates on a global scale. It is most likely that the Internet would be used in such a scenario. As part of a preliminary investigation into such a Trader, a prototype Trading Environment would allow the concept of global Trading for E-commerce services to be demonstrated.

8.2 Aims and Scope of the Prototype

The creation of a prototype Trader had the following aims:

- To investigate the mapping from the CPN model to an implementation language,
- To demonstrate the application of the interworking protocol modifications presented in section 6.5,
- To determine an effective means for Users (Importers) to access the Trader.

The implementation of Trader does not aim to be a complete implementation of the ODP Trading Standard [16]. As with the CPN model of the Trader presented in Chapter 6, only basic trading functionality is provided (export and import of offers). The constraint language used for

limiting matched offers is not implemented, nor is a sophisticated database used for maintaining a large offer space.

8.3 Requirements

When creating the prototype Trader, a number of requirements were identified. The prototype should:

- act as a concept demonstrator for a system which allows Users to interact with a Trader via a simple interface and to locate matching services from a small offer space.
- be extendable to allow for the addition of extra functionality at a later date. This would allow the prototype to act as a skeleton for a more sophisticated and feature-packed implementation.
- provide a client that can be executed in a heterogeneous environment (i.e. multiple client platforms should be supported).
- allow easy access using a Graphical User Interface (GUI).
- be implemented in an open, distributed manner (since the Trader is part of an OOBDS).
- be capable of utilising the Internet (for an increased user base).

8.4 Design Decisions

Based upon the requirements outlined in section 8.3, it was necessary to consider certain design issues. The most important of these were:

- the implementation language,
- the method for supporting distribution, and
- the front-end to the Trader.

8.4.1 Implementation Language

The choice of implementation language is very important and is one of the most fundamental decisions when implementing a software system since it is highly dependent upon the system's requirements. Languages considered for prototyping of the Trading test-bed were C [148], C++ [149] and Java [128], a language created by Sun Microsystems [121] in 1995.

Java was selected as the implementation language for creating the Trading test-bed because it is well represented in a number of important areas outlined in subsequent sections. In some of the areas considered, C and C++ would be sufficient, but Java is considered the most appropriate language for the prototype based upon the requirements for a portable, internet-based distributed

application. When choosing an implementation language, its ability to provide the following features has a significant impact on its applicability for prototyping the Trader.

8.4.1.1 Object-oriented

The Trader is an object in an object-based system which suggests that it would be relatively easy to implement in an Object-oriented language. The Trader uses interfaces and provides information hiding through encapsulation. It would be possible to implement the Trader using a procedural language such as C, but it was felt that this could reduce the flexibility and maintainability of the implementation. C++ utilises much of the syntax of C, but has the added feature of supporting object-oriented implementation. Java is a true Object-oriented language and was built from the ground up with object-orientation in mind [117].

8.4.1.2 Distributed

As the Trader is an inherently networked and distributed application, it requires a programming language that provides good support for networking. Such a language would need to provide facilities that enable the prototype to operate over the Internet and also provide tightly-coupled support for distribution.

Java's Remote Method Invocation (RMI) [127] (see section 8.4.2.3), is a language-specific mechanism that allows Java objects to invoke methods on other Java objects that may be located in another Java Virtual Machine (JVM) located elsewhere on the network. Another possibility is for the language to be capable of interfacing with a distribution mechanism such as the Open System Foundation (OSF) Distributed Computing Environment (DCE) (using C or C++) or the OMG's CORBA which can be used by all three languages. A more basic method for creating a distributed prototype would be to use Unix style sockets for inter-object communication. Java provides the possibility to utilise each of these three options.

8.4.1.3 Portable

In an OOBDS, it is common for software running on different hardware and Operating system configurations to interact. Thus, it is likely that applications in an OOBDS may require porting to other platforms as required. With this in mind, it is desirable for the source code of the prototype to be portable since it is intended that Traders will be able to run on multiple platforms. Specifically, it is desirable to create one Trader source code that can be used on multiple platforms without the need to re-implement for specific platforms.

C and C++ can be quite portable depending upon the application, but may require significant changes and re-coding to accommodate multiple platforms using the same source code (i.e. using `#ifdef` to selectively compile parts of the source code depending upon the platform). This is highly undesirable since it greatly increases the complexity, and reduces maintainability and readability of the software being developed.

Java byte code is platform independent and may be executed on any platform that provides a JVM, without the need to re-compile. Thus, by using Java, it is possible to compile the source

code once and have an application that runs on multiple platforms using byte codes that execute in any JVM. This means that applications may be written once and then run on multiple platforms without requiring a porting development phase.

8.4.1.4 Multi-threaded

The CPN model of the Trader includes the concept of multi-threading which is used by the Trader to service multiple requests at once, thereby providing performance gains and removing the possibility of deadlock as discussed in section 7.5.2. Java provides simple, built-in language support for using threads in applications. Other languages such as C and C++ require third party libraries to provide threading, which can be very complicated to use and make the source code less portable.

8.4.1.5 Easy to program

In choosing the implementation language, the simplicity of the language is important. In the case of Java, the language was designed to leverage existing expertise in C and C++ programming whilst removing many of the error prone aspects of those languages. A simple language allows the creation of a prototype in less time and with less errors.

Java does not include the concept of a pointer, which is a very common source of programming error in C and C++ [117]. Memory management in Java is simple, since objects in the application are automatically removed by a garbage collector when they go out of scope (no longer referenced by another object). These features result in a language that is very simple to use, employs a similar syntax to C and C++ and removes the most common source of programmer error in those languages (memory allocation).

8.4.1.6 Performance

For the purposes of the prototype, performance issues are not a high priority since its main requirement is to be used as a concept demonstrator. In a commercial implementation, performance factors may need to be addressed including database efficiency, bandwidth consumption and the total processing time for a Query.

Java byte codes are interpreted and may be considered slow when compared with C and C++ applications. When using a Just In Time (JIT) compiler, Java byte code executes at around 2–3 times slower than native code. However, it is also possible to compile Java into native code which provides performance that is comparable to that of C and C++ [128].

8.4.2 Support for Distribution

The implementation must provide a means for distribution and provide objects with a simple mechanism for invoking methods on other objects. Having decided upon Java as a programming language, the distribution aspects of implementing a Trader were addressed. A number of methods that can be used to create a distributed implementation of the Trader are outlined in this section.

8.4.2.1 Sockets

Sockets are available for use by applications written in many languages including C, C++ and Java. Sockets were used to implement inter-object communication in the first prototype implemented using Java (refer to section 8.6.1.1). This required creating a custom application-specific protocol for message passing between objects. This approach was adopted for the first version of the prototype because there were no other mechanisms to use for implementing inter-object messaging in Java 1.02. Using Sockets is a reasonably straight-forward mechanism for creating a prototype application, with the following disadvantages:

- the time required to implement client and server side classes for a message passing protocol (re-inventing the wheel),
- the lack of interoperability with other applications – i.e. whatever protocol that uses the sockets for communication must be understood by all parties that wish to communicate. This makes the protocol very application-specific.

Implementing a message passing protocol using the socket facilities of Java 1.02 proved to be time consuming and did not provide a common and widely available interface for Java clients to utilise. This is an extremely important requirement for the Trader since many different importers must be able to communicate with it to utilise its services.

8.4.2.2 CORBA

The option of using CORBA was discarded for a number of reasons. Firstly, there was no Java mapping to CORBA IDL when implementation commenced, effectively ruling out using Java in combination with CORBA, although there were mappings to C and C++. In addition, there were no free public domain ORBs available (late in 1996) and when the OMG's CORBA IDL Java mapping was complete, ORBs were still not available free of charge for research purposes.

When creating a small scale distributed application in a homogeneous environment, CORBA was considered to be too heavy-weight since it is designed to operate in a heterogeneous environment. Both the clients and servers were to be written in the same language and operate in the same environment, namely, the JVM. Using CORBA would allow the Trader to provide an interface that is accessible to any other object, regardless of its implementation language and platform, through the use of CORBA's Internet Inter-Orb Protocol (IIOP) [44].

Since the prototype was to be built from the ground up, there was no requirement to interface with legacy systems which is an important reason to utilise CORBA.

8.4.2.3 RMI

With the release of Java Development Kit (JDK) 1.1, support was included for Remote method Invocation (RMI). RMI makes it possible for objects to reference each other across JVM boundaries in a convenient manner. In practice, this means that objects can invoke methods on

other objects running elsewhere on the network. RMI requires that objects define an interface of publicly available methods so that it can generate stubs and skeletons in the same manner as CORBA's IDL compiler.

In JDK1.1, Object Serialisation was also added to the language. It allows objects to be transferred over a byte stream which can be a file or network connection. RMI uses object serialisation to transport objects over the network for use as parameters when methods are invoked. RMI provides a standardised access technique for Java objects to interact with remote objects via interfaces.

Using RMI provides an alternative solution to using sockets or CORBA. It is easy to use and is seamlessly integrated into the Java language. RMI requires objects to define their interfaces and having done so, provides a transparent mechanism for invocation of methods on remote interfaces. RMI is homogeneous and thus, RMI interfaces are only accessible to applications that are written in Java. For the purposes of the Trading test-bed however, this is not a significant problem since both the client and server are written in Java.

In a recent press release [61], Sun Microsystems announced that RMI would implement CORBA's IIOP thereby providing support for CORBA interoperability. This has the effect of ensuring the longevity of RMI whilst providing users with the possibility of accessing all of the CORBA services that are being created. This access facility is not bi-directional, since RMI is a superset of CORBA's IIOP. In essence, IIOP will be used as an over-the-wire transport protocol by RMI.

RMI was chosen as the preferred distribution mechanism because it provides a clearly defined service that is tightly coupled to the language and can be used by the application programmer to create distributed Java applications. Choosing RMI as a communications mechanism resulted in re-working of the Trading test-bed communications code to convert it from being socket-based to utilising RMI.

8.4.3 Trader Front-end

The Trader front-end (client) is the user-interface for Importers, as described in the Trading model of section 3.4.1. For ease of use, a GUI should be available to the Importer (human user). There were two possible implementation options for the trading client which are outlined below.

8.4.3.1 Standalone Application

This option required the creation of a standalone application that could connect to the Trader via a network. This application must be able to run on multiple platforms, which is easy to achieve using Java since the JVM has been ported to many platforms. However, a standalone application would be difficult to keep current when object interfaces change. This is possible since the client front-end would need to be a continually evolving software program that is subject to continuous development.

8.4.3.2 Downloadable Client in a Browser

A solution to the versioning problem is for the application to be downloaded when required. This avoids the problem of version control since all downloaded versions can be considered to be current, excepting locally cached versions.

Most computer users are familiar with the Internet browser interface first used by the WWW consortium with Mosaic [123] and later enhanced by Netscape Corporation [124] and Microsoft [147]. The inclusion of a Java interpreter into browsers allows the downloading and execution of applets within the browser. Thus, it is possible to download a client applet into the browser simply by *pointing* the browser to a Uniform Resource Locator (URL) that identifies its parent html page.

This solution addresses versioning problems, allows for easy modification of the client-side applet and leverages the widespread use of platform-independent browsers for accessing the Internet in general. With these advantages in mind, it was decided to utilise Java applets which can be downloaded as required, and utilise a familiar interface.

8.5 Prototype Trading Domain

In order to demonstrate the application of Trader to electronic commerce, it was necessary to select a service application domain that could be traded. At present, the prototype may be used for trading simple electronic commerce services as seen in the CPN model.

8.6 Using the CPN Model for Implementation

The analysed CPN model was used as a design for the implementation. It identified objects in the system and the processes required to implement Trader functionality. Two implementations of the Trader were created, where the first version evolved into the second. This section describes the differences between them and the rationale behind the changes that were made in the second implementation.

8.6.1 Implementation based upon early CPN Models

The first prototype Trader was created using [118,119] and was based upon early CPN models of the Trader [27,28]. This section discusses the first Trader implementation with emphasis placed on the elements which changed in the later version.

8.6.1.1 Message passing

In the CPN model of Chapter 6, the structure of a `Message` token is defined and it used for communication between objects. The first implementation utilised a direct mapping from the CPN model for inter-object communication. It included a `Message` class which was capable of transmitting itself over a socket connection to an associated `Message` instance located on a client.

This is very similar to the Object Serialisation [126] work that was incorporated into Java 1.1 to allow objects to be transported over a network or data stream.

Using sockets required the development of customised client and server socket classes that could send a **Message** instance between each other. These socket classes required a significant amount of time to create, since there were many synchronisation issues to consider and significant testing and debugging required.

Using instances of **Message** is conceptually similar to the CPN model described in Chapter 6, where message objects (tokens) are sent between objects. To communicate, an object (such as the Trader) would create a new **Message** instance and fill out its fields. It would then invoke a method on the custom client socket instance to send the message instance to the custom server socket instance, which would recreate the **Message** instance at the destination object.

8.6.1.2 Multi-threading

In order to allow multiple Queries to be processed concurrently, the first prototype was developed using threads. This allowed a new thread to be created for servicing each Query **Message** that was received. If this threading was not used, the Trader would block while processing a Query which would not be desirable. Threads were not used within the control flow of objects but to allow concurrent serving of Queries.

8.6.1.3 Offer Space and Link Space objects

The Offer Space and Link Space object implementations are very similar since their functionality is almost identical. This is also the case in the CPN model of Chapter 6. Data structures required by the implementation were based upon those used in the model and the Trading specification itself.

Having modelled the Trader, it was simple to identify the required functionality of the Link and Offer Space objects. Thus, when implementing with Java, the CPN model was used directly to identify the necessary data structures and behaviour.

8.6.1.4 Limitations

The scope of the prototype's functionality is based upon that of the CPN model rather than the standard. Thus, the prototype cannot claim full compliance with the trading standard in all areas. The first implementation did not properly address interworking between Traders since this facility had not yet been added to the CPN model, nor finalised in the Trading Standard at the time [14]. The communications protocol was not *open* and thus, third party applets could not participate in Trading unless they had knowledge of the custom client and server socket classes, and the **Message** class used for inter-object communication.

8.6.2 Re-implementing using RMI and Trader Standard 96

Using a revised CORBA-aligned standard [16], the Trader CPN model was updated as discussed in section 6.12. These extensions included implementing the revised interworking protocol (discussed in section 6.5) and also using RMI.

8.6.2.1 RMI as a Communications Medium

As with the first implementation, objects in the CPN model were mapped into standalone Java applications. However, in the second implementation, RMI was used as the communications medium with object interfaces being defined using Java's *Interface* construct. Object functionality was described in accompanying source files which provided an implementation of the methods declared in the interface.

RMI allows objects to invoke methods of other objects which are outside its JVM in a transparent manner. Once an object reference to a remote object is obtained, it can be used as if the object was located in the same JVM. This transparent support for distribution makes the creation of a distributed object based system much easier, since the programmer is not required to be concerned with the mechanics of distributing the application.

In order to pass parameters by value, RMI uses the Object Serialisation feature [126] to transport objects and their entire class tree (all object instances which they contain) to the remote object. This allows the remote object to invoke methods on the transported object when executing the invoked method, since it has been transmitted to the remote object. When the method is complete, the parameter objects are serialised and sent back to their origin along with any return values.

Using RMI's "white pages" service known as *rmiregistry* [122], objects can advertise themselves by associating a name with their location. Using *rmiregistry*, applications can obtain object references using the name of the desired object as a parameter. This raises the possibility of using a Trader to provide a service-based match, rather than a simple naming service as supplied by *rmiregistry*.

8.6.3 CPN to Java mapping

The CPN model provided a design specification that was analysed for all possible behaviours. The mapping was done by hand, where:

- object instances (ie. Traders, OfferSpace and LinkSpace objects) located on Trad_Env#2 are mapped onto separate Java applications running in separate JVMs.
- object interfaces such as the Trader's **Lookup** and **Register** interfaces are mapped to an `interface` construct in Java that defines the methods provided by the interface.
- logic associated with methods modelled in the CPN is directly mapped into equivalent Java code that implements the methods advertised by the Java interface. Specifically, all

CPN/ML functions on arcs in the CPN model were re-written in Java using the CPN/ML code as a basis.

Appendix B includes a complete listing of the Java source code whilst Appendix F contains an archive of the pre-compiled and configured Test-bed.

8.6.3.1 Data structures and Logic

Data structures from the CPN model were re-used and converted to classes in Java. Decision logic in the CPN model was converted into `if-then-else` clauses in Java, which was not as intuitive as the CPN, since the `if` portion of the code could become many lines distant to the accompanying `else` clause. In the CPN model however, the relationship between the `if` and `else` clauses was easily traced due to the CPN's graphical nature.

8.6.3.2 Concurrency

The CPN's concurrent "logic" was transformed into multiple independent Java objects, where each object contained a single thread of control. However, each object was able to service multiple RMI calls concurrently since RMI guarantees that each RMI call from a different JVM will utilise a new thread. For example, the OfferSpace object is capable of servicing requests from multiple Traders concurrently.

In some cases, concurrency included in the CPN model was lost in the mapping to a Java implementation. For example, when the Trader sends requests to both the OfferSpace and LinkSpace objects in `Process_Req#14`, they are acted upon concurrently. Either of the two operations can complete first in the CPN model, but in the Java implementation, the remote methods are invoked serially. Java RMI ensures that each RMI method invocation will run in its own thread if it crosses a JVM boundary, but makes no guarantees in other cases.

It would be possible to create a thread for each RMI method invocation, thereby allowing each thread to execute its RMI method invocation concurrently. When developing the second prototype however, concurrent execution of RMI methods was not investigated due to time constraints.

Thus, the fundamentally different execution models of Petri Nets and procedural languages is resolved by mapping the inter-object concurrency modelled by the CPN into multiple concurrent Java applications that communicate using RMI which is guaranteed to support concurrency via multi-threaded method servicing.

8.6.3.3 Offer Space and Link Space objects

Little change was made to the Offer Space and Link Space objects, the major concern being to improve the data structures and configuration mechanisms. Both objects were adapted to use RMI and thus, required interfaces to be specified. Since these objects do essentially the same thing (maintain a simple database for client Traders), they use very similar source code. Thus, when the Offer Space object had been coded and tested using a test applet, much of the code was re-used in the Link Space

8.7 Implementation Source code

In this section, the source code for the RMI Trader is described using a class diagram to illustrate the relationship between classes in the implementation. A complete listing of the Java source code is included in Appendix B.

8.7.1 Class Hierarchy

A class hierarchy is shown in figure 8.1 and was automatically created by *javadoc*[122], a utility which is part of the JDK1.1 distribution. It creates HTML documentation from specially formatted comments contained within the Java source code. By following the javadoc commenting format, it is possible to automatically generate hypertext documentation directly from the Java source code. This is very useful, since it allows the easy generation of up-to-date documentation directly from the source code, ensuring that the source code and documentation are continually aligned. Classes which are underlined have a hypertext link to an associated HTML documentation page.

Many of the classes implement `serializable` which is required if they are to be used as parameters in RMI method invocations. The `LinkSpace`, `OfferSpace` and `Trader` classes all extend the `java.rmi.server.RemoteServer` class. This is because they are autonomous applications that can be accessed via RMI method invocations.

In figure 8.2, a block diagram is used to illustrate how instances of classes are related. Interface classes are shown in a box with a dotted outline and if a class contains a reference to another class (uses relationship), then the class which is being used is shown inside the class which uses it.

8.7.2 Interfaces

Interfaces provided by objects in the system (included in figure 8.1) are used to specify methods that the objects can perform and are located in source files called *Objectname.java*. The object's functionality is then fleshed out in an associated *Objectname_impl.java* file that implements the methods previously specified in the interface. For example, the `OfferSpace` object's interface is defined in *OfferSpace.java*, and the methods defined in the interface are implemented in *OfferSpace_impl.java*.

The **Trader** interface extends the **Lookup** and **Register** interfaces. **Admin**, **Link** and **Proxy** interfaces could be added to the `Trader` at this level if required. For the purposes of the test-bed, these interfaces were not implemented.

8.7.3 Data structure classes

There is a need for some utility objects which are used to manage data structures required by the `OfferSpace`, `LinkSpace` and `Trader` objects. Some of these classes are abstract data types that provide a record-like functionality, where all fields in the class are public. i.e. `linkStore`, `offerStore`, `reqIdRecord`.

Class Hierarchy

- class java.lang.Object
 - class java.awt.Component (implements java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable)
 - class java.awt.Container
 - class java.awt.Panel
 - class java.applet.Applet
 - class [LinkSpaceApplet](#)
 - class [OfferSpaceApplet](#)
 - class [TradGuiApplet](#) (implements java.awt.event.ActionListener, java.awt.event.ItemListener)
 - interface [LinkSpace](#) (extends java.rmi.Remote)
 - interface [Lookup](#) (extends java.rmi.Remote)
 - interface [OfferSpace](#) (extends java.rmi.Remote)
 - interface [Register](#) (extends java.rmi.Remote)
 - class java.rmi.server.RemoteObject (implements java.rmi.Remote, java.io.Serializable)
 - class java.rmi.server.RemoteServer
 - class java.rmi.server.UnicastRemoteObject
 - class [LinkSpace_impl](#) (implements [LinkSpace](#))
 - class [OfferSpace_impl](#) (implements [OfferSpace](#))
 - class [Trader_impl](#) (implements [Trader](#))
 - interface [Trader](#) (extends [Lookup](#), [Register](#))
 - class [impPolicyRecord](#) (implements java.io.Serializable)
 - class [linkRecord](#) (implements java.io.Serializable)
 - class [linkStore](#)
 - class [offerStore](#)
 - class [reqIdRecord](#) (implements java.io.Serializable)
 - class [servProps](#) (implements java.io.Serializable)
 - class [pizzaServRec](#) (implements java.io.Serializable)
 - class [usedCarServRec](#) (implements java.io.Serializable)
 - class [serviceRecord](#) (implements java.io.Serializable)
 - class [offerRecord](#)
 - class [traderPolicyRecord](#) (implements java.io.Serializable)

Fig. 8.1: Class hierarchy from javadoc

For example, the `linkStore` class contains information regarding links and provides methods such as `addLink` and `removeLink()`. It maintains the links for a given `Trader` using its internal data structures. These classes provide functionality which can be used by other classes to perform their tasks, such as for example the `LinkSpace` object.

8.7.4 Applets

Three applets were created for testing of the `OfferSpace`, `LinkSpace` and `Trader` objects. They can be used when embedded in an HTML file and loaded into *appletviewer* or a JDK1.1 capable browser. *Appletviewer* is provided with the JDK1.1 and allows applets to be executed outside a browser. These applets allowed testing of RMI using sample data and provide an easy to use

mechanism for issuing remote method invocations. For more information on Testing of the test-bed, refer to section 8.9.

8.7.5 Sample Services

As used in the CPN model, pizza and used car service providers were used as sample services in the prototype. These services are very simple and only have three service parameters each. They are enough, however, to demonstrate the interworking of multiple traders. Each of these services is stored in a class (**pizzaServRec** and **usedCarServRec**) that act as a record. These classes inherit from **servProps** which is an Abstract Data Type (ADT). All services implement a **similarity()** function that provides a metric of how similar two services are. They also implement a **setData()** function that takes elements read from a service data file and enters them into the appropriate fields in the service record.

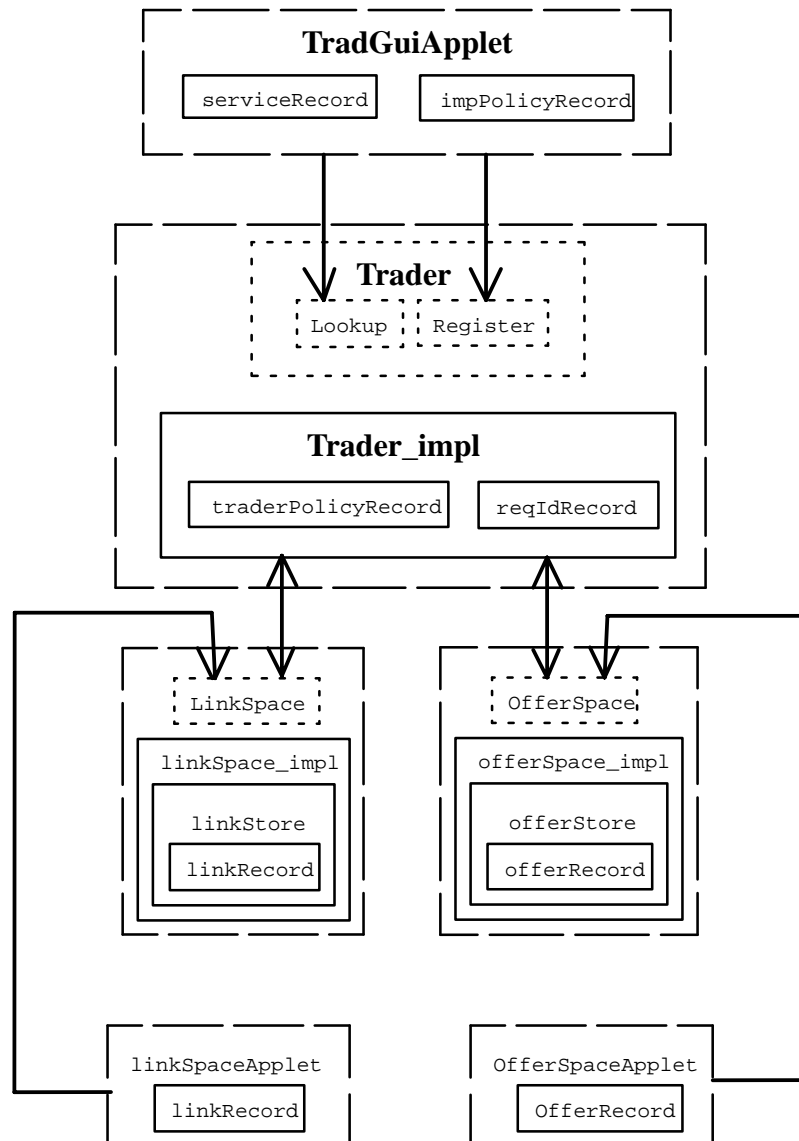


Fig. 8.2: Block Diagram of class relationships

If a type repository object was added to the Environment, the Trader would be able to determine the appropriate record based upon the service type and thus, be capable of initialising service records in a dynamic manner.

8.8 Running the Trading Environment

The following section provides instructions on how to get the Trading test-bed started. It assumes that the JDK1.1 has been successfully installed. Instructions are provided for a Windows 95 installation, but can be modified slightly for use on a computer running the Solaris operating system.

8.8.1 Compilation

The Java source files must be compiled into byte codes before the programs are executable. The Java compiler creates .class files which contain byte codes for the JVM. In order to compile the Java source code, the following command must be issued:

```
javac sourcefile.java
```

When invoking the compiler, the *.java* extension of the source file must be included on the command line. The system must be set up so that the CLASSPATH environment variable includes the JDK distribution's classes.

8.8.2 Starting the Objects in the System

In the basic system there are three types of objects that must be started: The Trader, Offer Space and Link Space objects. Prior to executing any objects in the system, the RMI Registry must be started. On a computer running Windows 95, this can be accomplished by typing:

```
start rmiregistry
```

The registry acts as a naming service, thereby allowing objects to register themselves and in doing so, binding a unique name to each object instance. This allows other objects to locate them locally or over the network.

To start the Trader, the user must enter:

```
java Trader_impl configfilename
```

where *configfilename* refers to the configuration file to be used for the Trader being started. Trader configuration files have the following format (italicised comments are not included in the file):

```
Trader_1           (trader_name)  
LUI_1             (lookup interface name)  
REG_1            (register interface name)  
2                 (default hop count)  
3                 (maximum hop count)  
LUI_1            (request id stem)  
if_no_local      (default follow rule)
```

```
always                (maximum follow rule)
OfferSpaceServer     (OfferSpace object name)
LinkSpaceServer      (LinkSpace object name)
```

The Offer Space and Link Space objects are executed using the following commands:

java OfferSpace_impl

These objects are initialised when they are created by reading from data files that contain each trader's offers and links. Offers are stored in files named **Trader_*.offers** and links are stored in **Trader_*.links**, where * denotes the Trader number. A small section of a sample offers file which illustrates the file format is given below (comments in italics).

```
Service                (keyword)
Used_Cars              (service type)
australian_motors     (service location)
The following lines contain parameters in the order required for the service type
record constructor (ie, usedCarServRec or pizzaServRec) until a blank line.
sports                (car type)
6                    (number of cylinders)
true                 (supports liquid petroleum gas (lpg))
```

An example of the format for the Link Space initialisation file for Trader_1's links is given below:

```
Link                  (keyword)
L1                    (name)
LUI_2                 (lookup interface name)
always                (default follow rule)
always                (limiting follow rule)
```

The links and offer space can be modified statically by editing the text data files used by the Offer and Link Space objects. Thus, arbitrary configurations of linked traders can be created by the addition/modification of the links for each Trader.

The Importer (client applet) can be started in the appletviewer which is provided with JDK1.1 and can be used to load and run applets. Alternatively, the Importer can be run within any Java 1.1 enabled browser such as Netscape's Navigator 4.01 which is bundled in the Communicator [124] suite of internet applications. Having followed these instructions, the test-bed is operating with one instance of the OfferSpace and LinkSpace objects, at least one Trader instance, and an Importer executing within the browser.

8.9 Testing of the Trading Environment

Objects were tested using sample data and client applets which were run in the JDK's appletviewer. The testing involved executing client applets which invoked methods on the object being tested.

8.9.1 OfferSpace

The **OfferSpaceApplet** was used to test the OfferSpace object. After accessing the **rmregistry** for a name lookup, it obtains an object reference to the OfferSpace object. It then creates three sample offers (one `pizza` offer and two `usedCar` offers) and adds them to the OfferSpace object on behalf of `Trader_1` using the `addOffers()` method. The applet then invokes the `printOffers()` method for both `Trader_1` and `Trader_2` objects, printing the current offers for each of those objects (three offers for `Trader_1` and none for `Trader_2`). This verifies that offers are created and stored successfully for each of the Traders.

Having printed the offers, a service record is created which is passed as a parameter when the `getOffers()` method is invoked. The request is for any matches with a `usedCar` service associated with `Trader_1`. Similarly, a request is also created for a `pizza` service associated with `Trader_2`. The ability to remove offers is also tested by printing `Trader_1`'s offers, removing an offer and then printing the offers again. Other methods such as `removeOffer()` and `sayHello()` were tested in previous versions of the applet.

8.9.2 Link Space

Testing of the LinkSpace object was similar to the testing of the OfferSpace object and was performed using the **LinkSpaceApplet**. Sample links were created and exported to the LinkSpace object using remote methods. As with the OfferSpace applet testing, the links were printed, one link deleted and the remaining links were printed again. This basic testing was sufficient to demonstrate correct functionality of the `addLink()`, `removeLink()` and `printLinks()` methods.

8.9.3 Trader

The Trader was tested firstly as a standalone and then when interworking. Both of these tests were performed using a client applet, as with the Offer and Link Space objects.

8.9.3.1 Standalone

The **TradGuiApplet** was used to test the Trader using a GUI that allows the user to select parameters and data values for new requests. Parameters for the Query such as `service type`, `hop_count` and `follow_policy` are able to be selected by the user.

The testing entailed creating requests for different services to ensure that the remote methods were being correctly invoked on the OfferSpace object. Since the Offer and Link Space objects had been previously tested, the Trader's internal logic was tested when it was directed to service a single non-interworking request.

8.9.3.2 Interworking Traders

The system was configured to contain four Traders, one OfferSpace and one LinkSpace object. The Offer and Link Space objects were compiled to initialise themselves from the appropriate files as described in section 8.8.2. The Traders were connected with the topology shown in figure

8.3, which is similar to that used in section 7.8 for analysis of deterministic offer space traversal by Queries.

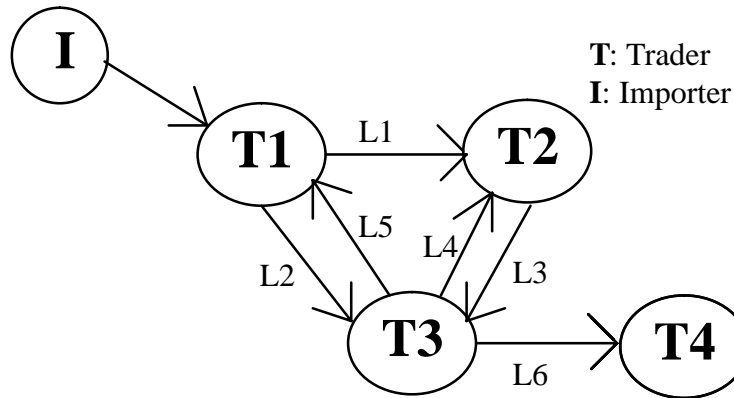


Fig. 8.3: Four linked Traders

This topology demonstrates a number of behavioural traits (when *hop_count*=2) that should be demonstrated including:

- Queries which are not re-sent to the Trader that immediately sent them,
- the deterministic traversal of a Query to **T4**.
- a Query which is not forwarded when its *hop_count*=0.
- a duplicate Query which is not processed locally and is not forwarded to linked Traders since its *hop_count* is less than that of a previously serviced Query.

For the purposes of the demonstration, only a single Importer is utilised, but it would be possible to use multiple Importers to concurrently query the four Traders in the system.

When connecting four Traders, it is possible to reproduce the trading scenarios presented in the analysis of section 7.7 and demonstrate the behaviours listed above. This provided an opportunity to experiment with Trading topologies and to examine Trader behaviour using the modified Interworking protocol.

A Query with the following values was created using the Importer's GUI.

```

service=pizza,
type=the_lot,
size=12,
delivery=true,
hop_count=3
LUI = LUI1
t_id = random
  
```

The Query is submitted to **T1** when the user presses the SUBMIT button in the GUI. **T1** searches locally for matching Offers and subsequently propagates the Query firstly to **T2** and then to **T3** with *hop_count* = 2. It is important to note that since these RMI calls are blocking and sequential, the Query is not sent by **T1** to **T3** until a response to the **T2** Query is obtained.

After **T2** has searched for a match locally, the Query is forwarded by **T2** to **T3** with a `hop_count = 2`. **T3** performs a local search of the offer space since the Query is unique. It also obtains its links from the `LinkSpace` object. It ignores the bi-directional link back to **T2** but pursues the links to **T1** and **T4**. The Query to **T1** is detected as a duplicate with a smaller `hop_count` and thus, **T1** returns an empty list of matching offers. However, at **T4** a matching offer is obtained. Since the Query's `hop_count=0`, no links need to be obtained. The result is returned to **T3** and then propagates back to **T2** and then finally back to **T1**.

At this point, the Query has been propagated by **T1** to **T2** but the **T3** link is yet to be explored. When the Query is sent to **T3** from **T1**, it is detected as being a duplicate with a larger `hop_count` than the Query it had previously serviced from **T2**, so a local search is not necessary, however a link search is necessary. The link to **T1** is ignored since it is bi-directional. The Query is forwarded to **T2** which detects it as a duplicate with a smaller `hop_count` and thus it returns an empty list of matching offers. **T4** is forwarded the Query and detects it to be a duplicate with a greater `hop_count` than the previous Query. Thus, it obtains its links from the `LinkSpace` object but is returned an empty list since no links exist for **T4**. Thus, **T4** returns an empty list of matching offers to **T3** which also returns an empty list of matching offers to **T1**.

T1 is able to return a matching Offer since the Query to **T2** resulted in a successful match (via interworking with **T4**). Since the `OfferSpace` object only contained a match to the Query for **T4**, the following list of matching offers was as returned to the importer:

```
[Pizza pizza_haven the_lot 12 true]
```

This represents an Offer provided by `pizza_haven` for a `Pizza` with `the_lot`, 12 inch diameter to be delivered.

A complete listing of the trace output from the above scenario is given in Appendix C. It is possible to visualise the distributed nature of the Trader service by inspecting the textual trace dumps. The reader is invited to utilise the test-bed source code (and pre-compiled byte code) and scenario scripts included in Appendix F for themselves.

Using the applet, interworking of Traders was tested using different `hop_counts`, service types and service parameters. By following the trace output from the Java code, it is possible to follow the execution of each Trader as requests are propagated between them. The Applet was also tested in Netscape Communicator 4.03 [124] which required a Java 1.1 patch to allow for the new Java 1.1 user interface event model. It is expected that Netscape will release a version of Communicator that supports both RMI and the Java 1.1 event model without the need for patching.

8.10 Limitations

The Trader that has been implemented does not comply with the Trading standard [16] in all areas. The constraint language used in offer selection has not been implemented. This may be added to the Trader by adding new functionality to the `Offer Space` object that interprets the

constraint language and performs searching accordingly. Since this implementation is a test-bed, it does not aim to be a complete implementation but rather an experiment into the applicability of Java when used for implementation of a Trader and validation of the Interworking protocol. In a full-scale implementation, it would be possible to use Java's JDBC (Java Database Connectivity) features to utilise advanced database functionality within the Offer Space object.

The GUI provided with the Client applet has not been optimised and provides an opportunity for further refinement. At present, the GUI allows the user to create simple queries to pre-defined services that exist in the Traders' offer spaces. A GUI that is able to dynamically query an Interface Repository to determine interface properties would provide a more flexible solution that could be applied to a more extensive prototype client.

8.11 Summary

A test-bed for the Trader has been implemented using Java and RMI. When creating a prototype Trader using Java, the mapping from CPNs to Java was investigated. It was found that the CPN model was readily mapped into Java, although in some cases, concurrent aspects of the CPN model were lost in the translation. The Trader which has been created in Java is of a preliminary nature and requires more work such as interfacing with a database and improvements to the User Interface.

The implementation utilises the interworking protocol improvements that were proposed in section 6.5. It has provided an opportunity to implement these optimisations and test them on a highly coupled Trader topology.

A browser-oriented client was developed which allows the Trader to be accessed from Java 1.1 compliant browsers. This is an easy way to access the Trader since the client applet can be embedded in a HTML page which can be downloaded from anywhere using the Internet.

The prototype Trader was useful for evaluating Java as an implementation language for creating a Trading service. It provides a basis for further refinement to bring the implementation into full compliance with the Trading Specification. Possible improvements include a full implementation of the constraint language, improved database functionality in the Offer Space object, inclusion of a type repository and a more sophisticated GUI for accessing the Traders.

Chapter 9

Conclusions

9.1 Contributions of the Dissertation

There are a number of significant results presented in this thesis covering the modelling, analysis and prototyping of the RM–ODP Trader.

An executable formal model of the Trader which was based upon the Trader’s computational and information viewpoints in the International Trading Standard [16] was developed. The CPN model includes the following features:

- multiple object instantiation (such as Traders, OfferSpace, LinkSpace, Importers and Exporters),
- generic object messaging interface (allows all objects to communicate),
- multiple threads of control within Trader objects,
- importing and exporting of offers by Importers and Exporters respectively,
- interworking between Trader objects.

The CPN model of the Trading environment is an example of an OOBDS. The CPN model was created as generically as possible with a view to maximising re–use when modelling different OOBDS’s.

The modelling phase resulted in the detection of some specification ambiguities in the Trader’s interworking protocol. As a result, a number of improvements to the interworking protocol were suggested.

The suggested improvements to the protocol ensure deterministic traversal of linked Traders and remove redundancy associated with following links that do not contribute to the set of

matched offers. The CPN model of the Trader was modified to reflect the suggested changes and was subsequently analysed using state space techniques.

The CPN model of the Trader and its environment was subsequently used for analysis of the Trader and its interworking protocol. Analysis was performed using a series of scenarios which were designed to test specific behavioural traits of the Trader under certain stopping conditions. Having analysed the simple scenarios, it was possible to reason about more complex scenarios. This is a new approach to analysis of OOBDS.

Occurrence Graphs with Equivalence classes were utilised in analysis of the Trader. This example of an OOBDS is a new domain for the application of OGs with Equivalence classes and is useful since it illustrates how Equivalence classes can be applied to the analysis of multi-threaded Object-based distributed systems.

The ongoing evolution of the Trading standard mandated an incremental approach to the modelling of the ODP Trader. The implications for the design and modification of CPNs to accommodate such an evolutionary model has been examined. This exercise also allowed the evaluation of CPN suitability for modelling OOBDS.

A variety of new application domains for Trading have been proposed, the most significant being the application of Trading for Electronic Commerce service location. A new Trading hierarchy scheme was proposed, where globally distributed Traders utilise multiple levels of links organised in a hierarchy to provide partitioning of interworking domains.

Using the CPN model as a basis, a prototype Trader was created using Java and RMI. This allowed a preliminary investigation into the applicability of using Java and RMI for the creation of distributed Internet applications. It also provided an opportunity to investigate the mapping from a CPN model into Java source code. The prototype Trader was used in a concept demonstrator for the application of Service Trading in an E-commerce environment.

9.2 Future Work

9.2.1 Extension of the Trader model

The Trader model could be extended to include detailed models of the Admin, Link and Proxy interfaces. This would allow investigation of the effects of dynamic link and Trader policy information with respect to interworking Traders. The Proxy interface could be added to the model and analysed to verify the Trading Standard's [16] specification of the interface and its functionality.

9.2.2 Stubborn Set Analysis of the Trader

With appropriate tool support, it would be possible to perform Stubborn Set analysis of the CPN model of the Trader developed during this thesis. Stubborn sets are known to be effective in reducing the state space of highly concurrent systems that contain multiple autonomous entities.

By combining stubborn sets with Equivalence classes, it is expected that the size of the model's OG would reduce significantly [79], thereby allowing further analysis of more complex trading scenarios.

9.2.3 Extended Trading Test-bed

An extension of the Java implementation could include a type repository which would provide a greater diversity and number of service types. The Offer Space object could be re-implemented using Java Database Connectivity (JDBC) library. This would allow the Offer Space to use a relational database for maintaining a much larger offer space and allow clients to perform complex Queries using the Trader's property constraint recipe language.

9.2.4 Investigate effective Trader topologies

Using the prototype, it would be possible to investigate effective topologies for Trading.. This could include applying hierarchies of links as proposed in Appendix D. At present, it is not clear what is an optimal scale and connectivity topology for Traders. Another area for future investigation is how to effectively partition the Trading offer space.

9.2.5 Investigate automatic mapping from CPN models to Java

Having created a detailed model of a system using CPNs, a direct mapping into a target language would allow models to be prototyped automatically. An automatic mapping from CPNs to Java code could be investigated since automatic code generation removes the possibility of adding errors to a prototype when hand-coding from a specification.

9.2.6 Extension of Interworking Analysis

An approach that can be used to analyse arbitrarily complex Trader Interworking topologies was demonstrated in Chapter 7. However, there remains scope for formalising the approach and for investigating other domains to which it may be applied.

References

ODP, CORBA and Trader

- [1] R. Soley, “*OMG: Creating Consensus in Object Technology*”, Proceedings of the 1st International Workshop on High Speed Networks and Open Distributed Platforms, St. Petersburg, Russia, June, 1995.
- [2] ISO/IEC Trading Tutorial (1996) “*Reference Model of Open Distributed Processing – Trading Function Annex A: Tutorial of the Trading Function*”. Available at: http://www.dstc.edu.au/AU/research_news/odp/trader/tr_tutorial.html
- [3] ISO/IEC DIS 10746–1 | ITU–T Recommendation X.901 (1995) “*Reference Model of Open Distributed Processing – Part 1: Overview and Guide to Use*”, Geneva, Switzerland.
- [4] ISO/IEC IS 10746–2 | ITU–T Recommendation X.902 (1995) “*Reference Model of Open Distributed Processing – Part 2: Descriptive Model*”, Geneva, Switzerland.
- [5] ISO/IEC IS 10746–3 | ITU–T Recommendation X.903 (1995) “*Reference Model of Open Distributed Processing – Part 3: Prescriptive Model*”, Geneva, Switzerland.
- [6] ISO/IEC DIS 10746–4 | ITU–T Recommendation X.904 (1995) “*Reference Model of Open Distributed Processing – Part 4: Architectural Semantics*”, Geneva, Switzerland.
- [7] Raymond, K. (1995) “*Reference Model for Open Distributed Processing (RM–ODP): Introduction*”, Open Distributed Processing: Experiences with distributed environments, K. Raymond, L. Armstrong (Eds), Chapman–Hall, 1995, pp. 3–14.
- [8] Linington, P., (1995) “*RM_ODP : The Architecture*”, Open Distributed Processing: Experiences with distributed environments, K. Raymond, L. Armstrong (Eds), Chapman–Hall, 1995, pp. 15–33.
- [9] ISO/IEC Trading (1993) “*Reference Model of Open Distributed Processing – Trading Function*”, ISO/IEC Project JTC1.21.59 | Draft ODP Trading Function. X.xxx, 12 November, 1993.
- [10] ISO/IEC Trading (1994) “*Reference Model of Open Distributed Processing – Trading Function*”, ISO/IEC DIS 13235 | Draft ODP Trading Function. X.9tr, 29 July, 1994.

- [11] ISO/IEC Trading (1994) “*Reference Model of Open Distributed Processing – Trading Function : Annex F – Interface Signature of the Computational Trading Function Specification in SDL/ASN.1*”, ISO/IEC CD 13235 (Annex F), 1994.
- [12] ISO/IEC Trading (1994) “*Reference Model of Open Distributed Processing – Trading Function : Annex G – Behaviour Specification of the Computational Trading Function Specification in SDL/ASN.1*”, ISO/IEC CD 13235 (Annex G), 1994.
- [13] ISO/IEC Trading (1994) “*Reference Model of Open Distributed Processing – Trading Function : Annex H – A sample Specification of a Concrete Trader Template and its Application using SDL/ASN.1*”, ISO/IEC CD 13235 (Annex H), 1994.
- [14] ISO/IEC Trading (1995) “*Reference Model of Open Distributed Processing – Trading Function*”, ISO/IEC DIS 13235 | Editors Draft DIS Text. X.9tr, 20 June, 1995.
- [15] ISO/IEC Trading (1996) “*Reference Model of Open Distributed Processing – Trading Function*”, ISO/IEC 2nd DIS 13235 | Revised DIS Text. X.950, 14 June, 1996.
Also available at:
http://www.dstc.edu.au/AU/research_news/odp/trader/standards.html
- [16] ISO/IEC Trading (1997) “*Reference Model of Open Distributed Processing – Trading Function*”, ISO/IEC 13235–1 (E) | Revised 2nd DIS Text for Publication. X.950–1, 20 January, 1997. Also available at:
http://www.dstc.edu.au/AU/research_news/odp/trader/standards.html
- [17] ISO/IEC Trading (1992) “*Working Document on Topic 9.1 – ODP Trader*”, ISO/IEC JTC1/SC21/WG7 N743.
- [18] Bietz, A., Berry, A., Lister, A. and Raymond, K.A. (1993) “*Introduction to Open Distributed Processing*”, Proceedings of the ACS Queensland Branch Conference – Overcoming Isolation: The Human–Computer Connection, Townsville, Australia, pp. 27–34.
- [19] A.P.M. Ltd., ANSAware 4.1 application programming in ANSAware, Document RM102.02, A.P.M. Cambridge Limited, Posiedon House, Castle Park, Cambridge CB3,0RD, U.K., February 1993.
- [20] Bietz, A., (1994) “*An overview of ANSAware’s Trader*”, Slides from the Trader Workshop, June 12, 1994. Available at:
http://www.dstc.edu.au/AU/research_news/odp/trader/presentations/aw_trader_slides.ps

- [21] Vogel, A., Bearman, M. and Beitz, A. (1995) “*Enabling Interworking of Traders*”, Open Distributed Processing: Experiences with distributed environments, K. Raymond, L. Armstrong (Eds), Chapman and Hall, 185–196.
Available at: <http://www.dstc.edu.au/AU/staff/andreas-vogel/papers/icodp95.ps>
- [22] CORBATrader at DSTC,
Information at: http://www.dstc.edu.au/AU/projects/corba_trader/fact_sheet.html
- [23] Brookes, W., Indulska, J., Bond, A., Yang, Z., (1995) “*Interoperability of Distributed Platforms: a Computability Perspective*”, Open Distributed Processing: Experiences with distributed environments, K. Raymond, L. Armstrong (Eds), Chapman–Hall, 1995, pp. 67–78.
- [24] Fischer, J., Prinz, A. and Vogel, A. (1993) “*Different FDT’s confronted with different ODP–viewpoints of the Trader*”, FME’93:Industrial Strength Formal Methods, J.C.P. Woodcock and P.G. Larsen (Eds), Springer–Verlag LNCS 670, pp. 322–350.
Available at:
http://www.dstc.edu.au/AU/research_news/odp/trader/papers/artikel.ps.gz
- [25] Tokmakoff, A., Billington, J. (1994) “*Consumer Services in Smart City Adelaide*”, Proc. of the 1st International Workshop on Home Oriented Informatics, Telematics and Automatics, Copenhagen, July 1994 pp. 201–210.
- [26] Tokmakoff A., Billington, J. (1995), “*Modelling of the ODP Trader for use in Resource Discovery*”, Proceedings of 2nd International Workshop on Community Networking, Princeton U.S.A., June 1995, pp. 155–162.
- [27] Tokmakoff, A., Billington, J. (1995) “*Coloured Petri Net Modelling of the ODP Trader for Use in Resource Discovery*”, 1st International Workshop on High Speed Networks and Open Distributed Platforms, Paper 1 Session 6, St. Petersburg, Russia, June 1995.
- [28] Tokmakoff, A., Billington, J. (1996) “*CPN Modelling of an Object Based System: The ODP Trader*”, Proceedings of 1st International Workshop on Formal Methods for Open Object–Based Distributed Systems, Paris, France, March 1996, pp. 245–260.
- [29] Tokmakoff, A., Billington, J. (1996) “*Modelling of Object Based Systems with Coloured Petri Nets*”, Proceedings of the IASTED International Conference on Modelling, Simulation and Optimisation, Gold Coast, Australia, May 1996, ISBN:0–88986–197–8.
- [30] Tokmakoff, A., Billington, J. (1996) “*Service Brokering in Object Based Systems: Advanced Information Services*”, 3rd IEEE Workshop on Community Networking, Antwerp, Belgium, May 1996, pp. 43–48.

- [31] Tokmakoff, A., Billington, J. (1996) “A CPN Model of an Object-based System”, 2nd Workshop on Object Oriented Programming and Models of Concurrency, 16th International Conference on Application and Theory of Petri Nets, Osaka Japan, June 1996, pp. 104–118.
- [32] Tokmakoff, A., Billington, J. (1996) “Using Coloured Petri Nets to aid the design of Object-based Systems”, Proceedings of the 1996 International Conference on Systems, Man and Cybernetics, Beijing, China, October 1996, pp. 3027–3032.
- [33] Billington, J., Tokmakoff, A., “Petri nets and Traders: Enabling technologies for virtual enterprises”, Presented at 6th IEEE Workshops on Enabling technologies: Infrastructure for Collaborative Enterprises (WETICE '97), Boston, U.S.A., June 1997.
- [34] Tokmakoff, A., Billington, J., (1997) “Service Trading in Mobile Environments”, Proc. of the 1997 International Conference on Information, Communications and Signal Processing: Trends in Information Systems Engineering and Wireless Multimedia Communications, IEEE Singapore Section, Singapore, September 1997, Volume 1 of 3, pp. 417–421.
- [35] Tokmakoff, A., Billington, J. (1998) “Reachability Analysis of the ODP Trader using Equivalence Classes”, Proceedings of the International Conference on Software Engineering: Education and Practice '98, Dunedin, New Zealand, 26–29 January, 1998, IEEE Computer Society Press, In Press.
- [36] Popien, C., Kuepper, A., (1996), “An Object-oriented description of services in a distributed system”, Proceedings of 1st International Workshop on Formal Methods for Open Object-Based Distributed Systems, Paris, France, March 1996, pp. 261–268.
- [37] Diagne, A., Estrailier, P. (1996), “Formal Specification and Design of Distributed Systems”, Proceedings of 1st International Workshop on Formal Methods for Open Object-Based Distributed Systems, Paris, France, March 1996 pp. 325–340.
- [38] Sinnott, R., Turner, K. J. (1995), “Applying the Architectural Semantics of ODP to Develop a Trader Specification”, Journal of Computer Networks and ISDN Systems, 29(4), March 1997, pp. 457–471.
- [39] Sinnott, R., Turner, K. J. (1996), “Specifying ODP Computational Objects in Z”, Proceedings of 1st International Workshop on Formal Methods for Open Object-Based Distributed Systems, Paris, France, March 1996, pp. 375–390.
- [40] Outhred, G., Potter, J., (1997), “An Enterprise Trader Model for DCOM”, Proceedings of the Joint IFIP International Conference on Open Distributed Processing and Distributed Platforms, Toronto, Canada, May 1997.

- [41] Sinnott, R., Turner, K. J. (1994), “*Modelling ODP Viewpoints*”, Workshop on Precise Behavioural Specifications in Object–Oriented Information Modelling, pp. 121–128, OOPSLA '94, Portland Oregon, October, 1994.
- [42] Dong, J. S., Duke, R. (1993) “*An Object–Oriented Approach to the Formal Specification of ODP Trader*”. Open Distributed Processing II, J. De Meer, B. Mahr, S. Storp (Eds), Berlin, North–Holland, pp. 341–352.
- [43] The Object Management Group Inc (1991), “*The Common Object Request Broker: Architecture and Specification Revision 1.1*”, OMG Press.
- [44] The Object Management Group Inc (1995), “*The Common Object Request Broker: Architecture and Specification Revision 2.0*”, July, 1995
Available at: <http://www.omg.org/pub/docs/orbos/96-03-04.pdf>
- [45] The Object Management Group Inc, “*OMG RFP5 Submission : Trading Object Service*”, May, 1996.
Available at: <http://www.omg.org/pub/docs/orbos/96-05-06.pdf>
- [46] The Object Management Group Inc, “*Description of New OMA Reference Model, Draft 1*”, May, 1996.
Available at: <http://www.omg.org/pub/docs/orbos/96-05-02.pdf>
- [47] Z. Yang and K. Duddy, (1995) “*Distributed Object Computing with CORBA*”, DSTC Technical Report 23, June 1995.
Available at: http://www.dstc.edu.au/AU/research_news/omg/report/report.ps
- [48] R. Orfali, D. Harkley, (1995) “*Client/Server with Distributed Objects*”, Byte April 1995, pp. 151–162.
- [49] H. Bowman, J. Derrick and M. Steen, (1995) “*Some results on cross viewpoint consistency checking*”, Open Distributed Processing: Experiences with distributed environments, K. Raymond, L. Armstrong (Eds), Chapman–Hall, 1995, pp. 397–412.
- [50] B. Kinane, (1995) “*Distributing Public Network Management Systems using CORBA*”, Open Distributed Processing: Experiences with distributed environments, K. Raymond, L. Armstrong (Eds), Chapman–Hall, 1995, pp. 117–130.
- [51] Raeder, G., Mazaher, S. (1995) “*Quality–of–Service Directed Targetting Based on the ODP Engineering Model*”, Open Distributed Processing: Experiences with distributed environments, K. Raymond, L. Armstrong (Eds), Chapman–Hall, 1995, pp. 372–383.
- [52] Waugh, A., Bearman, M (1995) “*Designing an ODP Trader Implementation using X.500*”, Open Distributed Processing: Experiences with distributed environments, K. Raymond, L. Armstrong (Eds), Chapman–Hall, 1995, pp. 133–144.

- [53] Davies, N., Friday, A., Blair, G.S., Chevert, K., (1997), “*Distributed Systems Support for adaptive mobile applications*”, Mobile Networks and Applications, Vol 1 No. 4, Baltzer Science Press, pp. 399–408.
- [54] Bates, J., Bacon, J. (1997), “*A Framework to support mobile users of multimedia applications*”, Mobile Networks and Applications, Vol 1 No. 4, Baltzer Science Press, pp. 409–419.
- [55] Diehl, N., Grill, D., Held, A., Kroh, R., Reigber, T., Zeigert, T., (1996), “*Mobile computing and service availability*”, Distributed Platforms, Schill, A., Mittasch, C., Spaniol, O., Poppien, C. (Eds), Chapman and Hall, pp. 44–56.
- [56] Friday, A. J., Blair, G. S., Cheverst, K. W. J., Davies, N. (1996), “*Extensions to ANSAware for advanced mobile applications*”, Distributed Platforms, Schill, A., Mittasch, C., Spaniol, O., Poppien, C. (Eds), Chapman and Hall, pp. 29–43.
- [57] Davies, N. (1996), “*The Impact of mobility on distributed systems platforms*”, Distributed Platforms, Proceedings of the IFIP/IEEE International Conference on Distributed Platforms: Client/Server and Beyond: DCE, CORBA, ODP and Advanced Distributed Applications, Schill, A., Mittasch, C., Spaniol, O., Poppien, C. (Eds), Chapman and Hall, pp. 18–25.
- [58] Netscape Communications Corp., (1996), “*New Netscape ONE Platform brings Distributed Objects to the Internet and Intranets*”, July 29 1996, Press Release.
Available at: <http://home.netscape.com/newsref/pr/newsrelease199.html>
- [59] Vinoski, S., (1997), “*CORBA : Integrating Diverse Applications Within Distributed Heterogenous Environments*”, IEEE Communications Magazine, February 1997, pp. 46–55.
- [60] Schmidt, D., (1997), “*Introduction to CORBA*”,
Available at: <http://www.cs.wustl.edu/~schmidt/corba-overview.html>
- [61] RMI and IIOP Press Release
Available at: <http://java.sun.com/pr/1997/june/statement970626-1.faq.html>

Petri Nets

- [62] Jensen, K. (1981) “*Coloured Petri Nets and the Invariant Method*”, Theoretical Computer Science 14, North-Holland, pp. 317–336.
- [63] Jensen, K., Rozenberg, G., (1991) “*High-level Petri Nets : Theory and Application*”, Springer-Verlag.

- [64] Dibold, H. (1992) “*Hierarchical Coloured Petri Nets for the Description of Services in an Intelligent Network*”, International Zurich Seminar on Digital Communications : IN and their Applications, 1992.
- [65] Billington, J., Wheeler, G. R. and Wilbur–Ham, M. C. (1988) “*PROTEAN : A High–level Petri Net tool for the Specification and Verification of Communication Protocols*”, IEEE Transactions on Software Engineering, Vol 14, No. 3.
- [66] Jensen, K. (1992) “*Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1 : Basic Concepts*”, EATCS Monographs on Theoretical Computer Science, Springer–Verlag.
- [67] Petri, C. A., (1962) “*Kommunikation mit Automaten*”, Schriften des IIM Nr. 2, Institut für Instrumentelle Mathematik, Bonn, 1962. English translation: Technical Report RADC–TR–65–377, Griffiths Air Force Base, New York, Vol. 1, Suppl. 1, 1966.
- [68] Jensen, K. (1994) “*Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2 : Analysis Methods*”, EATCS Monographs on Theoretical Computer Science, Springer–Verlag.
- [69] Jensen, K. (1997) “*Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 3 : Practical Use*”, EATCS Monographs on Theoretical Computer Science, Springer–Verlag.
- [70] Jørgensen, J.B., Mortensen, K. H. and Sousa, A. V., (1994) “*Modelling and Analysis of Distribution in BETA Using Coloured Petri Nets*”, Technical Report, Computer Science Department, Aarhus University, September 1994.
- [71] Billington, J., (1991), “*FORSEEing Quality Telecommunications Software*”, Proceedings of the First Australian Conference on Telecommunications Software, (ACTS) April 1991.
- [72] Billington, J., Wheeler, G. R. and Wilbur–Ham, M. C. (1988) “*PROTEAN : A High–level Petri Net tool for the Specification and Verification of Communication Protocols*”, IEEE Transactions on Software Engineering, Vol 14, No. 3.
- [73] Mortensen, K. H. and Pinci V. (1994) “*Modelling the Workflow of a Nuclear Waste Management Program*”, Application and Theory of Petri Nets 1994, LNCS 815, R. Valette (Ed), Springer–Verlag, pp. 376–395.
- [74] Wheeler, G., Valmari, A. and Billington, J. (1990) “*Baby TORAS Eats Philosophers but thinks about Solitaire*”, Proceedings of the Australian Software Engineering Conference (ASWEC) 1990.
- [75] Billington, J. (1991) “*FORSEEing Quality Telecommunications Software*”, Proceedings of the First Australian Conference on Telecommunications Software, (ACTS) April 1991.

- [76] Valmari, A. (1989) “*Stubborn sets for reduced state space generation*”, Advances in Petri Nets 1990, LNCS 483, Springer–Verlag 1990 pp. 491–515, originally appeared in Proceedings of 10th International Conference on Application and Theory of Petri Nets, Bonn, Vol II, pp. 1–22.
- [77] Valmari, A. (1994) “*State of the Art Report: Stubborn Sets*”, Construction and Analysis of Condensed State Spaces, Advanced Theory Tutorial Notes, 15th International Conference on the Theory and Application of Petri Nets, Zaragoza, Spain, 1994.
- [78] Tiisanen, M., (1994) “*Symbolic, Symmetry, and Stubborn Set Searches*”, Application and Theory of Petri Nets 1994, LNCS 815, R. Valette (Ed), Springer–Verlag, pp. 511–530.
- [79] Valmari, A. (1994) “*Stubborn Sets of Coloured Petri Nets*”, Construction and Analysis of Condensed State Spaces, Advanced Theory Tutorial Notes, 15th International Conference on the Theory and Application of Petri Nets, Zaragoza, Spain, 1994.
- [80] Moldt, D., (1994) “*Multi–Agent Systems based on Coloured Petri Nets*”, Application and Theory of Petri Nets 1997, LNCS 1248, P. Azema and G. Balbo (Eds), Springer–Verlag, pp. 82–101.
- [81] Lakos, C.A., (1994) “*From Coloured Petri Nets to Object Petri Nets*”, Technical Report R94–9, Networking Research Group, Department of Computer Science, University of Tasmania.
- [82] Lakos, C.A., (1997) “*On the Abstraction of Coloured Petri Nets*”, Application and Theory of Petri Nets 1997, LNCS 1248, P. Azema and G. Balbo (Eds), Springer–Verlag, pp. 42–61.
- [83] CPN Group, University of Aarhus (1996), Online Design/CPN User’s Manual.
Available at:
http://www.daimi.aau.dk/designCPN/man/X_2.0/REF/Reference.All.pdf
- [84] CPN Group, University of Aarhus (1996), Online Design/CPN Occurrence Graph User’s Manual.
Available at: **<http://www.daimi.aau.dk/designCPN/man/misc/OccGraph.All.pdf>**
- [85] CPN Group, University of Aarhus (1997), Online Design/CPN OEOS User’s Manual.
Available at: **<http://www.daimi.aau.dk/designCPN/libs/ogequiv/oeosman1–0.pdf>**
- [86] Wikstrom, A, (1987), “*Functional Programming using standard ML*”, Prentice–Hall.
- [87] Bastide, R. (1995) “*Approaches in unifying Petri nets and the Object–oriented Approach*”, Available at:
<http://www.dsi.unimi.it/Users/Labs/PetriLab/ws95/abstract/bastide.html>
- [88] ISO/IEC Petri Nets (1997) “*High–level Petri Nets – Concepts, Definitions, Graphical Notation*”, ISO/IEC CD 15909 V3.4, 2nd October, 1997.

Community Networking, Smart Cities and Information Infrastructure

- [89] Tredennick, N., (1996) “*Microprocessor Based Computers*”, IEEE Computer, October 1996, pp. 27–37.
- [90] Kraut, R., Scherlis, W., Mukhopadhyay, T., Manning, J., Kiesler, S., (1996) “*The Home Net Trial of Residential Internet Services*”, Communications of the ACM, December 1996, pp. 55–63.
- [91] Carroll, J. M., Rosson, M.B, (1996) “*Developing the Blacksburg Electronic Villiage*”, Communications of the ACM, December 1996, pp. 69–74.
- [92] City of Palo Alto, **Information at:** <http://www.city.palo-alto.ca.us>
- [93] Joint Venture Silicon Valley, **Information at:** <http://www.jointventure.org/>
- [94] Singapore IT2000, **Information at:** <http://www.ncb.gov.sg/>
- [95] MFP Australia Py. Ltd., **Information at:** <http://www.mfp.com.au/>
- [96] Connect 96 Prjoect Profile : MFP Australia Py. Ltd.
Available at: <http://www.svi.org/connect96/Profiles/MFP.htm>
- [97] Seattle Community Network, **Information at:** <http://www.scn.org/>
- [98] Bee, A., (1996) “*In search of speed and bandwidth*”, IEEE Computer, December 1996, pp. 12–14.
- [99] Anderson, H., (1997) “*Money and the Internet: a strange new relationship*”, IEEE Spectrum, February 1997, pp. 74–76.
- [100] Fancher, C. H., (1997) “*In your pocket: smartcards*”, IEEE Spectrum, February 1997, pp 47–53.
- [101] Baldwin, R. S., Chang, C. V., (1997) “*Locking the e-safe*”, IEEE Spectrum, February 1997, pp.40–46.
- [102] Rhodes, L. (1996), “*The Race for more Bandwidth*”, Wired 4.01, January 1996, pp. 140.
- [103] Sakakibara, I, Motohashi, Y, (1996) “*Trial of CATV, VOD and ISDN Services over FTTH*”, 3rd IEEE Workshop on Community Networking, Antwerp, Belgium, May 1996, pp. 33–38
- [104] Tsuji, H., Ohta, N., Suto, K., Morisaki, M., (1996) “*Regional PC Community/Group Communication Network Service System Driving FTTH Forward*”, 3rd IEEE Workshop on Community Networking, Antwerp, Belgium, May 1996, pp. 1–5.
- [105] Halfhill, T. R., (1996) “*Break the Bandwidth Barrier*”, Byte Magazine, September, 1996 pp. 68–80.

- [106] Byte Magazine, (1996) “*Lab Report: Bandwidth on a Budget: 34 Fast Modems*”, Byte Magazine, October 1997, pp. 76–85.
- [107] Shuler, D., (1996), “*New Community Networks – Wired for Change : Abstracts*”,
Available at: <http://www.scn.org/ip/commnet/abshome.htm>
- [108] Cowham, S., (1997), “*People power blocks cables*”, Guardian Messenger, May 21, 1997 pp.3.
- [109] Slonim, J., Bauer, M., (1995) “*New ways of Learning through the Global Information Infrastructure*”, Open Distributed Processing: Experiences with distributed environments, K. Raymond, L. Armstrong (Eds), Chapman–Hall, 1995, pp. 34–49.
- [110] Greenwald, J., (1997) “*Thinking Big*”, Wired Magazine 5.08, August 1997, pp. 95–144.
- [111] Multimedia Development Corporation
Information available at: <http://www.mdc.com.my/>
- [112] Multimedia Development Corporation: Infrastructure: Cyberjaya
Information available at: <http://www.mdc.com.my/infra/cyberjaya/environ/>

TINA

- [113] W. J. Barr, T. Boyd and Y. Inoue, (1993) “*The TINA Initiative*”, IEEE Communications Magazine, March 1993, pp. 70–76.
- [114] G. Nilsson, F. Dupuy and M. Chapman, (1995) “*An Overview of the Telecommunications Information Networking Architecture*”, Proceedings of TINA 95, Melbourne, pp. 1–12.

Distributed Operating Systems

- [115] W. Zhu, C.F. Steketee and B. Muilwijk, (1995) “*Load Balancing and Workstation Autonomy on Amoeba*”, Proceedings of 18th Australasian Computer Science Conference, Glenelg, South Australia, February 1995, pp. 588–597.
- [116] J. Mitchell, J. Gibbons, G. Hamilton, P. Kessler, Y. Khalidi, P. Kougiouris, P. Madanay, M. Nelson, M. Powell, S. Radia, (1994) “*An Overview of the Spring System*”, Proceedings of Compcon, February 1994.
Available at: <http://www.sun.com/techprojects/spring/papers/overview.ps>

Java

- [117] Sun Microsystems, “*The Java Language: A White Paper*”,
Available from: <http://java.sun.com/doc/overview/index.html>

- [118] Cornell, C., Horstmann, C.S., (1996), “*Core Java*”, Sunsoft Java Series, ISBN 0–13–56755–5.
- [119] Campiogne, M., Walrath, K., (1998), “*The Java Tutorial– online version*”,
Available from: <http://java.sun.com/docs/books/tutorial/>
- [120] Bank, D., (1995), “Java Saga”, Wired, December 1995, pp. 166.
- [121] Sun Microsystems. **Info at:** <http://www.sun.com/>
- [122] Java Development Kit (JDK) 1.1. **Available at:** <http://www.javasoft.com/>
- [123] University of Illinois at Urbana–Champaign, Mosaic Browser, **Info at:**
<http://www.ncsa.uiuc.edu/SDG/Software/MacMosaic/MacMosaicHome.html>
- [124] Netscape Corporation, **Info at:** <http://www.netscape.com/>
- [125] Sun Microsystems Distributed Systems Group,
Located at: <http://chatsubo.javasoft.com/>
- [126] Sun Microsystems, JDK Serialization Documentation
Located at: <http://www.javasoft.com/products/jdk/1.1/docs/guide/serialization/>
- [127] Sun Microsystems, JDK RMI Documentation,
Located at: <http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/>
- [128] van Hoff, A., (1997), “*The case for Java as a programming language*”, IEEE Internet Computing Magazine, January/February 1997, pp. 51–56.
- [129] General Magic, Odyssey – Mobile Java Agents
Located at: <http://www.genmagic.com.agents/>
- [130] ObjectSpace, Voyager – Mobile Java Agents
Located at: <http://www.objectspace.com/Voyager/>
- [131] Kiniry, J., Zimmerman, D., (1997), “*A Hands–On Look at Java Mobile Agents*”, IEEE Internet Computing Magazine, July/August 1997, pp. 21–30.

FDTs

- [132] ISO/IEC (1989) “*LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*”, ISO/IEC 8807, International Organisation for Standardisation, Geneva.
- [133] ISO/IEC (1989) “*A Formal Description Technique based on an Extended State Transition Model*”, ISO/IEC 9074, International Organisation for Standardisation, Geneva.

- [134] CCITT (1992) “*Specification and Description Language*”, CCITT Z.100, International Consultative Committee on Telegraphy and Telephony, Geneva.
- [135] ITU–T (1996) “*Message Sequence Chart*”, Z.120, ITU–T, Geneva.
- [136] Turner, K. J. (Ed) (1995), “*Using Formal Description Techniques: An Introduction to ESTELLE, LOTOS and SDL*”, John Wiley, N.Y., U.S.A.
- [137] ITU–T (1996) “*Message Sequence Chart (MSC)*”, ITU–T, Z.120, Geneva, April 1996.
- [138] Spivey, J. (1992), “*The Z Notation: a Reference Manual*”, Prentice–Hall International Series in Computing Science: C. A. R. Hoare Series Editor, Second Edition, Prentice–Hall International.
- [139] Duke, R., King, P., Rose, G., Smith, G., (1991), “*The Object–Z Specification Language*”, Technology of Object–oriented Languages and Systems; TOOLS 5, T. Korson, V. Viashnavi, B. Meyer (Eds), pp. 456–483, Prentice–Hall.
- [140] Rudolph, E., Grabowski, J., Graubmann, P., (1996), “*Tutorial on Message Sequence Charts (MSC’96)*”, Tutorial of the FORTE/PSTV’96 conference in Kaiserslautern, Germany, Oct 1996.

Misc

- [141] Garland, S., J., (1986), “*Introduction to Computer Science with applications in Pascal*”, Addison–Wesley, 1986, ISBN 0–201–04398–X.
- [142] US Robotics Corporation, **Information at:** <http://www.usr.com/>
- [143] Rockwell Corporation, **Information at:** <http://www.rockwell.com/>
- [144] Telstra Corporation, **Information at:** <http://www.telstra.com.au/>
- [145] The Object Management Group Inc, **Information at:** <http://www.omg.org/>
- [146] Optus Communications, **Information at:** <http://www.optus.com.au/>
- [147] Microsoft Corporation, **Info at:** <http://www.microsoft.com/>
- [148] Kernighan, B.W., Ritchie, D. M., (1988), “*The C Programming language*”, 2nd Edition, Prentice–Hall.
- [149] Stroustrup, B. (1991), “*The C++ Programming language*”, 2nd Edition, Addison–Wesley.

Appendix A

CPN model of the Trader

This Appendix includes code segments of the CPN model which were not included in their entirety in the Thesis body.

A.1 Global Declaration Node (GDN#1)

```
(** Global Declaration node **)

(** Identifiers of all the Objects in the System **)
color Id = with
    (** Trading Objects **)
    importer      |
    exporter      |
    search        |
    link_space    |
    offer_space   |
    (** Generic Interface Identifiers **)
    OSI           |
    LSI           |
    (** Lookup Interfaces **)
    lui_1         |
    lui_2         |
    lui_3         |
    lui_4         |
    (** Register Interfaces **)
    regi_1        |
    regi_2        |
    regi_3        |
    regi_4        |
    (** Offer Space Interfaces **)
    t1_osi        |
    t2_osi        |
    t3_osi        |
    t4_osi        |
    (** Link Space Interfaces **)
    t1_lsi        |
    t2_lsi        |
    t3_lsi        |
```

```

        t4_lsi      |
(** Service Providers **)
(**- Pizza providers -**)
    pizza_haven |
    pedro_pizza |
    pizza_chef  |
(**- Software providers -**)
    software_supermarket |
    advantek    |
    virgin      |
(**- Used Car providers -**)
    bob_moran   |
    aust_motors |
    champion    |
    NO_ID;

(** Colour defining the System's set of Lookup and Register Interfaces **)
color LUInt = subset Id with [lui_1,lui_2,lui_3,lui_4] declare in;
color RegInt = subset Id with [regi_1,regi_2,regi_3,regi_4] declare in;

(** Colour defining the System's set of OfferSpace Interface,
    and LinkSpace Interfaces **)
color OSInt = subset Id with [t1_osi,t2_osi,t3_osi,t4_osi] declare in;
color LSInt = subset Id with [t1_lsi,t2_lsi,t3_lsi,t4_lsi] declare in;

(** Generic Interface Descriptors **)
color Interfaces = subset Id with [OSI, LSI] declare in;

(** Names associated with Links **)
color Link_Name = with L1 | L2 | L3 |L4 | L5 | L6;

(** Definition of Link Follow Options **)
color Follow_Option = with no_option | local_only | if_no_local | always;

(** Link Data Structure including :
    Name, Lookup Interface, Register Interface,
    Default and Limiting Follow Behaviour **)
color Link = record n:Link_Name*
    lui:LUInt*
    regi:RegInt*
    def_fb:Follow_Option*
    limit_fb:Follow_Option;

(** A list of Links **)
color Link_List = list Link;

(** Identifiers for all of the possible Operations in the model **)
color Operation = with add_offer      |
    get_offers      |
    import          |
    query_offer_graph |
    export          |
    withdraw        |
    modify          |
    describe        |
    add_link        |
    remove_link     |
    modify_link     |
    describe_link   |
    list_links      |
    evaluate        |

```

```

        get_links      |
        error          |
        respond       |
        submit_order  |
        acknowledge;

(** Set of General Operations common to all objects **)
color Gen_Ops = subset Operation with
    [error, respond, acknowledge] declare in;

(** Set of Operations specific to the Link Space object**)
color Link_Ops = subset Operation with
    [get_links] declare in;

(** Set of Operations specific to the Offer Space object**)
color Offer_Ops = subset Operation with
    [add_offer, get_offers] declare in;

(** Set of Operations specific to the Lookup Interface **)
color Import_Ops = subset Operation with
    [import, query_offer_graph] declare in;

(** Set of Operations specific to the Register Interface **)
color Export_Ops = subset Operation with
    [export, withdraw, modify, describe] declare in;

(** Set of service interface identifiers. Same as the
    set of object identifiers **)
color Serv_Int_Id = Id;

(** Set of service types **)
color Int_Type = with pizza |
                software |
                used_car;

(** Set of possible service property modes **)
color PropertyMode = with NORMAL | READONLY | MANDATORY | MANDATORY_READONLY;

(** Set of service property names **)
color PropName = with Pizza_Name |
                Software_Name |
                Engine_Size |
                Car_Type;

(** Set of service property values **)
color PropValue = with the_lot |
                vegetarian |
                simcity |
                spreadsheet |
                six_cylinder |
                three_litre |
                sports |
                wagon;

(** Definition of a service property.
    Contains the Property Mode, Property name and the property value **)
color Properties = product PropertyMode*PropName*PropValue;

(** List of service properties **)
color Prop_List = list Properties;

```

```

(** A Service Type contains the Interface Type and a list of
    properties associated with the service available at the interface **)
color Serv_Type = product Int_Type*Prop_List;

(** Service Offer contains the Service Tyoe and a Service Interface Identifier
    to unquely identify the service and its provider **)
color Serv_Off = product Serv_Type*Serv_Int_Id;

(** General Colour Sets for use within the model **)
color boolean = bool with (FALSE, TRUE);

(** Small Integer ranges from -1 to 4 **)
color Small_Int = int with ~1..4;

(** Transaction identifier ranges from -1 to 4 **)
color Trans_Id = Small_Int;

(** Shorthand for Null number **)
val NO_NUM = ~1:Small_Int;

(** Stem used in idetifying the originator of a Request **)
color Request_Stem = Id;

(** Request_Id contains the request stem and also a transaction
    identifier. These elements uniquely identify a request from a
    specific object **)
color Request_Id = record stem:Request_Stem*
                        t_ident:Trans_Id;

(** List of request identifiers used by the Trader to record Queries
    that have been serviced **)
color Req_Id_List = list Request_Id;

(** Import Policy specified by the Importing Object **)
color Imp_Policy = record hop_count:Small_Int*
                        link_follow_rule:Follow_Option*
                        request_id:Request_Id*
                        exact_match_type:boolean;

(** List of import policies **)
color Imp_Policy_List = list Imp_Policy;

(** Record containing the Trader's references to its interfaces **)
color Trader_Components = record lookup_if:Id*
                                register_if:Id*
                                admin_if:Id;

(** Trader Import Attributes used to limit Trading behaviour **)
color Import_Attributes = record def_hop_count:Small_Int*
                                max_hop_count:Small_Int*
                                def_follow_policy:Follow_Option*
                                max_follow_policy:Follow_Option*
                                req_id_stem:Request_Stem;

(** A list of Offers used by the Offer Space Object **)
color Offer_List = list Serv_Off;

(** A list of offers and an associated transaction identifier **)
color Offer_List_Record = product Offer_List*Trans_Id;

(** Set of possible error conditions **)

```

```

color Error = with error_p | (** Parameter **)
                error_s | (** Security **)
                error_op | (** Operation **)
                error_exp; (** Export **)

(** Acknowledgement may be qualified or absolute **)
color Ack = with OK |
                qualified;

(** Generic NULL identifier
    Used when operations have no parameters.
    i.e., Op_Elem = null **)
color Null = with NULL;

(** Import Request contains the Service Type required and
    an associated Import Policy **)
color Imp_Req = product Serv_Type*Imp_Policy;

(** An operation element is selected from the union of these elements,
    depending upon the object which the operation element is intended for **)
color Op_Elem = union err:Error+
                imp:Imp_Req+
                exp_return:Small_Int+
                ack:Ack+
                none:Null+
                off:Serv_Off+
                link_l:Link_List+
                serv_t:Serv_Type+
                off_l:Offer_List;

(** Addressing mechanism that requires a sender and receiver object
    identifier **)
color Addresses = product Id*Id;

(** Operation Data contains the operation element and a transaction
    identifier **)
color Op_Data = product Op_Elem*Trans_Id;

(** Duplicate Data contains an Import Policy, an Operation Element and
    an associated transaction identifier. Used in the Process_Request
    page when processing duplicate messages **)
color Dup_Data = product Imp_Policy*Op_Elem*Trans_Id;

(** Data payload for messages contains the message operation and the
    Operation Data **)
color Data = product Operation*Op_Data;

(** Message is defined as a product of addressing information and
    the data contained within the message **)
color Message = product Addresses*Data;

(** Test is used for modelling syntax and parameter checks
    non-deterministically **)
color Test = bool with (FAIL, PASS);

(** Number of links associates the number of links used for a
    particular transaction identifier **)
color Num_Links = product Small_Int*Trans_Id;

(** Mode of operation for an interface **)
color Mode = with impt | expt | idle | link_impt;

```

```

(** basic token used throughout for boolean tests **)
color E = with e;

(** Used by the Offer Space object to store offers for
    multiple Traders.
    Used on page Offer_Space#4 **)
color Offer_Store = product OSInt*Offer_List;

(** Used by the Link Space object to store links for
    multiple Traders.
    Used on page Link_Space#5 **)
color Link_Store = product LSInt*Link_List;

(** Product of an object identifier, and two transaction identifiers
    This is used by the Trader to record a request's incoming
    transaction identifier and the new internal transaction identifier
    for replacement when the request has been processed **)
color TID_Store = product Id*Trans_Id*Trans_Id;

(** List of Transaction Identifiers
    Used by the Trader to hold a list of internal transaction identifiers
    on page Query#15 **)
color TID_List = list Trans_Id;

color Trans_Offers = product Offer_List*Trans_Id;

color Request_Record = product Id*Operation*Trans_Id;
color Matches_Record = product Small_Int*Trans_Id;
color Follow_Record = product Follow_Option*Trans_Id;
color Duplicate_Record = product boolean*Trans_Id;
color Exp_Off_Record = product Op_Elem*Trans_Id;

(** Functions declarations **)

(** Projection functions that extract elements from Message tokens **)
fun S((s,r),dat:Data):Message):Id = s;
fun R((s,r),dat:Data):Message):Id = r;

fun A((addr:Addresses,dat:Data):Message):Addresses = addr;
fun D((addr:Addresses,dat:Data):Message):Data = dat;

fun Op((addr:Addresses,(opn,(op_e,t_id))):Message):Operation = opn;
fun Op_E((addr:Addresses,(opn,(op_e,t_id))):Message):Op_Elem = op_e;
fun Op_D((addr:Addresses,(opn,(op_e,t_id))):Message):Op_Data = (op_e,t_id);
fun T_Id((addr:Addresses,(opn,(op_e,t_id))):Message):Trans_Id = t_id;

(** Projection function to extract the Request ID from an Import Request **)
fun Req_Id((serv:Serv_Type,imp_pol:Imp_Policy):Imp_Req):Request_Id = #request_id(imp_pol);

(** Projection function to extract the Import Policy from an Import Request **)
fun I_Pol((serv:Serv_Type,imp_pol:Imp_Policy):Imp_Req):Imp_Policy = imp_pol;

(** Projection function to extract the Service Interface Identifier and
    Service Type from a matched Offer **)
fun SIID(((int_t:Int_Type,prop_l:Prop_List):Serv_Type,s_int_id:Id):Serv_Off):Id =
s_int_id;

fun
SType(((int_t:Int_Type,prop_l:Prop_List):Serv_Type,s_int_id:Serv_Int_Id):Serv_Off):Serv_Ty
pe =

```

```

(int_t,prop_l);

(** Function used by the Offer Space object to match the
    requested service type with offers in the Offer Space **)
fun M_Off((d,p):Serv_Type,ol:Offer_List):Offer_List =
  if (null(ol)) then
    nil
  else
    let
      val ((desc,prop),intfce)= hd(ol)
    in
      if ((d=desc)andalso(p=prop))then
        ((desc,prop),intfce):M_Off((d,p),tl(ol))
      else
        M_Off((d,p),tl(ol))
    end;

(** Function used by the Offer_Space object to add offers to the
    offer space for a specific Trader.***)
fun Add_Offer((lnk,ol):Offer_Store, new_off:Serv_Off):Offer_Store =
  (lnk,(new_off::ol));

(** Function to set the Transaction identifier of a message to a new value
    Used in the Query#13 Page **)
fun Set_TID((addr:Addresses,(opn,(op_d,t_id)):Message,new_id:Trans_Id):Message =
  (addr,(opn,(op_d,new_id)));

(** Function to join two offer lists.
    Used on Page Query#13 **)
fun J_Off(l1,l2:Offer_List,t_id:Trans_Id):Trans_Offers = (l1^l2,t_id);

(** Function that returns a boolean value regarding whether the r_id
    parameter already exists within the import policy list.
    Used on page Process_Req#14 **)
fun Req_Id_Exists(r_id:Request_Id,r_list:Imp_Policy_List):bool =
  if(null(r_list)) then
    false
  else if(#request_id(hd(r_list))=r_id) then
    true
  else
    Req_Id_Exists(r_id,tl(r_list));

(** Function that searches a list of Import Policies and returns the element
    that matches the import policy parameter.
    Assumes there is a duplicate to match. **)
fun Dup_Match((st,i_pol):Imp_Req,ip_list:Imp_Policy_List,t_id:Trans_Id):Dup_Data =
  if (#request_id(hd(ip_list))=#request_id(i_pol)) then
    (hd(ip_list),(imp)(st,i_pol),t_id)
  else
    Dup_Match((st,i_pol),tl(ip_list),t_id);

(** Function to order the Import_Policies in a list
    Used on Page Process_Req#14 **)

fun OrderIP(iplist:Imp_Policy_List):Imp_Policy_List=
  if (length(iplist)=1) then
    iplist
  else

```

```

let
  val head = #t_ident(#request_id(hd(iplist)))
  val tail = tl(iplist)
in
  if (head < #t_ident(#request_id(hd(tail)))) then
    hd(iplist)::OrderIP(tail)
  else
    hd(tail)::OrderIP(hd(iplist)::tl(tail))
end;

(** This function assumes that a duplicate has been found, and therefore r_list is a
non-empty list **)
fun Update((st,ip):Imp_Req,r_list:Imp_Policy_List):Imp_Policy_List =
  if (#request_id(hd(r_list))=#request_id(ip)) then
    if (#hop_count(hd(r_list))) > (#hop_count(ip)) then
      r_list
    else
      ip::tl(r_list)
  else
    hd(r_list)::Update((st,ip),tl(r_list));

(** Functions used on the Query#13 Page **)

(** Function that compares the ASCII string of the Service Interface Identifier
in two Service Offers and returns the smallest. **)
fun smallest((st1,s_id1):Serv_Off, (st2,s_id2):Serv_Off)=
  if (mkst_col'Id(s_id1) < mkst_col'Id(s_id2)) then
    (st1,s_id1)
  else
    (st2,s_id2);

(** Function that orders the offers in ascending order, thereby ensuring that
queries that result in the same offers have the same marking.
This function could be extended to perform selection on the Offer list **)
fun Select(offers:Offer_List):Offer_List=
  if offers=nil then
    nil
  else if (length(offers)=1) then
    offers
  else
    let
      val head = hd(offers)
      val tail = tl(offers)
    in
      if smallest(head,hd(tail))=head then
        head::Select(tail)
      else
        hd(tail)::Select(head::tl(tail))
    end;

(** Functions for Link Policy Resolution used on Link_Import#15 **)

(** Function that is used to resolve the hop_count parameter.
Used by resolve_policy() **)
fun resolve_num(def:Small_Int, max:Small_Int, input:Small_Int): Small_Int =
  if(input=NO_NUM) then
    def
  else if(input>max) then
    max
  else

```

```

input;

(** Function that finds the smallest follow option based upon the rules in the
Trading Standard (1996) section 8.2.7.6 Link Follow Behaviour **)
fun smallest_option(o1:Follow_Option,o2:Follow_Option,o3:Follow_Option):Follow_Option =
  if ((o1=local_only) orelse (o2=local_only)orelse(o3=local_only))
  then local_only
  else if ((o1=if_no_local) orelse (o2=if_no_local) orelse (o3=if_no_local))
  then if_no_local
  else
  always;

(** Function that calculates the merged policy based upon the Trader, Link and Import
Policies **)
fun merge_policy_options(i_p:Imp_Policy, trad:Import_Attributes,
t_id:Trans_Id):Follow_Record=
  if(#link_follow_rule(i_p)<>no_option) then
    (smallest_option(#max_follow_policy(trad),#link_follow_rule(i_p),always),t_id)
  else
    (smallest_option(#max_follow_policy(trad),#def_follow_policy(trad),always),t_id);

(** Function that calculates the Import Policy which is passed on to Linked Traders **)
fun pass_on_follow_option(i_p:Imp_Policy, trad:Import_Attributes, lnk:Link):Follow_Option=
  if(#link_follow_rule(i_p)<>no_option) then
    smallest_option(#max_follow_policy(trad),#limit_fb(lnk),#link_follow_rule(i_p))
  else
    smallest_option(#max_follow_policy(trad),#def_fb(lnk),always);

(** Function that returns the unified follow behaviour **)
fun unify_link_behaviour(i_p:Imp_Policy, trad:Import_Attributes, lnk:Link):Follow_Option=
  if(#link_follow_rule(i_p)<>no_option) then
    smallest_option(#max_follow_policy(trad),#limit_fb(lnk),#link_follow_rule(i_p))
  else
    smallest_option(#max_follow_policy(trad),#limit_fb(lnk),#def_follow_policy(trad));

(** Function that returns the pass_on import policy **)
fun resolve_pass_on_policy(i_p:Imp_Policy, trad:Import_Attributes, lnk:Link):Imp_Policy=
  {
  hop_count=resolve_num(#def_hop_count(trad),#max_hop_count(trad),#hop_count(i_p))-1,
  link_follow_rule=unify_link_behaviour(i_p,trad,lnk),
  request_id=(#request_id(i_p)),
  exact_match_type=(#exact_match_type(i_p))
  };

(** Function that returns the resolved import policy **)
fun resolve_policy(i_p:Imp_Policy, trad:Import_Attributes, lnk:Link):Imp_Policy=
  {
  hop_count=resolve_num(#def_hop_count(trad),#max_hop_count(trad),#hop_count(i_p)),
  link_follow_rule=unify_link_behaviour(i_p,trad,lnk),
  request_id=(#request_id(i_p)),
  exact_match_type=(#exact_match_type(i_p))
  };

(** Function that creates pass on messages for all linked Traders **)
fun
Create_PO_Mess(ll:Link_List,(st,i_p):Imp_Req,trad:Import_Attributes,mess:Message):Message
ms=
  if (null(ll)) then
    empty
  else
    let

```

```

        val s = S(mess);
        val r = R(mess);
        val t_id = T_Id(mess);
        val new_policy = resolve_pass_on_policy(i_p,trad,hd(l1));
        val test_policy = resolve_policy(i_p,trad,hd(l1));
    in
        if (#link_follow_rule(test_policy)=local_only) orelse (s=(#lui(hd(l1)))) then
            Create_PO_Mess(tl(l1),(st,i_p),trad,mess)
        else

1'((r,#lui(hd(l1))):Addresses,(Op(mess),(imp(st,new_policy):Op_Elem,t_id):Op_Data)):Message)
            + Create_PO_Mess(tl(l1),(st,i_p),trad,mess)

    end;

(** Function that resolves the request's Import Policy and creates a message for all
    linked traders that the request should be forwarded to excluding the trader from
    whom the request was recieved **)
fun
Resolve_Local_ip(l1:Link_List,(st,i_p):Imp_Req,trad:Import_Attributes,mess:Message):Message ms=
    if (null(l1)) then
        empty
    else
        let
            val s = S(mess);
            val r = R(mess);
            val t_id = T_Id(mess);
            val new_policy = resolve_policy(i_p,trad,hd(l1));
        in
            if (#link_follow_rule(new_policy)=local_only) orelse (s=(#lui(hd(l1)))) then
                Resolve_Local_ip(tl(l1),(st,i_p),trad,mess)
            else

1'((r,#lui(hd(l1))):Addresses,(Op(mess),(imp(st,new_policy):Op_Elem,t_id):Op_Data)):Message)
                + Resolve_Local_ip(tl(l1),(st,i_p),trad,mess)

        end;

(** Remove any Links which point back to where the Message originated.
    Assumes that there is only one bidirectional link that points back! **)
fun Bidirectional_Link(l1:Link_List,mess:Message):boolean =
    if (l1=nil) then
        false
    else
        if (#lui(hd(l1)) = S(mess)) then
            true
        else
            Bidirectional_Link(tl(l1),mess)

(** Function to order the Transaction Identifiers in a list.
    Used on Page Query#13 **)
fun Order(idlist:TID_List):TID_List=
    if (length(idlist)=1) then
        idlist
    else
        let

```

```

        val head = hd(idlist)
        val tail = tl(idlist)
    in
        if (head < hd(tail)) then
            head::Order(tail)
        else
            hd(tail)::Order(head::tl(tail))
        end;

(** Functions used in the OEOS Occurrence Graph Creation **)

(** A utility function which maps out the Trans_Id in an Op_Dat **)
fun Mapout_Op_Dat_Trans_Id Op_Dat_ms =
    let
        (* Convert the multi-set into a list *)
        val Op_Dat_list = ms_to_list Op_Dat_ms

        (* MapoutOne removes Trans_Id from a Message *)
        (* using a pattern match *)
        fun MapoutOne (Op_E,_) = Op_E
    in
        (* mapout and convert back to multi-set *)
        list_to_ms (map MapoutOne Op_Dat_list)
    end;

(** A Utility function which maps out the Trans_Id in a Message **)
fun MapoutTrans_Id Mes_ms =
    let
        (* Convert the multi-set into a list *)
        val Mes_list = ms_to_list Mes_ms

        (* MapoutOne removes Trans_Id from a Message *)
        (* using a pattern match *)
        fun MapoutOne (Addr,(Op,(Op_E,_))) = (Addr,(Op,Op_E))
    in
        (* mapout and convert back to multi-set *)
        list_to_ms (map MapoutOne Mes_list)
    end;

(** A utility functions which maps out the Trans_Id in a Request_Record **)
fun MapoutTrans_Id_ReqRec Mes_ms =
    let
        (* Convert the multi-set into a list *)
        val Mes_list = ms_to_list Mes_ms

        (* MapoutOne removes Trans_Id from a Message *)
        (* using a pattern match *)
        fun MapoutOne (Sender,Op,_) = (Sender,Op)
    in
        (* mapout and convert back to multi-set *)
        list_to_ms (map MapoutOne Mes_list)
    end;

(** Declaration of Variables **)

(** Used all over the model when processing messages **)
var mess : Message;
var old_mess : Message;

(** Link_Import #15 **)

```

```

var pass_mess : Message;

var req : Message;
var new_mess : Message;
var result : Test;

(** Offer_Space#4 **)
var o_l : Offer_List;

(** Query#13 **)
var o_l1: Offer_List;
var o_l2: Offer_List;

(** Link_Space#5 & Link_Import#15**)
var l_l : Link_List;

var offer : Serv_Off;
var serv : Serv_Type;
var imp_req : Imp_Req;

(** Used in Export#19 operation to assign an identifier to
    exported offers **)
var exp_id : Small_Int;

var num_l : Small_Int;

var s: Id;
var r: Id;

var r_s: Id;
var r_c: Id;

var s_s: Id;
var s_c: Id;

var data : Data;

var opn : Operation;

var op_d : Op_Data;
var op_e : Op_Elem;

var addr : Addresses;

var a: Ack;

(** Transaction identifiers **)
var t_id : Trans_Id;
var req_tid : Trans_Id;

var tid_l : TID_List;
var exp_id_l : TID_List;

var st: Serv_Type;
var ip : Imp_Policy;
var ip_1 : Imp_Policy;
var ip_2 : Imp_Policy;

var dup : boolean;

var m : Mode;

```

```

var seen : Imp_Policy_List;
var i_r : Imp_Req;

var tr_imp_attrib : Import_Attributes;
var u_pol : Follow_Option;

var num_matches : Small_Int;

```

A.2 OEOS Functions

A.2.1 EquivBE()

```

(** Equivalent Binding Element Function **)

(** Offer Space Transitions **)
fun EquivBE (Bind.Offer_Space'For_Offer_Space (1,{mess = m1}),
            Bind.Offer_Space'For_Offer_Space (1,{mess = m2})) =
    (MapoutTrans_Id(1`m1)=MapoutTrans_Id(1`m2))

| EquivBE (Bind.Offer_Space'Add_Offer (1,{s=s1, r=r1, opn=opn1, op_e=op_e1, t_id =_,
offer=off1, o_l=ol1}),
          Bind.Offer_Space'Add_Offer (1,{s=s2, r=r2, opn=opn2, op_e=op_e2, t_id =_,
offer=off2, o_l =ol2})) =
    ((s1=s2) andalso (r1=r2) andalso (opn1=opn2) andalso (op_e1=op_e2) andalso (ol1=ol2)
andalso (off1 = off2))

| EquivBE (Bind.Offer_Space'Apply_Matching (1,{s=s1, r=r1, opn=opn1, op_e=op_e1, t_id =_,
serv=serv1, o_l=ol1}),
          Bind.Offer_Space'Apply_Matching (1,{s=s2, r=r2, opn=opn2, op_e=op_e2, t_id =_,
serv=serv2, o_l =ol2})) =
    ((s1=s2) andalso (r1=r2) andalso (opn1=opn2) andalso (op_e1=op_e2) andalso (ol1=ol2)
andalso (serv1 = serv2))

| EquivBE (Bind.Offer_Space'Send_Matched_Offers (1,{mess = m1, o_l=ol1, t_id=_}),
          Bind.Offer_Space'Send_Matched_Offers (1,{mess = m2, o_l=ol2, t_id=_})) =
    (MapoutTrans_Id(1`m1)=MapoutTrans_Id(1`m2)) andalso (ol1 = ol2)

(** Link Space Transitions **)
| EquivBE (Bind.Link_Space'Get_Links (1,{s=s1, op_e=op_e1, t_id =_, l_l=l_l1}),
          Bind.Link_Space'Get_Links (1,{s=s2, op_e=op_e2, t_id =_, l_l =l_l2})) =
    ((s1=s2) andalso (op_e1=op_e2) andalso (l_l1=l_l2))

(**
(** Importer Transitions **)
| EquivBE (Bind.Importer'Send_Query (1,{mess = m1}),
          Bind.Importer'Send_Query (1,{mess = m2})) =
    (MapoutTrans_Id(1`m1)=MapoutTrans_Id(1`m2))

| EquivBE (Bind.Importer'Query_Response (1,{mess = m1}),
          Bind.Importer'Query_Response (1,{mess = m2})) =
    (MapoutTrans_Id(1`m1)=MapoutTrans_Id(1`m2))

| EquivBE (Bind.Importer'Invoke_Method (1,{o_l = o_l1, tid_l = _}),

```

```

        Bind.Importer'Invoke_Method (1,{o_l1 = o_l2, tid_l1 = _})) = (o_l1=o_l2)

| EquivBE (Bind.Importer'Service_Ack (1,{mess = m1, tid_l1 = _}),
          Bind.Importer'Service_Ack (1,{mess = m2, tid_l1 = _})) =
(MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2))

(** Exporter does not require the t_id to be mapped out either **)

| EquivBE (Bind.Exporter'Export_Offer (1,{mess = m1}),
          Bind.Exporter'Export_Offer (1,{mess = m2})) =
(MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2))

| EquivBE (Bind.Exporter'Export_Reply (1,{mess = m1}),
          Bind.Exporter'Export_Reply (1,{mess = m2})) =
(MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2))

| EquivBE (Bind.Exporter'Accept_Order (1,{mess = m1}),
          Bind.Exporter'Accept_Order (1,{mess = m2})) =
(MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2))
**)

(** Trad_Int#3 does not contain any transitions that can occur **)

(** Inter-Obj#7 does not contain any transitions that can occur **)

(** OSI Transitions **)
| EquivBE (Bind.OSI'From_Offer_Space (1,{s = s1, r=r1, data=d1}),
          Bind.OSI'From_Offer_Space (1,{s = s2, r=r2, data=d2})) =
          (s1 = s2) andalso (r1=r2) andalso (d1=d2)

| EquivBE (Bind.OSI'To_Offer_Space (1,{s = s1, r=r1, data=d1}),
          Bind.OSI'To_Offer_Space (1,{s = s2, r=r2, data=d2})) =
          (s1 = s2) andalso (r1=r2) andalso (d1=d2)

(** LSI Transitions **)
| EquivBE (Bind.LSI'From_Link_Space (1,{s = s1, r=r1, data=d1}),
          Bind.LSI'From_Link_Space (1,{s = s2, r=r2, data=d2})) =
          (s1 = s2) andalso (r1=r2) andalso (d1=d2)

| EquivBE (Bind.LSI'To_Link_Space (1,{s = s1, r=r1, data=d1}),
          Bind.LSI'To_Link_Space (1,{s = s2, r=r2, data=d2})) =
          (s1 = s2) andalso (r1=r2) andalso (d1=d2)

(** Functional#8 does not contain any transitions that can occur **)

(** LUI Transitions **)
| EquivBE (Bind.LUI'From_LUI (1,{s = s1, r=r1, data=d1}),
          Bind.LUI'From_LUI (1,{s = s2, r=r2, data=d2})) =
          (s1 = s2) andalso (r1=r2) andalso (d1=d2)

| EquivBE (Bind.LUI'To_LUI (1,{s = s1, r=r1, data=d1}),
          Bind.LUI'To_LUI (1,{s = s2, r=r2, data=d2})) =
          (s1 = s2) andalso (r1=r2) andalso (d1=d2)

(** REGI Transitions **)
| EquivBE (Bind.REGI'From_REGI (1,{s = s1, r=r1, data=d1}),
          Bind.REGI'From_REGI (1,{s = s2, r=r2, data=d2})) =
          (s1 = s2) andalso (r1=r2) andalso (d1=d2)

| EquivBE (Bind.REGI'To_REGI (1,{s = s1, r=r1, data=d1}),
          Bind.REGI'To_REGI (1,{s = s2, r=r2, data=d2})) =

```

```

(s1 = s2) andalso (r1=r2) andalso (d1=d2)

(** Functional Operations **)
| EquivBE (Bind.Funct_Opns'Query_Message (1,{mess = m1}),
          Bind.Funct_Opns'Query_Message (1,{mess = m2})) =
          (MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2))

| EquivBE (Bind.Funct_Opns'New_Lookup_Request (1,{mess = m1}),
          Bind.Funct_Opns'New_Lookup_Request (1,{mess = m2})) =
          (MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2))

| EquivBE (Bind.Funct_Opns'Export_Message (1,{mess = m1}),
          Bind.Funct_Opns'Export_Message (1,{mess = m2})) =
          (MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2))

| EquivBE (Bind.Funct_Opns'New_Export_Request (1,{mess = m1}),
          Bind.Funct_Opns'New_Export_Request (1,{mess = m2})) =
          (MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2))

(** Export Operation Transitions **)
| EquivBE (Bind.Export'Export_Operation (1,{mess = m1, tid_l=t_l1}),
          Bind.Export'Export_Operation (1,{mess = m2, tid_l=t_l2})) =
          (MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2) andalso (t_l1=t_l2))

| EquivBE (Bind.Export'From_Offer_Space (1,{mess = m1}),
          Bind.Export'From_Offer_Space (1,{mess = m2})) =
          MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2)

| EquivBE (Bind.Export'Return_Result (1,{a=a1, old_mess = m1, tid_l=t_l1, exp_id_l =
exp_l1, t_id= tid1 }),
          Bind.Export'Return_Result (1,{a = a2, old_mess = m2, tid_l=t_l2,
exp_id_l=exp_l2, t_id = tid2})) =
          (MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2) andalso (t_l1=t_l2))

(** Query Operation Transitions **)
| EquivBE (Bind.Query'Check_Parameters (1,{mess = m1}),
          Bind.Query'Check_Parameters (1,{mess = m2})) =
          (MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2))

| EquivBE (Bind.Query'Check_Security (1,{mess = m1, tid_l =_}),
          Bind.Query'Check_Security (1,{mess = m2, tid_l =_})) =
          (MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2))

| EquivBE (Bind.Query'Accept_Trader_Links (1,{mess = m1}),
          Bind.Query'Accept_Trader_Links (1,{mess = m2})) =
          (MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2))

| EquivBE (Bind.Query'Offers_from_Linked (1,{mess = m1}),
          Bind.Query'Offers_from_Linked (1,{mess = m2})) =
          (MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2))

| EquivBE (Bind.Query'Add_Link_Offer (1,{o_l1=o_l1_1, o_l2=o_l2_1, num_l=n_l1, t_id =_}),
          Bind.Query'Add_Link_Offer (1,{o_l1=o_l1_2, o_l2=o_l2_2, num_l=n_l2, t_id =_})) =
          (o_l1_1=o_l1_2) andalso (o_l2_1=o_l2_2) andalso (n_l1=n_l2)

| EquivBE (Bind.Query'Local_Offers (1,{mess = m1}),
          Bind.Query'Local_Offers (1,{mess = m2})) =
          (MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2))

| EquivBE (Bind.Query'Add_Local_Offer (1,{o_l1 = o_l1_1, o_l2=o_l2_1, t_id =_}),
          Bind.Query'Add_Local_Offer (1,{o_l1 = o_l1_2, o_l2=o_l2_2, t_id =_})) =

```

```

        (ol1_1=ol1_2) andalso (ol2_1=ol2_2)

| EquivBE (Bind.Query'Apply_Selection_Constraints (1,{s=s1, r=r1, opn=opn1, op_e=op_e1,
o_1=ol1, num_l=n1, t_id=_}),
        Bind.Query'Apply_Selection_Constraints (1,{s=s2, r=r2, opn=opn2, op_e=op_e2,
o_1=ol2, num_l=n2, t_id=_})) =
        (s1=s2) andalso (r1=r2) andalso (opn1=opn2) andalso (op_e1=op_e2) andalso
(ol1=ol2) andalso (n1=n2)

| EquivBE (Bind.Query'Return_T_Id (1,{r=r1, mess = m1, req_tid =_, t_id =_, tid_l=_}),
        Bind.Query'Return_T_Id (1,{r=r2, mess = m2, req_tid =_, t_id =_, tid_l=_})) =
        (MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2)) andalso(r1=r2)

(** Transitions on Process_Req#14**)
| EquivBE (Bind.Process_Req'Is_Duplicate (1,{seen = s1, i_r = ir1, t_id =_}),
        Bind.Process_Req'Is_Duplicate (1,{seen = s2, i_r = ir2, t_id =_})) =
        (s1=s2) andalso (ir1=ir2)

| EquivBE (Bind.Process_Req'Is_Unique (1,{seen = s1, st=st1, ip=ip1, t_id =_}),
        Bind.Process_Req'Is_Unique (1,{seen = s2, st=st2, ip=ip2, t_id =_})) =
        (s1=s2) andalso (st1=st2) andalso (ip1=ip2)

| EquivBE (Bind.Process_Req'No_Link_Search_Reqd (1,{st=st1, ip=ip1, ip_l=old_ip1, mess=m1,
t_id=_}),
        Bind.Process_Req'No_Link_Search_Reqd (1,{st=st2, ip=ip2, ip_l=old_ip2, mess=m2,
t_id=_})) =
        (st1=st2) andalso (ip1=ip2) andalso
(MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2)) andalso (old_ip1=old_ip2)

| EquivBE (Bind.Process_Req'Link_Search_Reqd (1,{st = st1, ip = ip1, ip_l=old_ip1, t_id
=_}),
        Bind.Process_Req'Link_Search_Reqd (1,{st = st2, ip = ip2, ip_l=old_ip2, t_id
=_})) =
        (st1=st2) andalso (ip1=ip2) andalso (old_ip1=old_ip2)

| EquivBE (Bind.Process_Req'Unify_Policy (1,{st = st1, ip = ip1, tr_imp_attrib = tia1,
t_id =_}),
        Bind.Process_Req'Unify_Policy (1,{st = st2, ip = ip2, tr_imp_attrib = tia2,
t_id =_})) =
        (st1=st2) andalso (ip1=ip2) andalso (tia1=tia2)

| EquivBE (Bind.Process_Req'Local_Search_Only (1,{st=st1, ip=ip1, u_pol=upol1, mess=m1,
dup=dup1, t_id=_}),
        Bind.Process_Req'Local_Search_Only (1,{st=st2, ip=ip2, u_pol=upol2, mess=m2,
dup=dup2, t_id=_})) =
        (st1=st2) andalso (ip1=ip2) andalso (upol1=upol2) andalso
(MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2))

| EquivBE (Bind.Process_Req'Link_andor_Local_Search (1,{st=st1, ip=ip1, u_pol=upol1,
dup=dup1, t_id=_}),
        Bind.Process_Req'Link_andor_Local_Search (1,{st=st2, ip=ip2, u_pol=upol2,
dup=dup2, t_id=_})) =
        (st1=st2) andalso (ip1=ip2) andalso (upol1=upol2) andalso (dup1=dup2)

(** Transitions on the Import Links Page **)
| EquivBE (Bind.Link_Import'Process_Links (1,{tr_imp_attrib=tia1, st=st1, ip=ip1,
l_l=link1, mess=m1, t_id=_}),
        Bind.Link_Import'Process_Links (1,{tr_imp_attrib=tia2, st=st2, ip=ip2,
l_l=link2, mess=m2, t_id=_})) =
        (tia1=tia2) andalso (st1=st2) andalso (ip1=ip2) andalso (link1=link2)
andalso (MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2))

```

```

| EquivBE (Bind.Link_Import'Import_From_Link (1,{addr=add1, opn=opn1, st=st1, ip_1=ip11,
ip_2=ip21, t_id=_}),
          Bind.Link_Import'Import_From_Link (1,{addr=add2, opn=opn2, st=st2, ip_1=ip12,
ip_2=ip22, t_id=_})) =
          (add1=add2) andalso (opn1=opn2) andalso (st1=st2) andalso (ip11=ip12)
andalso (ip21=ip22)

| EquivBE (Bind.Link_Import'Select_Always (1,{addr=add1, opn=opn1, st=st1, ip=ip1,
t_id=_}),
          Bind.Link_Import'Select_Always (1,{addr=add2, opn=opn2, st=st2, ip=ip2,
t_id=_})) =
          (add1=add2) andalso (opn1=opn2) andalso (st1=st2) andalso (ip1=ip2)

| EquivBE (Bind.Link_Import'Select_If_No_Local (1,{addr=add1, opn=opn1, st=st1, ip=ip1,
t_id=_}),
          Bind.Link_Import'Select_If_No_Local (1,{addr=add2, opn=opn2, st=st2, ip=ip2,
t_id=_})) =
          (add1=add2) andalso (opn1=opn2) andalso (st1=st2) andalso (ip1=ip2)

| EquivBE (Bind.Link_Import'Send_If_No_Local (1,{mess=m1, pass_mess=pm1, num_l=n11,
num_matches=nm1, t_id=_}),
          Bind.Link_Import'Send_If_No_Local (1,{mess=m2,pass_mess=pm2, num_l=n12,
num_matches=nm2, t_id=_})) =
          (MapoutTrans_Id(1'm1)=MapoutTrans_Id(1'm2)) andalso
(MapoutTrans_Id(1'pm1)=MapoutTrans_Id(1'pm2)) andalso (nm1=nm2) andalso (n11=n12)

| EquivBE(_,_) = false;

```

A.2.2 EquivMark()

```

fun EquivMark (n1,n2) =

(** Marking on the Trad_Env Top Level Page **)
MapoutTrans_Id(Mark.Trad_Env'Comms_Medium 1 n1) ==
MapoutTrans_Id(Mark.Trad_Env'Comms_Medium 1 n2) andalso

(** The Link Space Object does not contain any places who marking changes
since there is no add_links() method **)

(** Marking on the Offer Space Object **)
MapoutTrans_Id(Mark.Offer_Space'Request 1 n1) ==
MapoutTrans_Id(Mark.Offer_Space'Request 1 n2) andalso

MapoutTrans_Id(Mark.Offer_Space'Request_Store 1 n1) ==
MapoutTrans_Id(Mark.Offer_Space'Request_Store 1 n2) andalso

Mapout_Op_Dat_Trans_Id(Mark.Offer_Space'Match_Data 1 n2) andalso
apout_Op_Dat_Trans_Id(Mark.Offer_Space'Match_Data 1 n1) ==

Mapout_Op_Dat_Trans_Id(Mark.Offer_Space'Matched_Offer 1 n1) ==
Mapout_Op_Dat_Trans_Id(Mark.Offer_Space'Matched_Offer 1 n2) andalso

(Mark.Offer_Space'Offer_Storage 1 n1) ==
(Mark.Offer_Space'Offer_Storage 1 n2) andalso

(Mark.Offer_Space'Semaphore 1 n1) ==
(Mark.Offer_Space'Semaphore 1 n2) andalso

(** Marking on the Importer **)
(** Don't map the t_id out siince it *IS* important for the Importer to know it **)

```

```

(Mark.Importer'Import_Request 1 n1) == (Mark.Importer'Import_Request 1 n2) andalso

(Mark.Importer'Matched_Offers 1 n1) == (Mark.Importer'Matched_Offers 1 n2) andalso
(Mark.Importer'Service_Completed 1 n1) == (Mark.Importer'Service_Completed 1 n2) andalso
(Mark.Importer'Ready_to_order 1 n1) == (Mark.Importer'Ready_to_order 1 n2) andalso

(** Markings on the Exporter **)
(** Don't map the t_id out siince it *IS* important for the Exporter to know it **)
(Mark.Exporter'Offers_to_Export 1 n1) == (Mark.Exporter'Offers_to_Export 1 n2) andalso

(** Remove the trans_id from the exported service handle **)
(Mark.Exporter'Offer_Identifiers 1 n1) == (Mark.Exporter'Offer_Identifiers 1 n2) andalso
(Mark.Exporter'Pending_Order 1 n1) == (Mark.Exporter'Pending_Order 1 n2) andalso

(** Marking on the Trader and Interfaces Page **)
MapoutTrans_Id(Mark.Trad_Int'To_Trader 1 n1) ==
MapoutTrans_Id(Mark.Trad_Int'To_Trader 1 n2) andalso

MapoutTrans_Id(Mark.Trad_Int'To_Interfaces 1 n1) ==
MapoutTrans_Id(Mark.Trad_Int'To_Interfaces 1 n2) andalso

(Mark.Trad_Int'Trader_Imp_Attrib 1 n1) ==
(Mark.Trad_Int'Trader_Imp_Attrib 1 n2) andalso

(** Markings on the Inter-Obj Interfaces **)
(** None required since the places have already been included**)

(** Markings on the OSI and LSI **)
(Mark.OSI'OSI_Int_Id 1 n1) ==
(Mark.OSI'OSI_Int_Id 1 n2) andalso

(Mark.LSI'LSI_Int_Id 1 n1) ==
(Mark.LSI'LSI_Int_Id 1 n2) andalso

(** Markings on the Functional Interfaces **)
(** None required since the places have already been included**)

(** Markings on the LUI and REGI **)
(Mark.LUI'LUI_Id 1 n1) ==
(Mark.LUI'LUI_Id 1 n2) andalso

(Mark.REGI'REGI_Id 1 n1) ==
(Mark.REGI'REGI_Id 1 n2) andalso

(** Marking on the Functional Operations **)
MapoutTrans_Id(Mark.Funct_Opns'Query_Operation 1 n1) ==
MapoutTrans_Id(Mark.Funct_Opns'Query_Operation 1 n2) andalso

MapoutTrans_Id(Mark.Funct_Opns'Export_Operation 1 n1) ==
MapoutTrans_Id(Mark.Funct_Opns'Export_Operation 1 n2) andalso

(** Marking on the Query Operation Page **)
MapoutTrans_Id(Mark.Query'Param_Check_Passed 1 n1) ==
MapoutTrans_Id(Mark.Query'Param_Check_Passed 1 n2) andalso

MapoutTrans_Id(Mark.Query'Mess_Store_1 1 n1) ==
MapoutTrans_Id(Mark.Query'Mess_Store_1 1 n2) andalso

MapoutTrans_Id(Mark.Query'Mess_Store_2 1 n1) ==
MapoutTrans_Id(Mark.Query'Mess_Store_2 1 n2) andalso

```

```

MapoutTrans_Id(Mark.Query'End_Of_Transaction 1 n1) ==
MapoutTrans_Id(Mark.Query'End_Of_Transaction 1 n2) andalso

(** Do not consider place Request_T_IDs since it deals only in t_id logic **)

(** Op_Dat places use the Mapout_Op_Dat_Trans_Id() function **)
Mapout_Op_Dat_Trans_Id(Mark.Query'Security_Check_Passed 1 n1) ==
Mapout_Op_Dat_Trans_Id(Mark.Query'Security_Check_Passed 1 n2) andalso

Mapout_Op_Dat_Trans_Id(Mark.Query'Links_to_Search 1 n1) ==
Mapout_Op_Dat_Trans_Id(Mark.Query'Links_to_Search 1 n2) andalso

Mapout_Op_Dat_Trans_Id(Mark.Query'Import_Request 1 n1) ==
Mapout_Op_Dat_Trans_Id(Mark.Query'Import_Request 1 n2) andalso

Mapout_Op_Dat_Trans_Id(Mark.Query'Num_Local_Matches 1 n1) ==
Mapout_Op_Dat_Trans_Id(Mark.Query'Num_Local_Matches 1 n2) andalso

Mapout_Op_Dat_Trans_Id(Mark.Query'Link_Offers 1 n1) ==
Mapout_Op_Dat_Trans_Id(Mark.Query'Link_Offers 1 n2) andalso

Mapout_Op_Dat_Trans_Id(Mark.Query'Local_Offers 1 n1) ==
Mapout_Op_Dat_Trans_Id(Mark.Query'Local_Offers 1 n2) andalso

Mapout_Op_Dat_Trans_Id(Mark.Query'Number_Of_Links 1 n1) ==
Mapout_Op_Dat_Trans_Id(Mark.Query'Number_Of_Links 1 n2) andalso

Mapout_Op_Dat_Trans_Id(Mark.Query'Remote_Offers 1 n1) ==
Mapout_Op_Dat_Trans_Id(Mark.Query'Remote_Offers 1 n2) andalso

(** Process Request page **)
(Mark.Process_Req'IWT_Request_Store 1 n1) == (Mark.Process_Req'IWT_Request_Store 1 n2)
andalso

MapoutTrans_Id_ReqRec(Mark.Process_Req'Duplicate_Message 1 n1) ==
MapoutTrans_Id_ReqRec(Mark.Process_Req'Duplicate_Message 1 n2) andalso

Mapout_Op_Dat_Trans_Id(Mark.Process_Req'Process_Message 1 n1) ==
Mapout_Op_Dat_Trans_Id(Mark.Process_Req'Process_Message 1 n2) andalso

Mapout_Op_Dat_Trans_Id(Mark.Process_Req'Import_Request_Store 1 n1) ==
Mapout_Op_Dat_Trans_Id(Mark.Process_Req'Import_Request_Store 1 n2) andalso

Mapout_Op_Dat_Trans_Id(Mark.Process_Req'Reduced_Policy 1 n1) ==
Mapout_Op_Dat_Trans_Id(Mark.Process_Req'Reduced_Policy 1 n2) andalso

Mapout_Op_Dat_Trans_Id(Mark.Process_Req'Processing_Duplicate 1 n1) ==
Mapout_Op_Dat_Trans_Id(Mark.Process_Req'Processing_Duplicate 1 n2) andalso

(** Import Linked Traders page **)
MapoutTrans_Id(Mark.Link_Import'Link_Pass_On_Messages 1 n1) ==
MapoutTrans_Id(Mark.Link_Import'Link_Pass_On_Messages 1 n2) andalso

MapoutTrans_Id(Mark.Link_Import'Link_Messages 1 n1) ==
MapoutTrans_Id(Mark.Link_Import'Link_Messages 1 n2) andalso

MapoutTrans_Id(Mark.Link_Import'Link_Searches_to_Reduce 1 n1) ==
MapoutTrans_Id(Mark.Link_Import'Link_Searches_to_Reduce 1 n2) andalso

MapoutTrans_Id(Mark.Link_Import'Requests_INLocal 1 n1) ==
MapoutTrans_Id(Mark.Link_Import'Requests_INLocal 1 n2) andalso

```

```
(** Export Operation page *)
MapoutTrans_Id(Mark.Export'Request 1 n1) ==
MapoutTrans_Id(Mark.Export'Request 1 n2) andalso

Mapout_Op_Dat_Trans_Id(Mark.Export'Match_Data 1 n1) ==
Mapout_Op_Dat_Trans_Id(Mark.Export'Match_Data 1 n2)
```

A.3 Occurrence Graph Functions

```
val dead = ListDeadMarkings();
length(dead);

OGSet.StopOptions{
Nodes=0,
Arcs=0,
Secs=600,
Predicate = fn _ => false};

OGSet.BranchingOptions{
TransInsts=0,
Bindings=0,
Predicate = fn _ => true};

OESet.Equivalence {
Mark=EquivMark
Bind=EquivBE,
Spce=OESpec};

CalculateOEGraph();
```

Appendix B

Java Source code: Trader Implementation

This Appendix includes the Java source code for an Implementation of the Trader. There are a number of classes defined and all the source is provided. In addition, a class hierarchy is provided, which is a graphical description of the relationships of classes within the Trader.

B.1 Interfaces

B.1.1 Lookup.java

```
import java.util.Vector;
import java.rmi.*;

/**
 * Lookup interface is used by the Trader to provide
 * clients with an interface for Querying its Offer Space.
 *
 * @version 1.00
 * @author Andrew Tokmakoff
 */
public interface Lookup extends java.rmi.Remote
{
    /**
     * Method that allows clients to query the offer space.
     *
     * @param originator the name of the object performing the query.
     * @param serv the service type that needs to be matched.
     * @param ip the import policy associated with the request.
     * @return A vector of matching services.
     */
    Vector query(String originator, serviceRecord serv, impPolicyRecord ip) throws
    java.rmi.RemoteException;
}
```

B.1.2 Register.java

```
import java.util.Vector;
import java.rmi.*;

/**
```

```

* Register interface is used by the Trader to provide
* clients with an interface for adding offers to its Offer Space.
*
* @version 1.00
* @author Andrew Tokmakoff
*/
public interface Register extends java.rmi.Remote
{
    /**
     * Method that allows clients to add offers to the offer space.
     *
     * @param serviceRecord service to be exported.
     * @return An integer representing the unique offer transaction
     *         identifier. -1 if the method failed.
     */
    int export(serviceRecord serv) throws java.rmi.RemoteException;
}

```

B.1.3 Trader.java

```

import java.util.Vector;
import java.rmi.*;

/**
 * Interface definition for the Trader Object.
 * Extends the Register and Lookup interfaces.
 *
 * @author Andrew Tokmakoff
 * @version 1.0
 */
public interface Trader extends Lookup,
                                Register
{
}

```

B.1.4 LinkSpace.java

```

import java.util.Vector;
import java.rmi.*;

/**
 * Interface definition for the Link Space Object
 *
 * @author Andrew Tokmakoff
 * @version 1.00
 */
public interface LinkSpace extends java.rmi.Remote
{
    /**
     * Function used for testing of RMI functionality
     */
    String sayHello() throws java.rmi.RemoteException;

    /**
     * A Function that allows a Trader to retrieve links from
     * the link Space.
     *
     * @param trader The Trader whose link space is to modified.
     */
    Vector getLinks(String trader) throws java.rmi.RemoteException;

    /**
     * Adds a link to a Trader's Link Space.
     *
     * @param trader The Trader whose link space is to modified.
     * @param link Link record to be added to the link space.
     */
}

```

```

    */
void    addLink(String trader, linkRecord link) throws java.rmi.RemoteException;

/**
 * Routine to remove a Link from the Link Space.
 *
 * @param trader The Trader whose Link space is to modified.
 * @param linkName Name of the Link to be removed.
 */
void    removeLink(String trader, String linkName) throws
java.rmi.RemoteException;

/**
 * Routine to print the Link Space to the screen.
 *
 * @param trader Trader whose Links are to be printed.
 */
void    printLinks(String trader) throws java.rmi.RemoteException;
}

```

B.1.5 OfferSpace.java

```

import java.util.Vector;
import java.rmi.*;

/**
 * Interface definition for the OfferSpace Object
 *
 * @author Andrew Tokmakoff
 * @version 1.00
 */
public interface OfferSpace extends java.rmi.Remote
{
    /**
     * Function used for testing of RMI functionality
     */
    String sayHello() throws java.rmi.RemoteException;

    /**
     * A Function that allows a Trader to retrieve offers from
     * the Offer Space.
     *
     * @param trader The Trader whose offer space is to modified.
     * @param serv the service that is being added to the offer space.
     */
    Vector getOffers(String trader, serviceRecord serv) throws
java.rmi.RemoteException;

    /**
     * Adds an offer to a Trader's Offer Space.
     *
     * @param trader The Trader whose offer space is to modified.
     * @param offer the offer to be added to the offer space.
     */
    int addOffer(String trader, offerRecord offer) throws java.rmi.RemoteException;

    /**
     * Routine to remove an offer from the Offer Space.
     *
     * @param trader The Trader whose offer space is to modified.
     * @param offerId Identifier of the offer to be removed.
     */
    void removeOffer(String trader, int offerId) throws java.rmi.RemoteException;

    /**
     * Routine to print the Offer Space to the screen.

```

```

*
* @param trader Trader whose offers are to be printed.
**/
void printOffers(String trader) throws java.rmi.RemoteException;

/**
* Routine to initialise the Offer Space from a test file.
*
* @param traderName Name of the Trader whose offer space is being initialised.
**/
void initOfferSpace(String traderName) throws java.rmi.RemoteException;
}

```

B.2 Abstract Data Types

B.2.1 impPolicyRecord.java

```

import java.io.*;

/**
*
* Import Policy Record.
* Used to store Import Policy Information
*
* @author Andrew Tokmakoff
* @version 1.00
*/
public class impPolicyRecord implements Serializable
{
    /**
    * The limiting hopCount for the request
    */
    public int hopCount;

    /**
    * Follow behaviour for links
    */
    public String linkFollowRule;

    /**
    * Request identifier record
    */
    public reqIdRecord requestId;

    /**
    * Boolean condition for exact matching on type
    */
    public boolean exactMatchType;

    /**
    * Constructor
    *
    * @param hc The limiting hopCount for the request.
    * @param lfr Follow behaviour for links
    * @param rid Request identifier record
    * @param emt Boolean condition for exact matching on type
    */
    public impPolicyRecord(int hc, String lfr, reqIdRecord rid, boolean emt)
    {
        hopCount = hc;
        linkFollowRule = lfr;
        requestId = rid;
        exactMatchType = emt;
    }
}

```

```

    }

    /**
     * Determines if two ImpPolicyRecords are equal
     *
     * @param other the impPolicyRecord for comparison.
     * @return String that represents the object's internal state.
     */
    public boolean equals(impPolicyRecord other)
    {
        if ((hopCount == other.hopCount) &&
            (linkFollowRule.equals(other.linkFollowRule)) &&
            (requestId.equals(other.requestId)) &&
            (exactMatchType == other.exactMatchType))
            return(true);
        else
            return(false);
    }

    /**
     * Converts the class to a String representation.
     * @return String that represents the object's internal state.
     */
    public String toString()
    {
        return(hopCount+" "+linkFollowRule+" "+requestId+" "+exactMatchType);
    }
}

```

B.2.2 linkRecord.java

```

import java.io.*;

/**
 *
 * Link Record.
 * Used to store Link information
 *
 * @author Andrew Tokmakoff
 * @version 1.00
 */
public class linkRecord implements Serializable
{
    /**
     * String Identifier of the link
     */
    private String name;

    /**
     * name of the Lookup Interface of the linked Trader
     * (Lookup is an Interface)
     * Could change this to an RMI Object reference (Lookup).
     */
    private String luiName;

    /**
     * name of the Register Interface of the linked Trader
     * (Lookup is an Interface)
     * Could change this to an RMI Object reference (Register).
     */
    // private String registerInterface

    /**
     * Followup Rules for Importing
     * May only take values:
     * local_only,

```

```

*   if_no_local or
*   always.
*
* String Identifier of the link
*/

private String defaultFollowRule;
private String limitingFollowRule;

/**
 * Constructor
 */
linkRecord()
{
}

/**
 * Constructor
 *
 * @param n Name of the Link.
 * @param luInt Lookup interface of the Trader that is being linked to.
 * @param defFR Default follow rule for the Link.
 * @param limFR Limiting follow rule for the Link.
 */
linkRecord(String n, String luInt, String defFR, String limFR)
{
    setName(n);
    setLUI(luInt);
    setDFR(defFR);
    setLFR(limFR);
}

/**
 * Accessor method
 * @return Name of the Link.
 */
public String getName()
{
    return(name);
}

/**
 * Mutator method
 * @param Name of the Link.
 */
public void setName(String n)
{
    name = n;
}

/**
 * Accessor method
 * @return Lookup interface.
 */
public String getLUI()
{
    return(luiName);
}

/**
 * Mutator method
 * @param Lookup Interface.
 */
public void setLUI(String i)
{
    luiName = i;
}

```

```

    }

    /**
     * Accessor method
     * @return Default follow rule.
     */
    public String getDFR()
    {
        return(defaultFollowRule);
    }

    /**
     * Mutator method
     * @param Default follow rule.
     */
    public void setDFR(String dfr)
    {
        defaultFollowRule = dfr;
    }

    /**
     * Accessor method
     * @return Limiting follow rule.
     */
    public String getLFR()
    {
        return(limitingFollowRule);
    }

    /**
     * Mutator method
     * @param Limiting follow rule.
     */
    public void setLFR(String lfr)
    {
        limitingFollowRule = lfr;
    }

    /**
     * Converts the class fields into a String for printing.
     */
    public String toString()
    {
        return(name+" "+luiName+" "+defaultFollowRule
            +" "+limitingFollowRule);
    }
}

```

B.2.3 offerRecord.java

```

import java.util.*;

/**
 * Class used for storing offer Information in the Offer Space object.
 *
 * @author Andrew Tokmakoff
 * @version 1.00
 */
public class offerRecord extends serviceRecord
{
    /**
     * The name of the trader associated with the Offer.
     */
    private String trader;

    /**
     * The offer Identifier associated with the Offer.

```

```

    */
private int offerIdent;

/**
 * Default constructor.
 */
public offerRecord()
{
}

/**
 * Initialising constructor.
 *
 * @param t Name of the Trader where the offer is located.
 * @param s Service name.
 * @param p Service properties.
 * @param l Service location.
 */
public offerRecord(String t, String s, servProps p, String l)
{
    super(s,p,l);
    setTrader(t);
    offerIdent = -1;
}

/**
 * Mutator method
 */
public void setTrader(String t)
{
    trader=t;
}

/**
 * Accessor method
 */
public String getTrader()
{
    return(trader);
}

/**
 * Mutator method
 */
public void setOfferIdent(int num)
{
    offerIdent = num;
}

/**
 * Accessor method
 */
public int getOfferIdent()
{
    return(offerIdent);
}

/**
 * Converts the class fields into a String for printing.
 */
public String toString()
{
    return(trader+" "+serviceType+" "+location+" "+properties+" "+offerIdent);
}
}

```

B.2.4 reqIdRecord.java

```
import java.io.*;

/**
 * Used to store Request Id Information.
 * It is used to record requests that are processed
 * so the Trader can check incoming requests to detect duplicate
 * messages.
 *
 * @author Andrew Tokmakoff
 * @version 1.00
 */
public class reqIdRecord implements Serializable
{
    /**
     * The source of the request
     */
    public String source;
    /**
     * Transaction identifier associated with the request.
     */
    public int tIdent;

    /**
     * Constructor
     *
     * @param s The source
     * @param t transaction identifier
     */
    public reqIdRecord(String s, int t)
    {
        source = s;
        tIdent = t;
    }

    /**
     * Determines if two reqIdRecords are equal
     *
     * @param other the reqIdRecord for comparison.
     * @return String that represents the object's internal state.
     */
    public boolean equals(reqIdRecord other)
    {
        if ((source.equals(other.source)) &&
            (tIdent == other.tIdent))
            return(true);
        else
            return(false);
    }

    /**
     * Converts the class to a String representation.
     *
     * @return String that represents the object's internal state.
     */
    public String toString()
    {
        return(source+" "+tIdent);
    }
}
```

B.2.5 linkStore.java

```
import java.util.Vector;

/**
 * A class which maintains links for a specific Trader.
 *
 * @author Andrew Tokmakoff
 * @version 1.00
 */
public class linkStore
{
    /**
     * Public member variables no need for accessor/mutator
     * methods - could add later if required.
     */

    /**
     * Name of the Trader whose links are stored.
     */
    public String trader;
    /**
     * A vector that contains the Trader's linkRecords.
     */
    public Vector links;

    /**
     * Constructor.
     *
     * @param t Trader name.
     */
    public linkStore(String t)
    {
        trader = t;
        links = new Vector(2,1);
    }

    /**
     * Adds a linkRecord to the Vector of links.
     *
     * @param t Trader name.
     */
    public void addLink(linkRecord lr)
    {
        links.addElement(lr);
    }

    /**
     * Removes a linkRecord from the Vector of links.
     *
     * @param n Link name.
     */
    public void removeLink(String n)
    {
        int i;

        for (i=0; i<links.size(); i++)
        {
            linkRecord temp = (linkRecord)links.elementAt(i);
            if(temp.getName().equals(n))
            {
                links.removeElementAt(i);
            }
        }
    }
}

/**
```

```

    * Converts the class to a String representation.
    *
    * @return String that represents the object's internal state.
    */
public String toString()
{
    int i;
    String out = trader+'\n';

    for (i=0; i<links.size(); i++)
    {
        out +=(" "+links.elementAt(i)+'\n');
    }
    return(out);
}
}

```

B.2.6 offerStore.java

```

import java.util.Vector;

/**
 * Used by the Offer Space Object to maintain offers for different
 * Traders.
 *
 * @author Andrew Tokmakoff
 * @version 1.00
 */
public class offerStore
{
    /**
     * Public member variables no need for accessor/mutator
     * methods - could add later if required.
     */

    /**
     * Name of the Trader whose offers are being stored.
     */
    public String trader;

    /**
     * Vector of Offer Records that contains the offers.
     */
    public Vector offers;

    /**
     * Constructor
     *
     * @param t Name of the Trader whose offers are being stored.
     */
    public offerStore(String t)
    {
        trader = t;
        offers = new Vector(2,1);
    }

    /**
     * Adds an Offer to the Offer Record.
     *
     * @param or Offer Record which is being updated.
     */
    void addOffer(offerRecord or)
    {
        offers.addElement(or);
    }
}

```

```

    * Removes an Offer from an Offer Record.
    *
    * @param num Offer identifier of the Record which is being removed.
    */
void removeOffer(int num)
{
    int i;

    for (i=0; i<offers.size(); i++)
    {
        offerRecord temp = (offerRecord)offers.elementAt(i);
        if(temp.getOfferIdent()==num)
        {
            offers.removeElementAt(i);
        }
    }
}

/**
 * Converts the class fields into a string for display.
 */
public String toString()
{
    int i;
    String out = trader+'\n';

    for (i=0; i<offers.size(); i++)
    {
        out +=(" "+offers.elementAt(i)+'\n');
    }
    return(out);
}
}

```

B.2.7 traderPolicyRecord.java

```

import java.io.*;

/**
 * Class used by the Trader for storing Trader Policy Information.
 * This information governs the Trader behaviour.
 *
 * @author Andrew Tokmakoff
 * @version 1.00
 */
public class traderPolicyRecord implements Serializable
{
    /**
     * Public member variables, therefore no accessor
     * or mutator functions required.
     */

    /**
     * default hop count for trader Query processing.
     */
    public int defHopCount;
    /**
     * maximum hop count for trader Query processing.
     */
    public int maxHopCount;
    /**
     * Stem used to identify unique Traders.
     */
    public String requestIdStem;
    /**
     * default follow rule for trader Query processing.

```

```

    */
    public String defFollowRule;
    /**
     * maximum follow rule for trader Query processing.
     */
    public String maxFollowRule;

    /**
     * Constructor.
     * @param dhc default hop count
     * @param mhc maximum hop count
     * @param rids Request Id Stem
     * @param dfr Default follow rule
     * @param mfr Maximum follow rule
     */
    public traderPolicyRecord(int dhc, int mhc, String rids, String dfr, String mfr)
    {
        defHopCount = dhc;
        maxHopCount = mhc;
        requestIdStem = rids;
        defFollowRule = dfr;
        maxFollowRule = mfr;
    }
}

```

B.3 Object implementations

B.3.1 Trader_impl.java

```

import java.io.*;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
import java.util.Random;

/**
 * A class which performs the high level Trading function.
 *
 * @author Andrew Tokmakoff
 * @version 1.00
 */
public class Trader_impl
    extends UnicastRemoteObject
    implements Trader
        // implements Register
        // implements Link
{
    /**
     * The name of this Trader
     */
    private static String name;
    /**
     * The name of this Trader's Lookup Interface
     */
    private static String lui;
    /**
     * The name of this Trader's Register Interface
     */
    private static String regi;
    /**
     * The Traders' trading policy
     */
    private traderPolicyRecord tradPolicy;

```

```

/**
 * Object reference to the linkSpace object
 */
private LinkSpace linkSpace;
/**
 * Object reference to the offerSpace object
 */
private OfferSpace offerSpace;
/**
 * Vector of requests that have been seen by this Trader
 */
private Vector reqSeen;
/**
 * Boolean that flags debugging output
 */
private boolean debug = true;
/**
 * Reference to the OfferSpace object
 */
private String OSRef;
/**
 * Reference to the LinkSpace object
 */
private String LSRef;

/**
 * Constructor
 *
 * @param n the name of the Trader.
 * @param args array of command line arguments
 */
public Trader_impl(String n, String args[]) throws RemoteException
{
    name=n;
    reqSeen = new Vector(3,2);

    if (args.length != 1)
    {
        System.out.println("Error: Must specify config file!");
        System.exit(0);
    }
    else
    {
        try
        {
            BufferedReader in = new BufferedReader(new FileReader(args[0]));

            name = in.readLine();
            lui = in.readLine();
            regi = in.readLine();

            tradPolicy = new traderPolicyRecord((new
Integer(in.readLine())).intValue(),
                (new Integer(in.readLine())).intValue(),
                in.readLine(),
                in.readLine(),
                in.readLine());
            OSRef = in.readLine();
            LSRef = in.readLine();

            in.close();

            // Locate the OS and LS Objects
            if (debug) System.out.println("Searching for OS and LS Objects!");
            linkSpace = (LinkSpace)Naming.lookup(LSRef);
            offerSpace = (OfferSpace)Naming.lookup(OSRef);
            if (debug) System.out.println("Found them!");

```

```

    }
    catch (Exception e)
    {
        System.out.println("Trader_impl server lookup exception: " +
e.getMessage());
        e.printStackTrace();
    }
    System.out.println("Config file = "+args[0]);
}
}

/**
 * Method that allows clients to query the offer space.
 *
 * @param originator the name of the object performing the query.
 * @param serv the service type that needs to be matched.
 * @param ip the import policy associated with the request.
 * @return A vector of matching services.
 */
public Vector query(String originator, serviceRecord serv, impPolicyRecord ip)
{
    Vector offers = new Vector(2,2);
    Vector links = null;
    impPolicyRecord dupPolicy = null;
    boolean duplicate=false;

    System.out.println("h_c="+ip.hopCount+" Started a new Query from
"+originator+": "+serv+" "+ip);
    checkParameters();
    checkSecurity();

    dupPolicy = reqIdExists(reqSeen,ip);
    if (dupPolicy != null)
    {
        duplicate = true;
    }

    if (debug) System.out.println("h_c="+ip.hopCount+" Duplicate = "+duplicate);

    if(duplicate)
    {
        // Request Is_Duplicate
        if (debug) System.out.println("h_c="+ip.hopCount+" Found a duplicate
message. dupPolicy="+dupPolicy);

        if((ip.hopCount <= dupPolicy.hopCount) || (ip.hopCount == 0))
        {
            if (debug) System.out.println("h_c="+ip.hopCount+" No Link Search Reqd :
Returning Empty offers List!");
            // No Link_Search_Reqd
            return(offers);
            // end of processing
        }
        else
        {
            if (debug) System.out.println("h_c="+ip.hopCount+" IP hc="+ip.hopCount+"
dupPolicy hc="+dupPolicy.hopCount);
        }
    }

    // Request Is_Unique
    // Unify_Policy
    String newPolicy = unifyPolicy(ip,tradPolicy);
    if (debug) System.out.println("h_c="+ip.hopCount+" IP="+ip);

    if((ip.hopCount == 0) || (newPolicy.equals("local_only")))
    {

```

```

// Local_Search_Only
if(duplicate)
{
    // Return empty list of matches
    if (debug) System.out.println("h_c="+ip.hopCount+" Local Search Only and
duplicate");
    return(offers);
    // end of processing
}
else
{
    // Not a duplicate so do Local Search only
    try
    {
        if (debug) System.out.println("h_c="+ip.hopCount+" Getting Local
Offers");

        // RMI on offerSpace Object
        offers = queryLocalOS(offers,serv);

        if (debug) System.out.println("h_c="+ip.hopCount+" Returning
="+offers);
        return(offers);
        // end of processing
    }
    catch (Exception e)
    {
        if (debug) System.out.println("h_c="+ip.hopCount+" QueryThread OS RMI
exception: " + e.getMessage());
        e.printStackTrace();
    }
}
else
{
    // Link_and/or_Local_Search
    // Get links from LinkSpace
    try
    {
        if(newPolicy.equalsIgnoreCase("always"))
        {
            if (debug) System.out.println("h_c="+ip.hopCount+" Unified =
"+newPolicy);
            if(duplicate==false)
            {
                // Get Local Offers
                if (debug) System.out.println("h_c="+ip.hopCount+" Getting Local
Offers");

                offers = queryLocalOS(offers,serv);
            }
            else
            {
                if (debug) System.out.println("h_c="+ip.hopCount+" Skipping local
search since we did it before!");
            }

            if (debug) System.out.println("h_c="+ip.hopCount+" Querying Linked
Traders");
            // Query Linked Traders
            offers = queryLinkedTraders(originator,offers,serv,ip);
        }
        else
        {
            // Unified Policy = If_No_Local
            // Get Local Offers
            if (debug) System.out.println("h_c="+ip.hopCount+" Unified =

```

```

if_no_local"+newPolicy);
        if (debug) System.out.println("h_c="+ip.hopCount+" Getting Local
Offers");
        offers = queryLocalOS(offers,serv);

        if(offers.size() == 0)
        {
            if (debug) System.out.println("h_c="+ip.hopCount+" No Local
Matches! Querying Linked Traders");
            // Query Linked Traders
            offers = queryLinkedTraders(lui,offers,serv,ip);

        }
    }
}
catch (Exception e)
{
    System.out.println("h_c="+ip.hopCount+" QueryThread OS RMI exception: " +
e.getMessage());
    e.printStackTrace();
}
}
// Return the matching offers
if (debug) System.out.println("h_c="+ip.hopCount+1)+" Returning =" +offers);

return(offers);
}

/**
 * Performs a linear
 *
 *
 */
Vector addOffers(Vector offers, Vector newOffers)
{
    for (int i=0; i<newOffers.size(); i++)
    {
        offers.addElement((offerRecord)newOffers.elementAt(i));
    }
    return(offers);
}

/**
 *
 *
 *
 */
Vector queryLocalOS(Vector offers, serviceRecord serv)
{
    try
    {
        Vector linkOffers = offerSpace.getOffers(name,serv);
        if (debug) System.out.println("Answer = "+linkOffers);

        for (int i=0; i<linkOffers.size(); i++)
        {
            offers.addElement((serviceRecord)linkOffers.elementAt(i));
        }
    }
    catch (Exception e)
    {
        System.out.println("Query Local OS RMI exception: " + e.getMessage());
        e.printStackTrace();
    }
    return(offers);
}
}

```

```

/**
 * Import from all linked Traders
 *
 */
Vector queryLinkedTraders(String originator, Vector offers, serviceRecord serv,
impPolicyRecord ip)
{
    Random rand = new Random();
    try
    {
        if (debug) System.out.println("h_c="+ip.hopCount+" Getting Links");
        Vector links = linkSpace.getLinks(name);
        if (debug) System.out.println("h_c="+ip.hopCount+" Links obtained:"+links);

        // Import from Linked Traders
        if (debug) System.out.println("h_c="+ip.hopCount+" Querying Linked
Traders!");

        // Decrementing hop_count
        --ip.hopCount;

        if(links != null)
        {
            for (int i=0; i<links.size(); i++)
            {
                linkRecord link = (linkRecord) links.elementAt(i);
                String newFollow = followPolicy(ip, tradPolicy, link);

                if(!newFollow.equals("local_only"))
                {
                    if(!link.getLUI().equals(originator))
                    {
                        if (debug) System.out.println("h_c="+ip.hopCount+1+" Linking:
originator="+originator+", target link.LUI="+link.getLUI());

                        Trader linkedTrader = (Trader) Naming.lookup(link.getLUI());
                        impPolicyRecord passIp = ip;
                        passIp.linkFollowRule = passOnPolicy(ip, tradPolicy, link);
                        Vector linkOffers = linkedTrader.query(lui, serv, passIp);

                        if(linkOffers != null)
                        {
                            for (int j=0; j<linkOffers.size(); j++)
                            {
                                offers.addElement((serviceRecord)linkOffers.elementAt(j));
                            }
                        }
                    }
                }
                else
                {
                    if (debug) System.out.println("h_c="+ip.hopCount+1+" Skipping
bi-directional Link to "+originator);
                }
            }
        }
    }
    catch (Exception e)
    {
        if (debug) System.out.println("h_c="+ip.hopCount+" Query Linked Traders RMI
exception: " + e.getMessage());
        e.printStackTrace();
    }
    return(offers);
}

```

```

}

/**
 * Unifies the Import Policy follow option and the Trader Policy follow option
 *
 * @param ip Import Policy Record
 * @param tp Trading Policy Record
 * @return The unified follow option string
 */
String unifyPolicy(impPolicyRecord ip, traderPolicyRecord tp)
{
    linkRecord lr = new linkRecord();
    lr.setLFR("always");
    return(followPolicy(ip, tp, lr));
}

/**
 * Determines the pass on follow option for linked Traders
 *
 * @param ip Import Policy Record
 * @param tp Trading Policy Record
 * @param link Link to be followed
 * @return The pass-on follow option string
 */
String passOnPolicy(impPolicyRecord ip, traderPolicyRecord tp, linkRecord link)
{
    if (!ip.linkFollowRule.equals(""))
    {
        return(ip.linkFollowRule);
    }
    else
    {
        return(smallestPolicy(link.getDFR(), tp.maxFollowRule, "always"));
    }
}

/**
 * Unifies the Import Policy follow option and the Trader Policy follow option for
 a particular link
 *
 * @param ip Import Policy Record
 * @param tp Trading Policy Record
 * @param link Link to be followed
 * @return The unified follow option string
 */
String followPolicy(impPolicyRecord ip, traderPolicyRecord traderPolicy,
linkRecord link)
{
    if (!ip.linkFollowRule.equals(""))
    {
        return(smallestPolicy(traderPolicy.maxFollowRule, link.getLFR(),
ip.linkFollowRule));
    }
    else
    {
        return(smallestPolicy(traderPolicy.maxFollowRule, link.getLFR(),
traderPolicy.defFollowRule));
    }
}

/**
 * Determines the smallest follow option from the 3 parameters. Based upon the
Trading Standard (1996) section 8.2.7.6 Link Follow Behaviour.
 *
 * @param o1 Follow Option 1
 * @param o2 Follow Option 2

```

```

    * @param o3 Follow Option 3
    * @return The smallest follow option string
    */
    String smallestPolicy(String o1, String o2, String o3)
    {
        if (o1.equalsIgnoreCase("local_only") || o2.equalsIgnoreCase("local_only") ||
o3.equalsIgnoreCase("local_only"))
        {
            return("local_only");
        }
        else
        {
            if (o1.equalsIgnoreCase("if_no_local") || o2.equalsIgnoreCase("if_no_local")
|| o3.equalsIgnoreCase("if_no_local"))
            {
                return("if_no_local");
            }
            else
            {
                return("always");
            }
        }
    }

    /**
    * Determines whether the Request Id element exists in the Vector of seen Request
    Ids.
    *
    * @param reqSeen Vector of Request Ids of processed Queries.
    * @param r_id Request Id to be tested for.
    * @return If the Request ID exists, then it is returned, else null.
    */
    impPolicyRecord reqIdExists(Vector reqSeen, impPolicyRecord r_id)
    {
        int i=0;
        boolean flag=false;

        if (debug) System.out.println("h_c="+r_id.hopCount+" Seen list = "+reqSeen);
        while(i<reqSeen.size())
        {
            impPolicyRecord temp = (impPolicyRecord)reqSeen.elementAt(i);
            if (debug) System.out.println("h_c="+r_id.hopCount+" Compare existing
record:"+temp.requestId+", New record:"+r_id.requestId+" Answer is
"+(temp.requestId).equals(r_id.requestId));
            if ((temp.requestId).equals(r_id.requestId))
            {
                if (debug) System.out.println("h_c="+r_id.hopCount+" Returning duplicate
Policy="+temp);
                return(temp);
            }
            i++;
        }
        impPolicyRecord temp = (impPolicyRecord) new impPolicyRecord(r_id.hopCount,
r_id.linkFollowRule, r_id.requestId, r_id.exactMatchType);
        reqSeen.addElement(temp);
        if (debug) System.out.println("h_c="+r_id.hopCount+" Just added "+r_id);
        return(null);
    }

    /**
    * Placekeeper function that checks Query parameters.
    */
    void checkParameters()
    {
        if (debug) System.out.println("Parameters checked!");
    }

```

```

/**
 * Placekeeper function that checks Security privileges of the importer.
 */
void checkSecurity()
{
    if (debug) System.out.println("Security checked!");
}

/**
 * Method that allows clients to add offers to the offer space.
 *
 * @param serviceRecord service to be exported.
 * @return An integer representing the unique offer transaction
 *         identifier. -1 if the method failed.
 */
public int export(serviceRecord serv)
{
    int offerIdent = -1;
    try
    {
        // Create a new offer record for the service record
        offerRecord newOffer = new offerRecord(name, serv.serviceType,
serv.properties, serv.location);

        // Add the offers to the Offer Space
        offerIdent = offerSpace.addOffer(name, newOffer);
        System.out.println("Added the offer to the Offer Space");
        newOffer.setOfferIdent(offerIdent);
    }
    catch (Exception e)
    {
        System.out.println("Export method exception: " + e.getMessage());
        e.printStackTrace();
    }
    return(offerIdent);
}

/**
 * Main function that creates a new Trader and binds it in the name server.
 */
public static void main(String args[])
{
    // Create and install a security manager
    System.setSecurityManager(new RMISecurityManager());

    System.out.println("In the Trading Object");

    try
    {
        Trader_impl obj = new Trader_impl(name, args);
        Naming.rebind(lui, obj); // boson for 28.8!
        System.out.println(lui+" bound in registry");
    }
    catch (Exception e) {
        System.out.println(name+" err: " + e.getMessage());
        e.printStackTrace();
    }
}
}

```

B.3.2 OfferSpace_impl.java

```

import java.util.*;
import java.rmi.*;

```

```

import java.io.*;
import java.rmi.server.UnicastRemoteObject;

/**
 * OfferSpace_impl implements the Offer Space Object.
 * It stores Offers for later retrieval.
 *
 * @version 1.00
 * @author Andrew Tokmakoff
 */
public class OfferSpace_impl
    extends UnicastRemoteObject
    implements OfferSpace
{
    /**
     * for debugging!
     */
    private static boolean debug = true;

    /**
     * A Vector containing the Offer Space
     */
    private Vector OfferSpace = new Vector();

    /**
     * The name of the Offer Space Instance
     */
    private String name;

    /**
     * Used to create a random (non-unique) Offer Identifier
     */
    private Random rand;

    /**
     * Constructor
     * @param s Name of the Offer Space object.
     */
    public OfferSpace_impl(String s) throws RemoteException
    {
        super();
        name = s;
        rand = new Random(1234);
    }

    /**
     * A Function that allows a Trader to retrieve offers from
     * the Offer Space.
     *
     * @param trader The Trader whose offer space is to modified.
     * @param serv the service that is being added to the offer space.
     */
    public Vector getOffers(String trader, ServiceRecord serv) throws RemoteException
    {
        Vector traderOffers = new Vector(2,2);
        Vector matchingOffers = new Vector(2,2);

        System.out.println("Doing the search!");
        System.out.println("Searching "+trader+" for "+serv);

        try
        {
            int i=0;
            boolean flag = false;

            // Locate the appropriate Trader record in the Offer Space
            while ((i<OfferSpace.size()) && (flag==false))

```

```

    {
        offerStore temp = (offerStore) OfferSpace.elementAt(i);
        if(temp.trader.equalsIgnoreCase(trader))
        {
            traderOffers = temp.offers;
            flag = true;
            if (debug) System.out.println("Found the Trader record");
        }
        if (debug) System.out.println("Element at "+i+" is "+temp);
        ++i;
    }

    // Search Trader offers for service type matches.
    for(i=0; i<traderOffers.size(); i++)
    {
        // Find offers that are of the correct service type
        if (((offerRecord)
traderOffers.elementAt(i)).getService()).equalsIgnoreCase(serv.getService()))
        {
            servProps matchProps =
((offerRecord)traderOffers.elementAt(i)).getProperties();
            servProps serviceProps = serv.getProperties();

            if (debug) System.out.println("Similarity
="+((servProps)matchProps).similarity((servProps)serviceProps));

            if(((servProps)matchProps).similarity(serviceProps)>=0.666)
            {
                if (debug) System.out.println("Service Type OK and properties
match!!");

                serviceRecord match = new serviceRecord();
                match.setService(serv.getService());

                match.setLocation(((offerRecord)traderOffers.elementAt(i)).getLocation());
                match.setProperties(matchProps);
                if (debug) System.out.println("Adding "+match);
                matchingOffers.addElement(match);
            }
            else
            {
                if (debug) System.out.println("Service Type OK, but not enough
matches");
            }
        }
    }

    System.out.println("Finished the search!");
    return(matchingOffers);
}
catch (Exception e)
{
    System.out.println("OfferSpace_Impl err: " + e.getMessage());
    e.printStackTrace();
}
return(null);
}

/**
 * Adds an offer to a Trader's Offer Space.
 *
 * @param trader The Trader whose offer space is to modified.
 * @param offer the offer to be added to the offer space.
 */
public int addOffer(String trader, offerRecord offer) throws
java.rmi.RemoteException
{

```

```

offer.setOfferIdent(rand.nextInt());
try
{
    int i;

    /**
     * Search the offer space for an existing element.
     * If one does not already exist, then create one
     * and add the offer to the offer space.
     */
    for (i=0; i<OfferSpace.size(); i++)
    {
        if(((offerStore)OfferSpace.elementAt(i)).trader.equalsIgnoreCase(trader))
            break;
    }

    if (i==OfferSpace.size())
    {
        if (debug) System.out.println("Creating OfferSpace element for "+trader);
        offerStore newLink = new offerStore(trader);
        newLink.offers.addElement(offer);
        OfferSpace.addElement(newLink);
    }
    else
    {
        if (debug) System.out.println("Found existing OfferSpace element for
"+trader);
        ((offerStore)OfferSpace.elementAt(i)).offers.addElement(offer);
    }
}
catch (Exception e)
{
    System.out.println("OfferSpace_Impl err: " + e.getMessage());
    e.printStackTrace();
}
return(offer.getOfferIdent());
}

/**
 * Routine to remove an offer from the Offer Space.
 *
 * @param trader The Trader whose offer space is to modified.
 * @param offerId Identifier of the offer to be removed.
 */
public void removeOffer(String trader, int offerId) throws
java.rmi.RemoteException
{
    try
    {
        int i;

        for (i=0; i<OfferSpace.size(); i++)
        {
            if(((offerStore)OfferSpace.elementAt(i)).trader.equalsIgnoreCase(trader))
                break;
        }

        if (i==OfferSpace.size())
        {
            if (debug) System.out.println("No OfferSpace element for "+trader);
            if (debug) System.out.println("removeOffer("+offerId+") failed!");
        }
        else
        {
            if (debug) System.out.println("Found existing OfferSpace element for
"+trader);
            if (debug) System.out.println("Removing "+offerId+ " from

```

```

"+(offerStore)OfferSpace.elementAt(i));
        ((offerStore)OfferSpace.elementAt(i)).removeOffer(offerId);
    }
}
catch (Exception e)
{
    System.out.println("OfferSpaceImpl err: " + e.getMessage());
    e.printStackTrace();
}
}

/**
 * Routine used during testing of RMI
 */
public String sayHello() throws java.rmi.RemoteException
{
    return("Offer Space says hi!");
}

/**
 * Routine to print the Offer Space to the screen.
 *
 * @param t Trader whose offers are to be printed.
 */
public void printOffers(String trader) throws java.rmi.RemoteException
{
    int i;
    boolean flag=false;
    for (i=0; i<OfferSpace.size(); i++)
    {
        String current_trader = ((offerStore)OfferSpace.elementAt(i)).trader;
        if (current_trader.equalsIgnoreCase(trader))
        {
            System.out.println("Offers for "+OfferSpace.elementAt(i));
            flag=true;
        }
    }
    if (flag == false) System.out.println("No Offers for "+trader);
}

/**
 * Routine to initialise the Offer Space from a test file.
 *
 * @param traderName Name of the Trader whose offer space is being initialised.
 */
public void initOfferSpace(String traderName) throws java.rmi.RemoteException
{
    String temp;
    int i=0;

    try
    {
        BufferedReader in = new BufferedReader(new FileReader(traderName+
".offers"));

        while ((temp = in.readLine()) != null)
        {
            if (temp.equalsIgnoreCase("Service") )
            {
                offerRecord tempOffer = new offerRecord();
                // Set the owner (trader) of the Offer
                tempOffer.setTrader(traderName);

                // Read Service Type
                temp= in.readLine();
                tempOffer.setService(temp);
            }
        }
    }
}

```

```

if (debug) System.out.println("temp="+temp);

servProps serv = null;

/**
 * Hard-coded service specific section.
 * Could be modified to use a Type Repository
 * That returns the appropriate service Record
 * based upon the service type string.
 */
if(temp.equalsIgnoreCase("pizza"))
{
    serv = (pizzaServRec) new pizzaServRec();
}
else if(temp.equalsIgnoreCase("used_cars"))
{
    serv = (usedCarServRec) new usedCarServRec();
}
else
{
    System.out.println("Error : Unknown Service Type!" +serv);
}

// Read Service Location
temp= in.readLine();
tempOffer.setLocation(temp);

Vector data = new Vector(3,2);

// Read First Service Property
temp= in.readLine();

while(!(temp.equalsIgnoreCase("")))
{
    data.addElement(temp);
    if (debug) System.out.println("added temp="+temp);
    temp=in.readLine();
}

// Set service attributes
serv.setData(data);
tempOffer.properties = serv;

if (debug) System.out.println("Adding "+data);

if (debug) System.out.println("tempOffer="+tempOffer);
addOffer(traderName, tempOffer);
}
    }
        in.close();
    } catch (FileNotFoundException e) {
        System.err.println("File Not found. InitOfferSpace: " + e);
    } catch (IOException e) {
        System.err.println("File error. InitOfferSpace: " + e);
    }
}

/**
 * Main function that creates an Offer Space Object, binds it
 * to the name server and initialises the Offer Space for three
 * Traders.
 */
public static void main(String args[])
{
    // Create and install a security manager

```

```

System.setSecurityManager(new RMISecurityManager());

System.out.println("In the Offer Space Object");

try
{
    OfferSpace_impl obj = new OfferSpace_impl("OfferSpaceServer");
    Naming.rebind("OfferSpaceServer", obj);    // boson for 28.8!
    System.out.println("OfferSpaceServer bound in registry");

    /**
     * Initialise three Traders.
     * Could be moved to an admin. interface
     */
    obj.initOfferSpace("trader_1");
    obj.initOfferSpace("trader_2");
    obj.initOfferSpace("trader_3");
    obj.initOfferSpace("trader_4");

    /**
     * Print out the Offers for each Trader
     */
    if (debug)
    {
        obj.printOffers("Trader_1");
        obj.printOffers("Trader_2");
        obj.printOffers("Trader_3");
        obj.printOffers("Trader_4");
    }
}
catch (Exception e) {
    System.out.println("OfferSpaceImpl err: " + e.getMessage());
    e.printStackTrace();
}
}
}

```

B.3.3 LinkSpace_impl.java

```

import java.io.*;
import java.util.*;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

/**
 * LinkSpace_impl implements the Link Space Object.
 * It maintains a database of Trader Links for later retrieval.
 *
 * @version 1.00
 * @author Andrew Tokmakoff
 */
public class LinkSpace_impl
    extends UnicastRemoteObject
    implements LinkSpace
{
    /**
     * for debugging!
     */
    private static boolean debug = false;

    /**
     * A Vector containing the Link Space
     */
    private Vector linkSpace = new Vector();

```

```

/*
 * The name of the Link Space Instance
 */
private String name;

/**
 * Constructor
 * @param s Name of the Link Space object.
 */
public LinkSpace_impl(String s) throws RemoteException
{
    super();
    name = s;
}

/**
 * A Function that allows a Trader to retrieve links from
 * the link Space.
 *
 * @param trader The Trader whose link space is to modified.
 */
public Vector getLinks(String trader) throws RemoteException
{
    try
    {
        for (int i=0; i<linkSpace.size();i++)
        {
            linkStore temp = (linkStore) linkSpace.elementAt(i);
            if(temp.trader.equalsIgnoreCase(trader))
            {
                return(temp.links);
            }
        }
    }
    catch (Exception e)
    {
        System.out.println("HelloImpl err: " + e.getMessage());
        e.printStackTrace();
    }
    return(null);
}

/**
 * Adds a link to a Trader's Link Space.
 *
 * @param trader The Trader whose link space is to modified.
 * @param link Link record to be added to the link space.
 */
public void addLink(String trader, linkRecord link) throws
java.rmi.RemoteException
{
    try
    {
        int i;

        for (i=0; i<linkSpace.size(); i++)
        {
            if(((linkStore)linkSpace.elementAt(i)).trader.equalsIgnoreCase(trader))
                break;
        }

        if (i==linkSpace.size())
        {
            if (debug) System.out.println("Creating linkSpace element for "+trader);
            linkStore newLink = new linkStore(trader);
            newLink.links.addElement(link);
            linkSpace.addElement(newLink);
        }
    }
}

```

```

        }
        else
        {
            if (debug) System.out.println("Found existing linkSpace element for
"+trader);
            ((linkStore)linkSpace.elementAt(i)).links.addElement(link);
        }
    }
    catch (Exception e)
    {
        System.out.println("HelloImpl err: " + e.getMessage());
        e.printStackTrace();
    }
}

/**
 * Routine to remove a Link from the Link Space.
 *
 * @param trader The Trader whose Link space is to modified.
 * @param linkName Name of the Link to be removed.
 */
public void removeLink(String trader, String linkName) throws
java.rmi.RemoteException
{
    try
    {
        int i;

        for (i=0; i<linkSpace.size(); i++)
        {
            if(((linkStore)linkSpace.elementAt(i)).trader.equalsIgnoreCase(trader))
                break;
        }

        if (i==linkSpace.size())
        {
            if (debug) System.out.println("No linkSpace element for "+trader);
            if (debug) System.out.println("removeLink("+linkName+") failed!");
        }
        else
        {
            if (debug) System.out.println("Found existing linkSpace element for
"+trader);
            if (debug) System.out.println("Removing "+linkName+" from
"+(linkStore)linkSpace.elementAt(i));
            ((linkStore)linkSpace.elementAt(i)).removeLink(linkName);
        }
    }
    catch (Exception e)
    {
        System.out.println("HelloImpl err: " + e.getMessage());
        e.printStackTrace();
    }
}

/**
 * Routine used during testing of RMI
 */
public String sayHello() throws java.rmi.RemoteException
{
    return("Link Space says hi!");
}

/**
 * Routine to print the Link Space to the screen.
 *
 * @param t Trader whose Links are to be printed.

```

```

    /**
    public void printLinks(String t)
    {
        int i;
        boolean flag=false;
        for (i=0; i<linkSpace.size(); i++)
        {
            String current_trader = ((linkStore)linkSpace.elementAt(i)).trader;
            if (current_trader.equalsIgnoreCase(t))
            {
                if (debug) System.out.println("printLinks() for
"+linkSpace.elementAt(i));
                flag=true;
            }
        }
        if (flag == false)
        {
            if (debug) System.out.println("No Links for "+t);
        }
    }

    /**
    * Routine to initialise the Link Space from a test file.
    *
    * @param traderName Name of the Trader whose Link space is being initialised.
    */
    public void initLinkSpace(String traderName) throws java.rmi.RemoteException
    {
        String temp;
        int i=0;

        try
        {
            BufferedReader in = new BufferedReader(new FileReader(traderName+".links"));

            while ((temp = in.readLine()) != null)
            {
                if (temp.equalsIgnoreCase("Link") )
                {
                    linkRecord tempLink = new linkRecord();

                    // Read Link name
                    temp= in.readLine();
                    tempLink.setName(temp);

                    // Read Lookup Interface name
                    temp= in.readLine();
                    tempLink.setLUI(temp);

                    // Read Default Follow Rule
                    temp= in.readLine();
                    tempLink.setDFR(temp);

                    // Read Limiting Follow Rule
                    temp= in.readLine();
                    tempLink.setLFR(temp);

                    if (debug) System.out.println("tempLink="+tempLink);
                    addLink(traderName, tempLink);
                }
            }

            in.close();
        } catch (FileNotFoundException e) {
            System.err.println("FileStreamsTest: " + e);
        } catch (IOException e) {
            System.err.println("FileStreamsTest: " + e);
        }
    }

```

```

    }
}

/**
 * Main function that creates an Link Space Object, binds it
 * to the name server and initialises the Link Space for three
 * Traders.
 */
public static void main(String args[])
{
    // Create and install a security manager
    System.setSecurityManager(new RMISecurityManager());

    System.out.println("In the Link Space Object");

    try
    {
        LinkSpace_impl obj = new LinkSpace_impl("LinkSpaceServer");
        Naming.rebind("LinkSpaceServer", obj); // boson for 28.8!
        System.out.println("LinkSpaceServer bound in registry");

        /**
         * Initialise three Traders.
         * Could be moved to an admin. interface
         */
        obj.initLinkSpace("trader_1");
        obj.initLinkSpace("trader_2");
        obj.initLinkSpace("trader_3");

        /**
         * Print out the Links for each Trader
         */
        if(debug)
        {
            obj.printLinks("Trader_1");
            obj.printLinks("Trader_2");
            obj.printLinks("Trader_3");
        }
    }
    catch (Exception e) {
        System.out.println("LinkSpaceImpl err: " + e.getMessage());
        e.printStackTrace();
    }
}
}

```

B.4 Applets

B.4.1 OfferSpaceApplet.java

```

import java.awt.*;
import java.util.*;
import java.rmi.*;

/**
 * A class used by the client to test the Offer Space object.
 *
 * @author Andrew Tokmakoff
 * @version 1.00
 */
public class OfferSpaceApplet extends java.applet.Applet
{

```

```

private String message;
private String tName = "Trader_1";
private int offerIdent1 = -1;
private int offerIdent2 = -1;
private int offerIdent3 = -1;

/**
 * Called when the applet starts in the browser.
 */
public void init()
{
    try
    {
        // Look up the Offer Space object
        OfferSpace obj = (OfferSpace)Naming.lookup("OfferSpaceServer");

        // Create a new service record
        pizzaServRec offerParams1 = new pizzaServRec();
        offerParams1.pizzaType="the_lot";
        offerParams1.size=16;
        offerParams1.delivery=true;

        // Create a new service record
        usedCarServRec offerParams2 = new usedCarServRec();
        offerParams2.carType="sports";
        offerParams2.cylinders=8;
        offerParams2.lpg=true;

        // Create a new service record
        usedCarServRec offerParams3 = new usedCarServRec();
        offerParams3.carType="wagon";
        offerParams3.cylinders=6;
        offerParams3.lpg=false;

        // Create a new offer record for each service record
        offerRecord newOffer1 = new offerRecord(tName,"pizza",
offerParams1,"pizza_haven");
        offerRecord newOffer2 = new offerRecord(tName,"used_cars",offerParams2,
"bob_moran");
        offerRecord newOffer3 = new offerRecord(tName,"used_cars",offerParams3,
"australian_motors");

        // Add the offers to the Offer Space
        offerIdent1 = obj.addOffer("Trader_1",newOffer1);
        newOffer1.setOfferIdent(offerIdent1);

        offerIdent2 = obj.addOffer("Trader_1",newOffer2);

        newOffer2.setOfferIdent(offerIdent2);

        offerIdent3 = obj.addOffer("Trader_1",newOffer3);

        newOffer3.setOfferIdent(offerIdent3);

        // Print the Offer space of the Traders
        obj.printOffers("Trader_1");
        obj.printOffers("Trader_2");

        // Remove an offer
        removeOffer("Trader_1", offerIdent3);

        // Ensure that it was removed
        obj.printOffers("Trader_1");

        // Create service properties for retrieval
        usedCarServRec carProps = new usedCarServRec();
        carProps.carType = "wagon";

```

```

        carProps.cylinders = 6;
        carProps.lpg = false;

        // Create a service record for retrieval
        serviceRecord SR = new serviceRecord("used_cars",carProps,"");

        // Query the offer space object
        Vector results = obj.getOffers("Trader_1",SR);
        System.out.println("Results from Car search\n"+results);

        // Create service properties for retrieval
        pizzaServRec pizzaProps = new pizzaServRec();
        pizzaProps.pizzaType = "the_lot";
        pizzaProps.size = 12;
        pizzaProps.delivery = false;

        // Create a service record for retrieval
        SR = new serviceRecord("pizza",pizzaProps,"");

        // Query the offer space object
        results = obj.getOffers("Trader_2",SR);
        System.out.println("Results from Pizza search\n"+results);

        message = obj.sayHello();
    }
    catch (Exception e)
    {
        System.out.println("OfferSpaceApplet exception: " + e.getMessage());
        e.printStackTrace();
    }
}

/**
 * Method used to display the results from interaction with
 * the Offer Space object.
 *
 * @param g Graphics context
 */
public void paint(Graphics g)
{
    g.drawString(message, 25, 50);
}
}

```

B.4.2 LinkSpaceApplet.java

```

import java.awt.*;
import java.rmi.*;

/**
 * A class used by the client to test the Link Space object.
 *
 * @author Andrew Tokmakoff
 * @version 1.00
 */
public class LinkSpaceApplet extends java.applet.Applet
{
    private String message;

    /**
     * Called when the applet starts in the browser.
     */
    public void init()
    {
        try
        {

```

```

// Look up the Link Space object
LinkSpace obj = (LinkSpace)Naming.lookup("LinkSpaceServer");

// Create some new link records
linkRecord newLink1 = new linkRecord("L1","lui_2","if_no_local","always");
linkRecord newLink2 = new
linkRecord("L2","lui_1","local_only","if_no_local");

// Add the links to the link space
obj.addLink("Trader_1",newLink1);
obj.addLink("Trader_1",newLink2);

// Print Trader 1's links (compare with the link added)
obj.printLinks("Trader_1");
System.out.println(newLink1);

// Print Trader 2's links (compare with the link added)
obj.printLinks("Trader_2");
System.out.println(newLink2);

// Remove Link L1 from Trader 1
obj.removeLink("Trader_1","L1");

// Print Trader 1's links
obj.printLinks("Trader_1");

message = obj.sayHello();
}
catch (Exception e)
{
    System.out.println("HelloApplet exception: " + e.getMessage());
    e.printStackTrace();
}
}

/**
 * Method used to display the results from interaction with
 * the Link Space object.
 *
 * @param g Graphics context
 */
public void paint(Graphics g)
{
    g.drawString(message, 25, 50);
}
}

```

B.4.3 TradGuiApplet.java

```

import java.rmi.*;
import java.util.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.util.Random;

/**
 * A class used by the client to Import from the Trader.
 *
 * @author Andrew Tokmakoff
 * @version 0.8
 */
public class TradGuiApplet extends Applet
    implements ActionListener,
        ItemListener

```

```

{
    private Choice serviceType;

    private Choice carType;
    private Choice cylinders;
    private Choice lpg;

    private Choice destTrader;
    private Choice hopCount;
    private Choice followPolicy;
    private Choice reqId;

    private Panel p;
    private Panel sPolicy;
    private Panel textp;
    private Panel resultPanel;
    private MenuBar mbar;
    private TextArea result;

    private Trader trader;

    // Globals for use in the Trading App

    private static final int maxID = 64;

    private usedCarServRec carProps;
    private serviceRecord sR;
    private reqIdRecord idR;
    private impPolicyRecord iPl;

    /**
     * Name of this Client Object
     */
    public static final String name = "Netscape Browser";

    /**
     * Called when the applet starts in the browser.
     */
    public void init()
    {
        // Short term hack for Communicator
        // netscape.security.PrivilegeManager.enablePrivilege("UniversalConnect");

        setBackground(Color.cyan);
        textp = new Panel();
        textp.setLayout(new FlowLayout(FlowLayout.CENTER));

        Button newRequest = new Button("New Request");
        newRequest.addActionListener(this);
        textp.add(newRequest);

        Button cancel = new Button("Cancel");
        cancel.addActionListener(this);
        textp.add(cancel);

        Button retry = new Button("Retry");
        retry.addActionListener(this);
        textp.add(retry);

        Button exit = new Button("EXIT");
        exit.addActionListener(this);
        textp.add(exit);

        Button submit = new Button("Submit");
        submit.addActionListener(this);
        textp.add(submit);
    }
}

```

```

add("North",textp);

// Add GUI for selecting the Service Parameters
p = new Panel();
p.add(new Label("Select Service Type"));

serviceType = new Choice();
serviceType.addItemListener(this);
serviceType.addItem("Used_Cars");
serviceType.addItem("Pizza");
p.add(serviceType);

p.add(new Label("Car Type"));

carType = new Choice();
carType.addItemListener(this);
carType.addItem("Sports");
carType.addItem("Wagon");
p.add(carType);

p.add(new Label("Cylinders"));
cylinders = new Choice();
cylinders.addItemListener(this);
cylinders.addItem("4");
cylinders.addItem("6");
cylinders.addItem("8");
p.add(cylinders);

p.add(new Label("Fitted with LPG"));
lpg = new Choice();
lpg.addItemListener(this);
lpg.addItem("True");
lpg.addItem("False");
p.add(lpg);
add("Center", p);

// Add GUI for Search policy selection

sPolicy = new Panel();

sPolicy.add(new Label("Destination Trader Lookup Interface"));
destTrader = new Choice();
destTrader.addItemListener(this);
destTrader.addItem("LUI_1");
destTrader.addItem("LUI_2");
destTrader.addItem("LUI_3");
sPolicy.add(destTrader);
destTrader.select(0);

sPolicy.add(new Label("Link Search policy"));
followPolicy = new Choice();
followPolicy.addItemListener(this);
followPolicy.addItem("always");
followPolicy.addItem("if no local");
followPolicy.addItem("local only");
sPolicy.add(followPolicy);

sPolicy.add(new Label("Hop Count"));
hopCount = new Choice();
hopCount.addItemListener(this);
hopCount.addItem("0");
hopCount.addItem("1");
hopCount.addItem("2");
hopCount.addItem("3");

```

```

sPolicy.add(hopCount);

sPolicy.add(new Label("Request Id"));
reqId = new Choice();
reqId.addItemListener(this);
reqId.addItem("-1");
reqId.addItem("-2");
reqId.addItem("-3");
reqId.addItem("-4");
sPolicy.add(reqId);

add("East", sPolicy);

resultPanel = new Panel();
resultPanel.add(new Label("Search Results"));
result = new TextArea(10,80);
resultPanel.add(result);
add("South", resultPanel);
}

/**
 * Function that listens for actions that have occurred to buttons in the GUI.
 *
 * @param event ActionEvent that occurred.
 */
public void actionPerformed(ActionEvent event)
{
    String source = event.getActionCommand();
    if (source.equals("New Request"))
    {
        System.out.println("Got New Request!");
    }
    else if (source.equals("Cancel"))
    {
        System.out.println("Got Cancel");
    }
    else if (source.equals("Retry"))
    {
        System.out.println("Got Retry");
    }
    else if (source.equals("EXIT"))
    {
        System.out.println("Got Exit");
        System.exit(0);
    }
    else if (source.equals("Submit"))
    {
        System.out.println("Got Submit");

        // Connect to the Appropriate Trader
        try
        {
            trader = (Trader)Naming.lookup(destTrader.getSelectedItem());

            if (trader != null) System.out.println("Trader != null");
            else
                System.out.println("Trader reference IS null!");

            System.out.println("Searched for:"+destTrader.getSelectedItem());

            // Extract the service properties
            String service = serviceType.getSelectedItem();
            System.out.println("Service Type: "+service);

            String car = carType.getSelectedItem();
            System.out.println("Car Type: "+car);

```

```

String cyl = cylinders.getSelectedItemAt();
System.out.println("Cylinders: "+cyl);

String lpgState = lpg.getSelectedItemAt();
System.out.println("LPG: "+lpgState);

// extract the Import Policy info
String follPolicy = followPolicy.getSelectedItemAt();
System.out.println("Follow Policy: "+follPolicy);

String tempString = hopCount.getSelectedItemAt();
System.out.println("Hop Count: "+tempString);

Integer tempInt = new Integer(tempString);
int hc = tempInt.intValue();

tempString = reqId.getSelectedItemAt();
System.out.println("Request ID: "+tempString);

tempInt = new Integer(tempString);
int rId = tempInt.intValue();

usedCarServRec carProps = new usedCarServRec();
carProps.carType = car;

tempInt = new Integer(cyl);
carProps.cylinders = tempInt.intValue();

Boolean tempBool = new Boolean(lpgState);
carProps.lpg = tempBool.booleanValue();

serviceRecord sR = new serviceRecord(service,carProps,"");

reqIdRecord idR = new reqIdRecord("Importer",rId);

impPolicyRecord iP1 = new impPolicyRecord(hc,follPolicy,idR,true);

Vector results = null;

results = trader.query("Importer",sR,iP1);

System.out.println("Results from 1st Car search\n"+results);

String message = "Query Results for request "+rId+"
are:"+results.toString()+"\n";
    result.append(message);
}
catch (Exception e)
{
    System.out.println("Trading Client exception: " + e.getMessage());
    e.printStackTrace();
}
}
else
{
    System.out.println("Got another event!");
}
}

/**
 * Function that listens to changes of state in the List and Choice GUI objects.
 *
 * @param event Event that occurred.
 */

```

```

public void itemStateChanged(ItemEvent event)
{
    ItemSelectable source = event.getItemSelectable();

    if(source instanceof List)
    {
        System.out.println("Got an event from a List");
        Object selected = event.getItem();
        System.out.println("Selected "+selected);
    }
    else if(source instanceof Choice)
    {
        String selected = (String) event.getItem();
        System.out.println("Selected "+selected);
    }
}
}

```

B.5 Sample Service Types

B.5.1 servProps.java

```

import java.io.*;
import java.util.*;

/**
 * Generic class used for all properties classes.
 *
 * @author Andrew Tokmakoff
 * @version 1.00
 */
public class servProps implements Serializable
{
    /**
     * Abstract function that determines how similar two services are.
     *
     * @param obj Object to be compared with
     * @return Boolean value for equality.
     */
    public float similarity(Object obj)
    {
        return(0);
    }

    /**
     * Abstract function that is used by the class to initialise its
     * internal fields.
     * @param data Vector of data items to be used in initialisation.
     *             Ordering is assumed to be related to the declaration
     *             of member fields.
     */
    public void setData(Vector data)
    {
    }
}

```

B.5.2 serviceRecord.java

```

import java.util.*;
import java.io.*;

/**

```

```

* An abstract class which stores service type and properties.
*
* @author Andrew Tokmakoff
* @version 1.00
*/
public class serviceRecord implements Serializable
{
    /**
     * the type of service being stored
     */
    protected String serviceType;
    /**
     * properties of the service being stored
     */
    protected servProps properties;
    /**
     * location of the service being stored
     */
    protected String location;

    /**
     * Default constructor.
     */
    public serviceRecord()
    {
    }

    /**
     * Initialising constructor.
     *
     * @param s Service name.
     * @param p Service properties.
     * @param l Service location.
     */
    public serviceRecord(String s, servProps p, String l)
    {
        setService(s);
        setProperties(p);
        setLocation(l);
    }

    /**
     * Accessor Method
     */
    public String getService()
    {
        return(serviceType);
    }

    /**
     * Mutator Method
     */
    public void setService(String s)
    {
        serviceType = s;
    }

    /**
     * Accessor Method
     */
    public servProps getProperties()
    {
        return(properties);
    }

    /**
     * Mutator Method

```

```

    */
    public void setProperties(servProps p)
    {
        properties = p;
    }

    /**
     * Accessor Method
     */
    public String getLocation()
    {
        return(location);
    }

    /**
     * Mutator Method
     */
    public void setLocation(String l)
    {
        location = l;
    }

    /**
     * Converts the class fields into a string for display.
     */
    public String toString()
    {
        return(serviceType+" "+location+" "+properties);
    }
}

```

B.5.3 pizzaServRec.java

```

import java.io.*;
import java.util.*;

/**
 * Class used to describe the pizza service.
 * Must extend servProps and re-implement similarity()
 *
 * @author Andrew Tokmakoff
 * @version 1.00
 */
public class pizzaServRec extends servProps implements Serializable
{
    private static boolean debug = false;

    /**
     * String that denotes the type of pizza.
     */
    public String pizzaType;

    /**
     * Diameter of pizza in inches.
     */
    public int size;

    /**
     * Flag to denote whether the shop supports delivery.
     */
    public boolean delivery;

    /**
     * Constructor.
     */
    public pizzaServRec()
    {

```

```

    pizzaType = "";
    size = 0;
    delivery = false;
}

/**
 * Method that compares two pizzaServRec objects for equality.
 *
 * @param obj Object to be compared with.
 * @return boolean value for equality.
 */
public boolean equal(Object obj)
{
    pizzaServRec other = (pizzaServRec) obj;

    if ((other.pizzaType).equals(pizzaType) &&
        (other.size == size) &&
        (other.delivery == delivery))
        return(true);
    else
        return(false);
}

/**
 * Method that determines how similar the pizzaServRec parameter objects
 * is with this one.
 *
 * @param obj Object to be compared with.
 * @return Value between 0 and 1 that gives a percentage similarity.
 */
public float similarity(Object obj)
{
    int hits=0;

    pizzaServRec other = (pizzaServRec) obj;

    if (debug)
    {
        System.out.println("Testing for Similarity");
        System.out.println(this+" "+other);
    }

    if ((other.pizzaType).equalsIgnoreCase(pizzaType))
    {
        // It is twice as important to have the correct Pizza Type!
        // Simulates use of the constraint language
        hits++;
        hits++;
    }

    if ((other.size == size))
    {
        hits++;
    }
    if ((other.delivery == delivery))
    {
        hits++;
    }
    System.out.println("Total matches: "+hits+"Return value= "+(float)hits/3);

    return((float)hits/3);
}

/**
 * Converts the class fields into a string for display.
 */
public String toString()
{
    return(pizzaType+" "+size+" "+delivery);
}

```

```

/**
 * Function that is used by the class to initialise its internal
 * fields.
 * @param data Vector of data items to be used in initialisation.
 *           Ordering is assumed to be related to the order of
 *           declaration of member fields.
 */
public void setData(Vector data)
{
    int i=0;

    pizzaType=(String) data.elementAt(i++);

    Integer s = new Integer((String)data.elementAt(i++));
    size=(int) s.intValue();

    if(((String)data.elementAt(i++)).lastIndexOf("true")!= -1)
        delivery=true;
    else
        delivery=false;
}
}

```

B.5.4 usedCarServRec.java

```

import java.io.*;
import java.util.*;

/**
 * Class used to describe the used car service.
 * Must extend servProps and re-implement similarity()
 *
 * @author Andrew Tokmakoff
 * @version 1.00
 */
public class usedCarServRec extends servProps implements Serializable
{
    /**
     * String denoting the car type.
     */
    public String carType;

    /**
     * The number of cylinder the car has.
     */
    public int cylinders;

    /**
     * Flag that indicates whether the car supports LPG fuel.
     */
    public boolean lpg;

    /**
     * For debugging!
     */
    private static boolean debug = false;

    /**
     * Constructor
     */
    public usedCarServRec()
    {
        carType = "";
        cylinders = 0;
        lpg = false;
    }
}

```

```

/**
 * Method that compares two usedCarServRec objects for equality.
 *
 * @param obj Object to be compared with.
 * @return Boolean value for equality.
 */
public boolean equal(Object obj)
{
    usedCarServRec other = (usedCarServRec) obj;

    if ((other.carType).equals(carType) &&
        (other.cylinders == cylinders) &&
        (other.lpg == lpg))
        return(true);
    else
        return(false);
}

/**
 * Method that determines how similar the usedCarServRec parameter objects
 * is with this one.
 *
 * @param obj Object to be compared with.
 * @return Value between 0 and 1 that gives a percentage similarity.
 */
public float similarity(Object obj)
{
    int hits=0;

    usedCarServRec other = (usedCarServRec) obj;

    if (debug)
    {
        System.out.println("Testing for Similarity");
        System.out.println(this+" "+other);
    }

    if ((other.carType).equalsIgnoreCase(carType))
    {
        hits++;
    }
    else
    {
        // Penalise matching if its the incorrect Car Type
        hits--;
    }

    if ((other.cylinders == cylinders))
    {
        hits++;
    }
    if ((other.lpg == lpg))
    {
        hits++;
    }
    return((float)hits/3);
}

/**
 * Converts the class fields into a string for display.
 */
public String toString()
{
    return(carType+" "+cylinders+" "+lpg);
}

/**
 * Function that is used by the class to initialise its internal
 * fields.

```

```

    * @param data Vector of data items to be used in initialisation.
    *           Ordering is assumed to be related to the order of
    *           declaration of member fields.
    */
public void setData(Vector data)
{
    int i=0;

    carType=(String) data.elementAt(i++);

    Integer s = new Integer((String)data.elementAt(i++));
    cylinders=(int) s.intValue();

    if(((String)data.elementAt(i++)).lastIndexOf("true")!=-1)
        lpg=true;
    else
        lpg=false;
}
}

```

Appendix C

Prototype Trader Trace Output

This Appendix includes the trace output from sample runs of Interworking Traders as discussed in Chapter 8. In this example, the Importer Queried trader_1 for the following:

```
service=pizza,  
type=the_lot,  
size=12,  
delivery=true,  
hop_count=3  
t_id = random
```

using the following Trading interconnection topology:

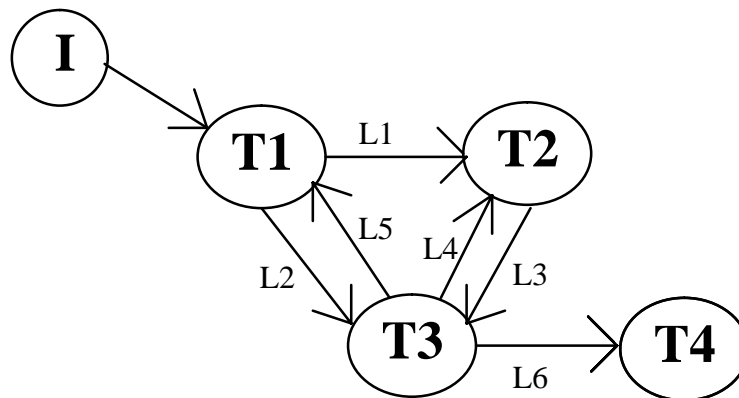


Fig. C.1: Four linked Traders

C.1 OfferSpace

```
In the Offer Space Object  
OfferSpaceServer bound in registry  
temp=Used_Cars  
Adding [sports, 6, true]  
tempOffer=trader_1 Used_Cars australian_motors sports 6 true 0  
Creating OfferSpace element for trader_1  
temp=Pizza  
Adding [vegetarian, 16, false]  
tempOffer=trader_1 Pizza pedro's_pizza vegetarian 16 false 0
```

```

Found existing OfferSpace element for trader_1
temp=Pizza
Adding [vegetarian, 10, false]
tempOffer=trader_1 Pizza pizza_haven vegetarian 10 false 0
Found existing OfferSpace element for trader_1
Initialised Offer Space
temp=Used_Cars
Adding [sports, 8, true]
tempOffer=trader_2 Used_Cars bob_moran sports 8 true 0
Creating OfferSpace element for trader_2
temp=Pizza
Adding [vegetarian, 10, true]
tempOffer=trader_2 Pizza pedro's_Pizza vegetarian 10 true 0
Found existing OfferSpace element for trader_2
temp=Pizza
Adding [the_lot, 16, false]
tempOffer=trader_2 Pizza pizza_haven the_lot 16 false 0
Found existing OfferSpace element for trader_2
Initialised Offer Space
temp=Used_Cars
Adding [sports, 8, true]
tempOffer=trader_3 Used_Cars pete's_cars sports 8 true 0
Creating OfferSpace element for trader_3
temp=Pizza
Adding [the_lot, 16, true]
tempOffer=trader_3 Pizza sealand_pizza the_lot 16 true 0
Found existing OfferSpace element for trader_3
temp=Pizza
Adding [marinara, 10, true]
tempOffer=trader_3 Pizza pizza_haven marinara 10 true 0
Found existing OfferSpace element for trader_3
Initialised Offer Space
temp=Used_Cars
Adding [Wagon, 6, true]
tempOffer=trader_4 Used_Cars Toky's_Motors Wagon 6 true 0
Creating OfferSpace element for trader_4
temp=Pizza
Adding [vegetarian, 10, false]
tempOffer=trader_4 Pizza pedro's_pizza vegetarian 10 false 0
Found existing OfferSpace element for trader_4
temp=Pizza
Adding [the_lot, 12, true]
tempOffer=trader_4 Pizza pizza_haven the_lot 12 true 0
Found existing OfferSpace element for trader_4
Initialised Offer Space
Offers for trader_1
trader_1 Used_Cars australian_motors sports 6 true -1517918040
trader_1 Pizza pedro's_pizza vegetarian 16 false 1115789266
trader_1 Pizza pizza_haven vegetarian 10 false -208917030

Offers for trader_2
trader_2 Used_Cars bob_moran sports 8 true 1019800440
trader_2 Pizza pedro's_Pizza vegetarian 10 true -611652875
trader_2 Pizza pizza_haven the_lot 16 false 1362132786

Offers for trader_3
trader_3 Used_Cars pete's_cars sports 8 true 1968097058
trader_3 Pizza sealand_pizza the_lot 16 true -1933932397
trader_3 Pizza pizza_haven marinara 10 true 1442904595

Offers for trader_4
trader_4 Used_Cars Toky's_Motors Wagon 6 true 397902075
trader_4 Pizza pedro's_pizza vegetarian 10 false 875635521
trader_4 Pizza pizza_haven the_lot 12 true 64950077

Doing the search!
Searching Trader_1 for Pizza The Lot 12 true

```

Found the Trader record
Element at 0 is trader_1
trader_1 Used_Cars australian_motors sports 6 true -1517918040
trader_1 Pizza pedro's_pizza vegetarian 16 false 1115789266
trader_1 Pizza pizza_haven vegetarian 10 false -208917030

Found a match at1 Pizza=Pizza
Doing similarity on m=vegetarian 16 false s= The Lot 12 true
Total matches: 0Return value= 0.0
Similarity =0.0
Total matches: 0Return value= 0.0
Service Type OK, but not enough matches
Found a match at2 Pizza=Pizza
Doing similarity on m=vegetarian 10 false s= The Lot 12 true
Total matches: 0Return value= 0.0
Similarity =0.0
Total matches: 0Return value= 0.0
Service Type OK, but not enough matches
Finished the search!

Doing the search!
Searching Trader_2 for Pizza The Lot 12 true
Element at 0 is trader_1
trader_1 Used_Cars australian_motors sports 6 true -1517918040
trader_1 Pizza pedro's_pizza vegetarian 16 false 1115789266
trader_1 Pizza pizza_haven vegetarian 10 false -208917030

Found the Trader record
Element at 1 is trader_2
trader_2 Used_Cars bob_moran sports 8 true 1019800440
trader_2 Pizza pedro's_Pizza vegetarian 10 true -611652875
trader_2 Pizza pizza_haven the_lot 16 false 1362132786

Found a match at1 Pizza=Pizza
Doing similarity on m=vegetarian 10 true s= The Lot 12 true
Total matches: 1Return value= 0.33333334
Similarity =0.33333334
Total matches: 1Return value= 0.33333334
Service Type OK, but not enough matches
Found a match at2 Pizza=Pizza
Doing similarity on m=the_lot 16 false s= The Lot 12 true
Total matches: 0Return value= 0.0
Similarity =0.0
Total matches: 0Return value= 0.0
Service Type OK, but not enough matches
Finished the search!

Doing the search!
Searching Trader_3 for Pizza The Lot 12 true
Element at 0 is trader_1
trader_1 Used_Cars australian_motors sports 6 true -1517918040
trader_1 Pizza pedro's_pizza vegetarian 16 false 1115789266
trader_1 Pizza pizza_haven vegetarian 10 false -208917030

Element at 1 is trader_2
trader_2 Used_Cars bob_moran sports 8 true 1019800440
trader_2 Pizza pedro's_Pizza vegetarian 10 true -611652875
trader_2 Pizza pizza_haven the_lot 16 false 1362132786

Found the Trader record
Element at 2 is trader_3
trader_3 Used_Cars pete's_cars sports 8 true 1968097058
trader_3 Pizza sealand_pizza the_lot 16 true -1933932397
trader_3 Pizza pizza_haven marinara 10 true 1442904595

Found a match at1 Pizza=Pizza
Doing similarity on m=the_lot 16 true s= The Lot 12 true
Total matches: 1Return value= 0.33333334
Similarity =0.33333334

```

Total matches: 1Return value= 0.33333334
Service Type OK, but not enough matches
Found a match at2 Pizza=Pizza
Doing similarity on m=marinara 10 true s= The Lot 12 true
Total matches: 1Return value= 0.33333334
Similarity =0.33333334
Total matches: 1Return value= 0.33333334
Service Type OK, but not enough matches
Finished the search!
Doing the search!
Searching Trader_4 for Pizza The Lot 12 true
Element at 0 is trader_1
trader_1 Used_Cars australiano_motors sports 6 true -1517918040
trader_1 Pizza pedro's_pizza vegetarian 16 false 1115789266
trader_1 Pizza pizza_haven vegetarian 10 false -208917030

Element at 1 is trader_2
trader_2 Used_Cars bob_moran sports 8 true 1019800440
trader_2 Pizza pedro's_Pizza vegetarian 10 true -611652875
trader_2 Pizza pizza_haven the_lot 16 false 1362132786

Element at 2 is trader_3
trader_3 Used_Cars pete's_cars sports 8 true 1968097058
trader_3 Pizza sealand_pizza the_lot 16 true -1933932397
trader_3 Pizza pizza_haven marinara 10 true 1442904595

Found the Trader record
Element at 3 is trader_4
trader_4 Used_Cars Toky's_Motors Wagon 6 true 397902075
trader_4 Pizza pedro's_pizza vegetarian 10 false 875635521
trader_4 Pizza pizza_haven the_lot 12 true 64950077

Found a match at1 Pizza=Pizza
Doing similarity on m=vegetarian 10 false s= The Lot 12 true
Total matches: 0Return value= 0.0
Similarity =0.0
Total matches: 0Return value= 0.0
Service Type OK, but not enough matches
Found a match at2 Pizza=Pizza
Doing similarity on m=the_lot 12 true s= The Lot 12 true
Total matches: 2Return value= 0.6666667
Similarity =0.6666667
Total matches: 2Return value= 0.6666667
Service Type OK and properties match!!
Adding Pizza pizza_haven the_lot 12 true
Finished the search!

```

C.2 LinkSpace

```

In the Link Space Object
LinkSpaceServer bound in registry
tempLink=L1 LUI_2 always always
Creating linkSpace element for trader_1
tempLink=L2 LUI_3 always always
Found existing linkSpace element for trader_1
tempLink=L3 LUI_3 always always
Creating linkSpace element for trader_2
tempLink=L4 LUI_2 always always
Creating linkSpace element for trader_3
tempLink=L5 LUI_1 always always
Found existing linkSpace element for trader_3
tempLink=L6 LUI_4 always always
Found existing linkSpace element for trader_3
printLinks() for trader_1

```

```

L1 LUI_2 always always
L2 LUI_3 always always

printLinks() for trader_2
  L3 LUI_3 always always

printLinks() for trader_3
  L4 LUI_2 always always
  L5 LUI_1 always always
  L6 LUI_4 always always

Returning Links for Trader_1

  [L1 LUI_2 always always, L2 LUI_3 always always]

Returning Links for Trader_2

  [L3 LUI_3 always always]

Returning Links for Trader_3

  [L4 LUI_2 always always, L5 LUI_1 always always, L6 LUI_4 always always]

Returning Links for Trader_3

  [L4 LUI_2 always always, L5 LUI_1 always always, L6 LUI_4 always always]

Returning Links for Trader_4

```

C.3 Traders

C.3.1 Trader 1

```

In the Trading Object
Searching for OS and LS Objects!
Found them!
Config file = trader1.cfg
LUI_1 bound in registry
h_c=3 Started a new Query from Importer:Pizza The Lot 12 true 3 always Importer 222
true
Parameters checked!
Security checked!
h_c=3 Seen list = []
h_c=3 Just added 3 always Importer 222 true
h_c=3 Duplicate = false
h_c=3 IP=3 always Importer 222 true
h_c=3 Unified = always
h_c=3 Getting Local Offers
Answer = []
h_c=3 Querying Linked Traders
h_c=3 Getting Links
h_c=3 Links obtained:[L1 LUI_2 always always, L2 LUI_3 always always]
h_c=3 Querying Linked Traders!
h_c=3 Linking: originator=Importer, target link.LUI=LUI_2
h_c=0 Started a new Query from LUI_3:Pizza The Lot 12 true 0 always Importer 222
true
Parameters checked!
Security checked!
h_c=0 Seen list = [3 always Importer 222 true]
h_c=0 Compare existing record:Importer 222, New record:Importer 222 Answer is true
h_c=0 Returning duplicate Policy=3 always Importer 222 true
h_c=0 Duplicate = true

```

```
h_c=0 Found a duplicate message. dupPolicy=3 always Importer 222 true
h_c=0 No Link Search Reqd : Returning Empty offers List!
h_c=3 Linking: originator=Importer, target link.LUI=LUI_3
h_c=3 Returning =[Pizza pizza_haven the_lot 12 true]
```

C.3.2 Trader 2

```
In the Trading Object
Searching for OS and LS Objects!
Found them!
Config file = trader2.cfg
LUI_2 bound in registry
h_c=2 Started a new Query from LUI_1:Pizza The Lot 12 true 2 always Importer 222
true
Parameters checked!
Security checked!
h_c=2 Seen list = []
h_c=2 Just added 2 always Importer 222 true
h_c=2 Duplicate = false
h_c=2 IP=2 always Importer 222 true
h_c=2 Unified = always
h_c=2 Getting Local Offers
Answer = []
h_c=2 Querying Linked Traders
h_c=2 Getting Links
h_c=2 Links obtained:[L3 LUI_3 always always]
h_c=2 Querying Linked Traders!
h_c=2 Linking: originator=LUI_1, target link.LUI=LUI_3
h_c=2 Returning =[Pizza pizza_haven the_lot 12 true]
h_c=1 Started a new Query from LUI_3:Pizza The Lot 12 true 1 always Importer 222
true
Parameters checked!
Security checked!
h_c=1 Seen list = [2 always Importer 222 true]
h_c=1 Compare existing record:Importer 222, New record:Importer 222 Answer is true
h_c=1 Returning duplicate Policy=2 always Importer 222 true
h_c=1 Duplicate = true
h_c=1 Found a duplicate message. dupPolicy=2 always Importer 222 true
h_c=1 No Link Search Reqd : Returning Empty offers List!
```

C.3.3 Trader 3

```
In the Trading Object
Searching for OS and LS Objects!
Found them!
Config file = trader3.cfg
LUI_3 bound in registry
h_c=1 Started a new Query from LUI_2:Pizza The Lot 12 true 1 always Importer 222
true
Parameters checked!
Security checked!
h_c=1 Seen list = []
h_c=1 Just added 1 always Importer 222 true
h_c=1 Duplicate = false
h_c=1 IP=1 always Importer 222 true
h_c=1 Unified = always
h_c=1 Getting Local Offers
Answer = []
h_c=1 Querying Linked Traders
h_c=1 Getting Links
h_c=1 Links obtained:[L4 LUI_2 always always, L5 LUI_1 always always, L6 LUI_4 always
always]
h_c=1 Querying Linked Traders!
h_c=1 Skipping bi-directional Link to LUI_2
```

```

h_c=1 Linking: originator=LUI_2, target link.LUI=LUI_1
h_c=1 Linking: originator=LUI_2, target link.LUI=LUI_4
h_c=1 Returning =[Pizza pizza_haven the_lot 12 true]
h_c=2 Started a new Query from LUI_1:Pizza The Lot 12 true 2 always Importer 222
true
Parameters checked!
Security checked!
h_c=2 Seen list = [1 always Importer 222 true]
h_c=2 Compare existing record:Importer 222, New record:Importer 222 Answer is true
h_c=2 Returning duplicate Policy=1 always Importer 222 true
h_c=2 Duplicate = true
h_c=2 Found a duplicate message. dupPolicy=1 always Importer 222 true
h_c=2 IP hc=2 dupPolicy hc=1
h_c=2 IP=2 always Importer 222 true
h_c=2 Unified = always
h_c=2 Skipping local search since we did it before!
h_c=2 Querying Linked Traders
h_c=2 Getting Links
h_c=2 Links obtained:[L4 LUI_2 always always, L5 LUI_1 always always, L6 LUI_4 always
always]
h_c=2 Querying Linked Traders!
h_c=2 Linking: originator=LUI_1, target link.LUI=LUI_2
h_c=2 Skipping bi-directional Link to LUI_1
h_c=2 Linking: originator=LUI_1, target link.LUI=LUI_4
h_c=2 Returning =[]

```

C.3.4 Trader 4

```

In the Trading Object
Searching for OS and LS Objects!
Found them!
Config file = trader4.cfg
LUI_4 bound in registry
h_c=0 Started a new Query from LUI_3:Pizza The Lot 12 true 0 always Importer 222
true
Parameters checked!
Security checked!
h_c=0 Seen list = []
h_c=0 Just added 0 always Importer 222 true
h_c=0 Duplicate = false
h_c=0 IP=0 always Importer 222 true
h_c=0 Getting Local Offers
Answer = [Pizza pizza_haven the_lot 12 true]
h_c=0 Returning =[Pizza pizza_haven the_lot 12 true]
h_c=1 Started a new Query from LUI_3:Pizza The Lot 12 true 1 always Importer 222
true
Parameters checked!
Security checked!
h_c=1 Seen list = [0 always Importer 222 true]
h_c=1 Compare existing record:Importer 222, New record:Importer 222 Answer is true
h_c=1 Returning duplicate Policy=0 always Importer 222 true
h_c=1 Duplicate = true
h_c=1 Found a duplicate message. dupPolicy=0 always Importer 222 true
h_c=1 IP hc=1 dupPolicy hc=0
h_c=1 IP=1 always Importer 222 true
h_c=1 Unified = always
h_c=1 Skipping local search since we did it before!
h_c=1 Querying Linked Traders
h_c=1 Getting Links
h_c=1 Links obtained:null
h_c=1 Querying Linked Traders!
h_c=1 Returning =[]

```

C.4 Importer Applet

```
Selected Pizza
Changed the labels:Mode=Pizza
Selected The Lot
Selected 12
Selected 3
Got Submit
Searched for:LUI_1
Service Type: Pizza
Prop 1: The Lot
Prop 2: 12
Prop 3: True
Follow Policy: always
Hop Count: 3
Request ID: Random
Did a Random no222
PizzaProps=The Lot 12 true
ServiceProps=The Lot 12 true
Sending:sPizza The Lot 12 true i:3 always Importer 222 true
Got Exit
```

Appendix D

Some thoughts on Trading Applications and Topologies

This appendix includes a discussion of a number of prospective layers of application domains for service Trading. The discussion includes an investigation of the application of Trading to the domain of E-commerce in the capacity of providing a resource location service. In addition, a number of new Trader interconnection topologies which facilitate global interconnection of Traders are proposed.

D.1 Layers of Trading

There are three levels of system software which may benefit from the use of a Trader to provide service brokering, as shown in figure D.1 [30]. At the lowest level, load balancing may be performed by a Distributed Operating System (DOS). In order to provide advanced telecommunications services such as video and audio streams, a distributed computing solution known as the Telecommunications Information Networking Architecture (TINA) [114] is being developed by the TINA Consortium which is comprised of many global Telecommunication Companies.

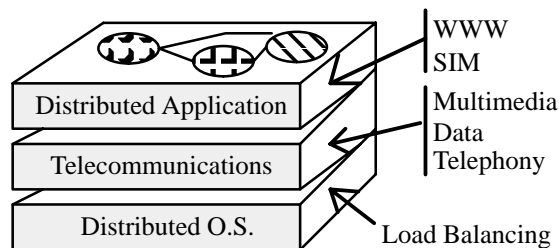


Fig. D.1: Three Layers of Trading

At the topmost level, E-commerce applications such as WWW commercial activities may utilise a Trader for Web site location [26]. Fig. D.1 illustrates three layers of object-based

software which could utilise Trading. In the following sections, each of these layers of Trading are discussed.

D.1.1 Load Balancing in Distributed Operating Systems

A Distributed Operating System (DOS) is required to manage the resources of many computers which are cooperating and sharing resources. It is desirable to develop a mechanism which allows load balancing to improve system performance. One solution is to make use of interworking Traders to assist the DOS when it performs process migration [115].

The user of a workstation may export the workstation's resources to a Trader, making them available for utilisation by non-local processes. Normally, this would occur during periods of low utilisation such as weekends, or after hours. It is important for the user to maintain control over the workstation's availability for Trading since the user will experience performance reduction as utilisation increases. In return for participation, the workstation user may receive improved bandwidth allocation or an increased disk quota as compensation.

Trading provides the DOS with a mechanism to broker computational resources residing in nodes of the system. Properties used in the Trading may include the number of processes currently running, free memory, I/O usage and communication bandwidth of the node. This is an example of Trading at a low level of system software and may be applied to any distributed computing system.

Sun Microsystems has developed the Spring DOS which is object-oriented and aims to make significant improvements upon Unix [116]. As in all object-based systems, the naming and location of objects is an essential service which the DOS must provide.

D.1.2 Telecommunications Information Networking Architecture

TINA is based upon technology standards such as RM-ODP, Intelligent Networks (IN), Telecommunications Management Networks (TMN), Asynchronous Transfer Mode (ATM) and CORBA [114]. It aims to provide an "open" architecture which addresses the needs of existing voice telephony and also allows for new services such as interactive multimedia services including video conferencing. It will allow telecommunications service providers to create "high level" services which make use of lower level services.

TINA's aim is to provide a Distributed Processing Environment (DPE) which considers services and management functions to be software entities operating on a platform independent basis, thereby allowing distribution of objects and high levels of software re-use. TINA uses an IDL which is based on the RM-ODP IDL but defines an additional interface type to allow streams of data flow in addition to operation invocations. Thus, TINA-IDL supports both Stream and Operational Interfaces [114].

In order to quickly introduce new services, a service advertising/matching capability is clearly beneficial. Services may be defined as *building blocks* which are entities containing more than one object. Building blocks are considered to be one unit of distribution, with all objects residing

in the same node. Objects within the building block may offer services in the form of a “contract” to the outside world. It is these contracts which are brokered by the Trader [113].

For example, a Multimedia stream building block may need to locate a Video stream building block and an Audio stream building block in order to provide a service. It may do so dynamically through the use of a Trader. The telecommunications system can be considered to be a distributed computing system and hence, a middle layer Trading application domain.

D.1.3 Open Distributed Applications

At the highest level, applications which operate on top of a DOS utilise the network services of TINA. An example of an Open Distributed Application is the Service Interaction Manager [14] which uses a Trader to locate and access commercial service sites on a user’s behalf. In this capacity, the Trader acts like an Electronic Yellow Pages Service, accepting a request for a particular type of commercial service and returning a list of matching offers.

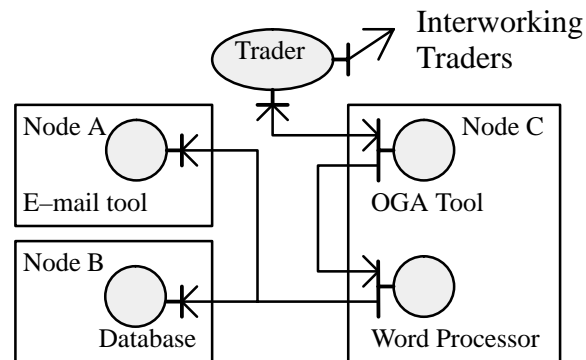


Fig. D.2: Object-Based Desktop Applications Interworking

On a smaller scale, applications such as desktop productivity tools may be implemented in a distributed manner. This would mean that applications are themselves considered objects, and capable of providing a service to clients.

As an example, consider a user’s desktop where an email tool, a database application, an Occurrence Graph Analysis tool (OGA) and a word processor are in use. Having performed Petri Net analysis with the OGA Tool, the user may instruct the package to use the word processor as a client and download the recent results to a new document. Having used the word processor to add comments to the new results, the user may decide to store the results in a database and e-mail a colleague regarding the recent work. Using the word processor as the Importer (using the other applications as servers), data may be written to the database and e-mailed as required. The word processor would use a Trader to dynamically locate the Exporting objects which may be running on different nodes in the Distributed System. A Trader operating using Microsoft’s Distributed Common Object Model (DCOM) has been proposed [40]. Such a Trader would make it easy for desktop applications to utilise each other’s services in a corporate local area network (LAN) environment.

With applications as distributed objects, data interchange and manipulation between packages may be accomplished simply by an object method invocation. In such systems, all application packages are seamlessly integrated and platform independent.

D.1.4 Trading in Mobile Environments

There are a number of possible applications of an Object Trading service in Mobile Environments, for both clients and servers [34].

D.1.4.1 Re-binding of Services

As client applications move throughout the mobile environment, there is a need for currently utilised services to be re-bound to appropriate servers located in the client's new domain [55]. This may be automatically accomplished by the Trader if it is notified of an impending domain transition by the mobile environment. With automatic service re-binding, it is as if the client has traded for a continuous service, rather than a one time service location request, since the system automatically re-binds to new servers as required.

Similarly, when users of a mobile multimedia system [54] move, they notify the Trader of their new location at a workstation (analogous to an active badge). This allows the system to track them and move software objects to the appropriate physical location.

In the case where servers are mobile [56], such as when there is a need to access data from other terminals in the mobile environment, there is a need to re-bind clients to other available servers when servers become unavailable. When servers move, it would be possible to re-route requests to its new location but this would involve network overheads.

D.1.4.2 Changing Quality of Service

Another factor which is important in mobile environments is the fluctuating quality of service (QoS) [57]. Since it is possible for the QoS at a mobile server to degrade to such an extent that it becomes too low, a client may use a Trader in an attempt to locate a server with a better QoS. This is an important factor in maintaining a reliable mobile service for clients.

D.1.4.3 Process Migration for load balancing

As clients move between domains, it is possible for servers to become overloaded and cause a degradation in their QoS. In such a situation, Trading may be applied at a lower level, where servers use a Trader to perform load balancing or process migration as discussed in section D.1.1.

With the emergence of Java as a distributed, platform independent, interpreted programming language, it becomes possible for the simple migration of Java applications, since the code will run on any platform which has a Java Virtual Machine. Examples of tools for creating mobile Java Agents include ObjectSpace's Voyager [130], which is an extension of Java RMI and General Magic's Odyssey [129], which is CORBA compatible. An overview of Java Mobile Agent technologies is given in [131].

D.2 Trader Organisation Topologies

The Interworking protocol of the Trader has been designed to allow arbitrarily complex interconnection between Traders. It is likely that interworking topologies will depend on the scale of Trading. This section proposes and discusses several original topologies for the organisation of Traders.

D.2.1 Server Farms

In the server farm topology, multiple Traders are tightly coupled and connected in a tree formation using a high speed LAN as shown in figure D.3. The Traders operate in specific *service domains* determined by factors such as regional boundaries, product type or any criteria that permits logical partitioning of the entire offer space. The top-most Trader acts as the main or root Trader, accepting all Queries from Importers. Based upon the service type, the root Trader forwards Queries to lower level Traders that it has links to and which specialise in certain sections of the offer space. Thus, it is important to allow the Trader to dynamically determine which links to follow, based upon the context of the Query being serviced.

An example of this topology is given in figure D.3, where the Importer submits a Query to the Root_Trader indicating the service to be matched. The Root_Trader then forwards the Query, adding the `starting_trader` Query parameter (discussed in section 3.4.8.2), which, given a path name to it, allows the search to commence at a specific Trader. A Query for a `used_car` service would require `Retailers.Motor_Vehicles.Used` to be used as a path to the starting Trader, as shown with dashed arrows in figure D.3.

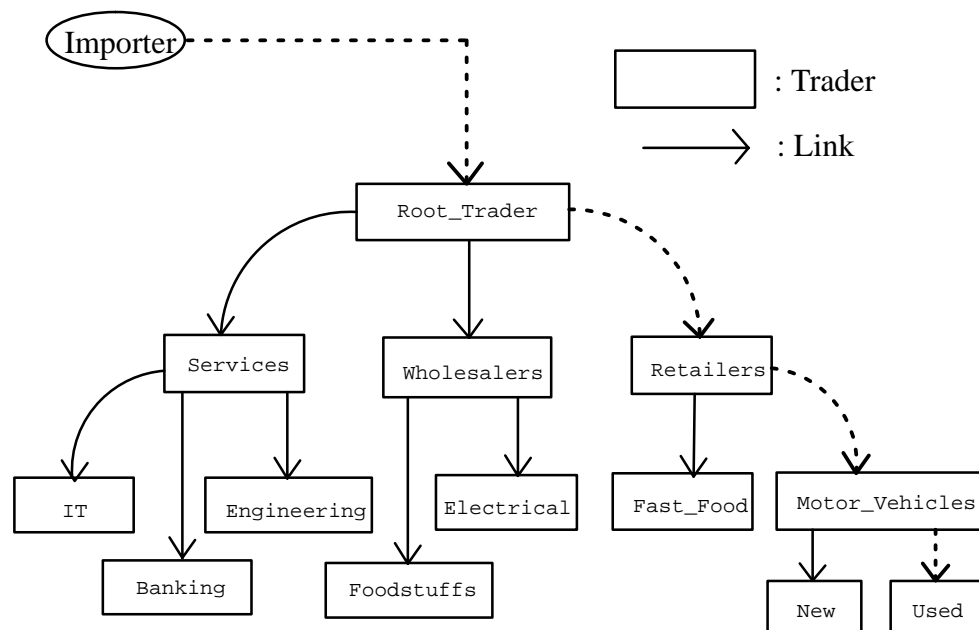


Fig. D.3: Server Farm using a Trading path

The benefits of this topology are evident through parallel independent servicing of Queries by multiple Traders. When the Trader farm is privately owned (e.g. as a commercial service advertising agency), it would need to manage millions of Offers and would be subjected to very

high connection rates from Importers. In such a situation, performance is an important factor in ensuring usability. This topology allows the power of domain-specific Traders to be utilised in parallel and removes the possibility of forwarding Queries to Traders which do not manage relevant offers.

D.2.2 Planes of Linked Traders

In this topology, Traders utilise a multi-layered approach to linking, thereby providing layering of interworking Traders. A Trader may operate at a local level, such as on a *city*-wide basis as shown in figure D.4.

An example of this is a Trader that is used for advertising commercial services on behalf of shopping centres, local businesses and other service providers (e.g. *Sea-side_Trader*). It is also possible for shopping centres to manage their own Trader, depending upon the scale of the operation (e.g. *Mall_Trader* for a large Shopping Mall complex with hundreds of shops). City-level Traders can be linked to each other on a peer-to-peer basis, resulting in a city-wide Trading layer.

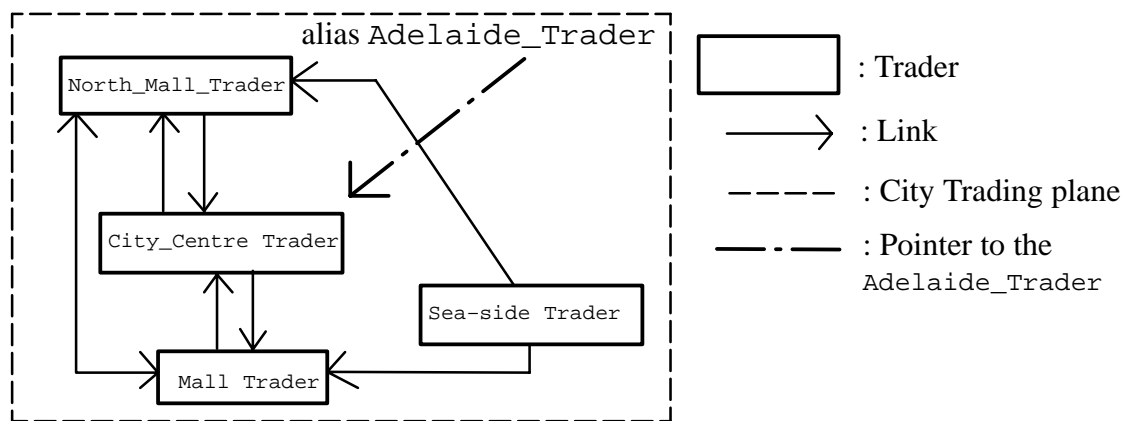


Fig. D.4: City-level Traders

On a *national* level (shown in figure D.5), city-level Traders are connected in a mesh, where each city-level Trader is linked to all other city Traders. This allows Importers to locate services on an inter-city scale.

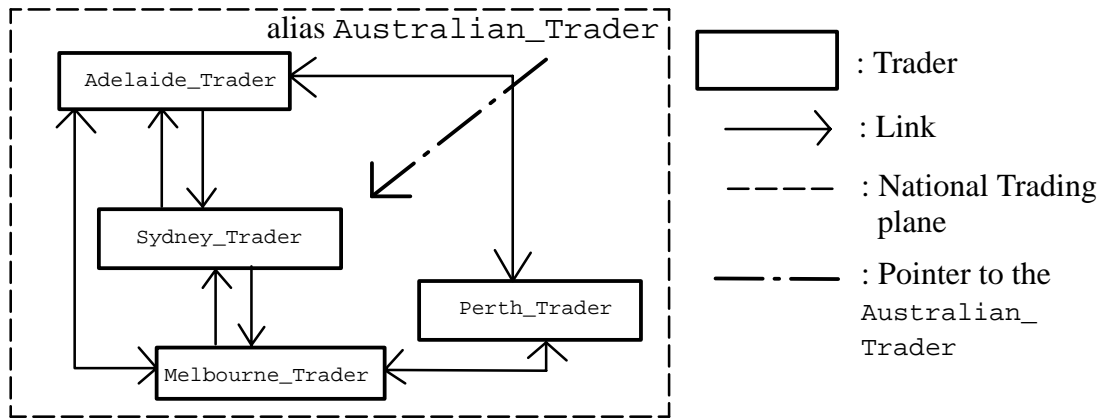


Fig. D.5: National-level Traders

At the top-most *international* level (shown in figure D.6), Traders may be connected to their national level Trader that is nominated as the *root* Trader for a country. Each country could designate a Trader to perform the root Trader role as described above. Such an organisation topology allows Queries to be directed to a specific Trader that processes the request as normal.

Using this topology, Queries may be performed locally, nationally, or internationally, depending upon the Trading link plane that is used when submitting the Query.

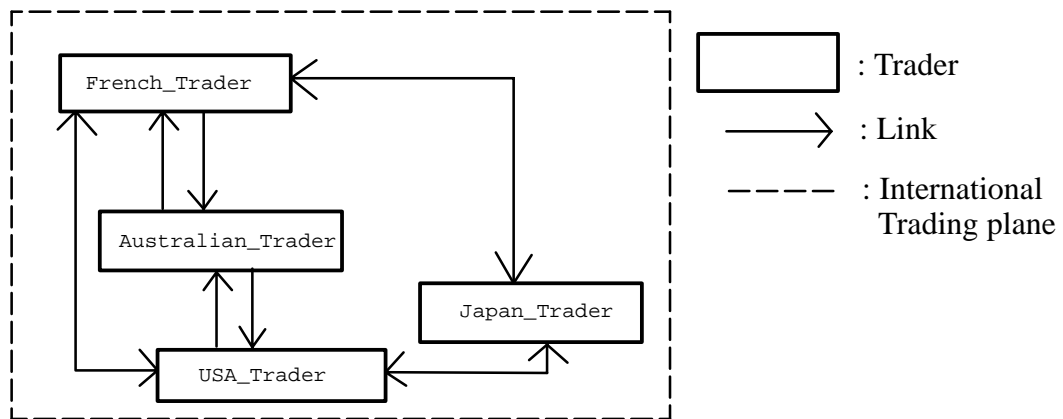


Fig. D.6: International-level Traders

D.2.3 Examples

D.2.3.1 Local level Trading

Consider a motor vehicle driver who is getting low on fuel and decides to obtain more fuel. By submitting a request to the city-level Trader (using a mobile computer system), the driver is able to locate the “best” fuel pump station based upon price, affiliation and physical location. The request remains at a *city* level since the service **must** be provided locally.

D.2.3.2 National level Trading

A Query is made for a VOD service (discussed in section 2.2.2) that is capable of providing a specific video title at a specified quality of service (QoS). In such a scenario, the Query is

forwarded to the national Trader, as the request will be forwarded on an *inter-city* scale. The consumer does not care about the location of the service provider, and may decide to choose the provider that minimises cost and/or maximises QoS. This can be specified by the Importer in the initial Query. International VOD providers are not considered in the request since it would require huge amounts of international bandwidth to provide the service. In some scenarios, it may be necessary to Trade on an *international* level in order to find domain-specific titles (e.g. original 1970's Hong Kong action movies).

D.2.3.3 International level Trading

Consider a consumer who desires a specific bottle of rare wine. The product is not required immediately, but good pricing and service are important. An appropriate Query would be issued to the designated Australian *international* Trader for forwarding. The search would propagate internationally until it stopped (though a `hop_count` limitation) having located a wine merchant that can provide the requested bottle at a good price and deliver it promptly. It may be that the “best” service provider is located in Sweden. This is an example of using the *international* plane of Trader links, where the Query hops from *city* → *national* → *international* and down through the tree of Traders again in a globally concurrent manner (multiple concurrent Queries at each plane of Trading).

D.3 Summary

In this appendix, the concept of applying Traders to the problem of locating services in an electronic commerce environment has been presented. A new topology for the interconnections of multiple Traders operating at different levels of abstraction has also been presented. The concepts raised in this appendix are areas for further research.

Appendix E

Publications

E.1 Conferences and Workshops

1. A. Tokmakoff and J. Billington, “**Consumer Services in Smart City Adelaide**”, Proc. of the 1st International Workshop on Home Oriented Informatics, Telematics and Automatics, Copenhagen, July 1994, pp. 201–210.

Introduction: The Smart City Adelaide (SCA) concept aims to define the infrastructure and applications which are required to provide an electronic medium for provision of digital metropolitan services. It is envisaged that these services will be offered to the Multi Function Polis (MFP) and metropolitan areas of “Smart City” Adelaide, the capital city of South Australia, Australia. It is anticipated that the applications within SCA will include on–line trading, remote learning and Automatic Meter Reading (AMR) which is a telemetry service. After discussing the background of the MFP Project and its key themes and aims, related projects under way worldwide will be mentioned, and the MFP’s position relative to these projects will be explored. Details of the possible services to be provided will be given, with both short and long term goals discussed. The technologies and techniques to be considered when developing SCA will then be identified, with particular reference to formal specification and modelling methods. Important aspects to be considered when designing SCA are discussed. In particular, user interface requirements, specific services which may be supplied, user access methods and consistency across applications will be mentioned.

2. A. Tokmakoff & J. Billington, “**Coloured Petri Net Modelling of the ODP Trader for use in Resource Discovery**”, Proc. of the 1st International Workshop on High Speed Networking and Open Distributed Platforms, St. Petersburg, Russia, Session 6, Paper 1, 13–15 June 1995.

Abstract: As high speed networking becomes more widespread and increasingly accessible from home, a huge variety of new vendors and services will become available to consumers. These services will fall into three main categories: information services, shopping and entertainment. One of the major problems facing an increasingly information–rich society is how to quickly and conveniently locate information which is provided by different servers throughout a large distributed system. The Reference Model for Open Distributed Processing (RM–ODP) deals with the problem of locating services/resources in a heterogeneous Open Distributed System through the standardisation of a Trading function which is realised by a Trader. In order to increase our understanding of Trader’s dynamic behaviour, a CPN model has been created. The Trader’s Computational Viewpoint has been

modelled using a Hierarchy of CPNs which allow functional blocks to be modelled separately and their inter-relationships defined. A discussion of the model and techniques for its analysis are presented.

3. A. Tokmakoff & J. Billington, “**Modelling of the ODP Trader for use in Resource Discovery**”, Proc. of the 2nd IEEE Workshop on Community Networking, Princeton, N.J., U.S.A., 20–22 June 1995, IEEE, ISBN:0–7803–2756–X, pp. 155–162.

Abstract: As information technology expands from the workplace to the home, a huge variety of new vendors and services will become available to consumers. These services will fall into three main categories: information services, shopping and entertainment. One of the major problems facing an increasingly information-rich society is how to quickly and conveniently locate information which is provided by different servers throughout a large distributed system. The Reference Model for Open Distributed Processing (RM-ODP) deals with the problem of locating services/resources in a heterogeneous Open Distributed System through the standardisation of a Trading function which is realised by a Trader. Although initially intended for dynamic binding of computational resources in a distributed system, it is apparent that the Trading function has applications in Community Networking for service/resource location. The authors are interested in validating the Trader standard using Coloured Petri Nets (CPNs) and investigating Trader’s applicability in the provision of advanced information services.

4. A. Tokmakoff & J. Billington, “**CPN Modelling of an Object Based System : The ODP Trader**”, Proc. of the 1st IFIP International Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS’96), Paris, France, March 1996, ISBN:0–412–79770–4, Chapman and Hall, pp. 245–260.

Abstract: Advances in computing power and networking technologies are leading the way to a new generation of information systems, where heterogeneous computers share resources and cooperate in a distributed manner. Such systems are inherently complex due to concurrency and synchronisation of activities operating in parallel, making design and implementation much more challenging than yesterday’s stand-alone personal computers. In order to engineer these Open Distributed Systems, tools are required to aid in the design and analysis processes. In this paper, Coloured Petri Nets and the Design/CPN tool are used to model the ODP Trader which has been recently standardised by ISO and ITU-T. The Trader is presented in an electronic commerce environment, where interworking of Traders is demonstrated.

Keywords: Coloured Petri Nets, ODP Trader, Electronic Commerce.

5. A. Tokmakoff & J. Billington “**Modelling of Object Based Systems with Coloured Petri Nets**”, Proc. of the IASTED International Conference on Modelling, Simulation and Optimisation, Gold Coast, Australia, 6–9 May 1996, Published on CD-ROM (The International Association for Science and Technology), ISBN:0–88986–197–8.

Abstract: The Reference Model for Open Distributed Processing (RM-ODP) defines an architecture which allows heterogeneous software components to interact, where entities in the system are Objects. In RM-ODP, entire applications may be considered to be Objects. In order to allow dynamic location of services/resources, a Trading function has been defined which is performed by a Trader instance. In this paper, a Coloured Petri Net model of the RM-ODP Trader and its environment is presented. In addition, the software package Design/CPN[™] used in the modelling of Trader is described in detail.

Keywords: Modelling, Petri Nets, RM-ODP Trader

6. A. Tokmakoff & J. Billington, “**Service Brokering in Object Based Systems : Advanced Information Services**”, Proc. of the 3rd IEEE Workshop on Community

Networking, Antwerp, Belgium, 23–24 May 1996, IEEE Press, ISBN:0–7803–3304–7, pp. 43–48.

Abstract: Open Object–Based Systems are emerging as the underlying foundation for the design of large scale distributed applications. As an integral component of the ISO/ITU–T Reference Model for Open Distributed Processing, service brokering may be applied at multiple layers of the software hierarchy, spanning Distributed Operating Systems, through Telecommunication service provision up to the Application domain. This paper presents brokering of services as an essential service in three important application domains which require dynamic location of objects. In addition, it investigates CORBA and Java as possible implementation technologies.

7. A. Tokmakoff & J. Billington, “**A Coloured Petri Net model of an Open Object–based System**”, Proc. of the 2nd International Workshop on Object–Oriented Programming and Models of Concurrency, Osaka, Japan, 24 June 1996, pp. 104–118.

Abstract: Open Object–based systems are becoming increasingly used as the basis for design of distributed systems. These systems are inherently complex due to concurrency and synchronisation of activities operating in parallel. In order to engineer these Open Distributed Systems, tools are required to aid in the design and analysis processes. In this paper, Coloured Petri Nets and the Design/CPN™ tool are used to model the ODP Trader which has been recently standardised by ISO and ITU–T. The Trader is an integral component of an object–based system because it provides a dynamic resource location service. A Coloured Petri Net model of the Trader is presented in an electronic commerce environment, where interworking of Traders is demonstrated.

Keywords: Coloured Petri Nets; ODP Trader; Object–based System.

8. A. Tokmakoff & J. Billington, “**Using Coloured Petri Nets to aid the design of Object–based Systems**”, Proc. of the 1996 IEEE International Conference on Systems, Man and Cybernetics (SMC’96), IEEE, Beijing, China, 14–17 October 1996, ISBN:–7803–3280–6, Volume 4, pp. 3027–3032.

Abstract: As middleware and networking technologies improve, heterogenous distributed systems are expected to become commonplace. An Object–based approach to designing these systems has been adopted by ISO and ITU–T in their standardisation of the Reference Model for Open Distributed Processing (RM–ODP). An important infrastructure Object within these systems is the recently standardised Trader, which provides Objects with a dynamic resource location service. In order to engineer reliable Open Object–based Distributed Systems, it is important to perform modelling and analysis as part of the design process. In this paper, we model the Trader using Coloured Petri Nets and the Design/CPN™ tool. The model is described, and verification of the model using simulation and Occurrence Graph Analysis is presented.

Keywords: Modelling, Simulation, Distributed Processing, Petri Nets

9. J. Billington, & A. Tokmakoff, “**Petri nets and Traders: Enabling technologies for virtual enterprises**”, Presented at the IEEE 6th Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE’97), Boston, U.S.A., 18–20 June 1997.

Abstract: The development of the necessary information infrastructure required to support the rapid establishment of business partnerships for the life time of a project is seen to be an important activity by many countries. Consortia established for the life–time of a project are known as virtual enterprises. This paper discusses some of the work being undertaken in two International Standards subcommittees, and shows how they are relevant to the development of the infrastructure required for virtual enterprises. The first area is that of Open Distributed Processing (ISO/IEC JTC1/SC21/WG7) and particularly the use of Traders to locate potential business partners and the resources required within the consortium to undertake the project. The second area is that of techniques to

specify and design systems (and business processes) and formats for their interchange between design groups, being undertaken in ISO/IEC JTC1/SC7/WG11. It is seen that the WG11 standards will facilitate distributed design teams.

Keywords: Virtual Enterprises, International Standards, Traders, High-level Petri Nets, Modelling, Information Infrastructure, Distributed Design.

10. A. Tokmakoff & J. Billington “Service Trading in Mobile Environments”, Proc. of the 1st International Conference on Information, Communications and Signal Processing: Trends in Information Systems Engineering and Wireless Multimedia Communications (ICICS’97), Singapore, 9–12 September 1997, Volume 1 of 3, ISBN:0–7803–3676–3, pp. 417–421.

Abstract: One of the most important considerations for applications operating in a mobile environment is the ability to dynamically adjust to changing resource availability. With a dynamic resource location service as part of the infrastructure, mobile applications are able to re-bind to service providers as they move within the mobile environment and as available services change. As part of the ISO/IEC and ITU-T standardisation of the Reference Model for Open Distributed Processing (RM-ODP), a Trading function has been identified. It is an important infrastructure service that allows clients to locate services using service types and associated parameters as selection criteria. In order to engineer reliable Open Object-based Distributed Systems, it is important to perform modelling and analysis as part of the design process. This paper introduces the reader to the concept of Service Trading and indicates how it may be utilised by applications in a mobile environment. In addition, Coloured Petri Nets are shown to be useful for the modelling and verification of distributed mobile applications.

Keywords: Modelling, ODP Trader, Object-based Systems, Petri Nets

10. A. Tokmakoff & J. Billington “Reachability Analysis of the ODP Trader using Equivalence Classes”, Proc. of the 1998 IEEE Conference on Software Engineering: Education and Practice, Dunedin, New Zealand, 26–29 January, 1998, pp. .

Abstract: As part of the ISO/IEC standardisation of the Reference Model for Open Distributed Processing (RM-ODP), a Trading function has been identified. It is an important infrastructure service that allows clients to locate services using service types and associated parameters as selection criteria. This paper introduces the reader to the concept of Service Trading and presents a Coloured Petri Net model of the trader’s object interactions. The model is analysed using Occurrence Graphs with Equivalence Classes which results in a significantly reduced equivalent Occurrence Graph.

Keywords: Modelling, ODP Trader, Object-based Systems, Petri Nets

E.2 Journals

A. Tokmakoff & J. Billington, “**Coloured Petri Net Modelling of the ODP Trader for use in Resource Discovery**”, Computer Communications Journal, Volume 19, Number 1 Elsevier Science B.V., January 1996, ISSN:0140–3664, pp. 39–48.

(Revised version of paper 2 presented at the HSN-ODP Conference).

Appendix F

CD containing Source code and CPN Trader model

This Appendix includes the CD which contains the Trader source code and the CPN model.

F.1 Design/CPN Trader Model

The Trader model was developed using Design/CPN V3.04 and the OEOS 1.01 Library on the Linux operating system. It is suggested that the reader obtain a licence for Design/CPN [83] and perform simulations and/or analysis using the Trader model.

F.2 Trader Source Code

In order to utilise the Trader source code, the reader is referred to Chapter 8 which provides details regarding compilation, configuration and execution of the Trader Electronic Commerce test-bed. It is suggested that the reader obtain a copy of the latest JDK1.1 [122] release.

F.3 Directory Listing

```
total 6
drwxr-xr-x  5 andrewt  users      1024 Mar 30 07:34 ./
drwxr-xr-x  7 andrewt  users      1024 Mar 30 01:27 ../
-rw-r--r--  1 andrewt  users        528 Mar 22 20:43 README.cdrom
drwxr-xr-x  5 andrewt  users      1024 Mar  2 23:17 ch5/
drwxr-xr-x  9 andrewt  users      1024 Mar 30 07:34 ch7/
-rw-r--r--  1 andrewt  users         0 Mar 30 07:35 index.txt
drwxr-xr-x  2 andrewt  users      1024 Mar 22 20:42 java/

ch5:
total 10
drwxr-xr-x  5 andrewt  users      1024 Mar  2 23:17 ./
drwxr-xr-x  5 andrewt  users      1024 Mar 30 07:34 ../
```

```

drwxr-xr-x 4 andrewt users 1024 Mar 3 00:18 2_tids/
drwxr-xr-x 2 andrewt users 1024 Mar 3 00:21 3_tids_OEOS/
drwxr-xr-x 2 andrewt users 1024 Feb 24 23:39 5_tids_OEOS/
-rw-r--r-- 1 andrewt users 2074 Mar 2 23:16 equiv_be.sml
-rw-r--r-- 1 andrewt users 1661 Mar 2 23:16 equiv_mark.sml

```

ch5/2_tids:

```

total 6
drwxr-xr-x 4 andrewt users 1024 Mar 3 00:18 ./
drwxr-xr-x 5 andrewt users 1024 Mar 2 23:17 ../
drwxr-xr-x 2 andrewt users 1024 Mar 2 23:16 2_tids_OEOS/
drwxr-xr-x 2 andrewt users 1024 Feb 24 23:15 2_tids_OG/
-rw-r--r-- 1 andrewt users 1350 Mar 3 00:18 sim_report.txt

```

ch5/2_tids/2_tids_OEOS:

```

total 13189
drwxr-xr-x 2 andrewt users 1024 Mar 2 23:16 ./
drwxr-xr-x 4 andrewt users 1024 Mar 3 00:18 ../
-rw-r--r-- 1 andrewt users 672 Feb 24 22:48 2525.node
-rw-r--r-- 1 andrewt users 659 Feb 24 22:49 2684.node
-rw-r--r-- 1 andrewt users 661 Feb 24 22:48 2685.node
-rw-r--r-- 1 andrewt users 655 Feb 24 22:49 2704.node
-rw-r--r-- 1 andrewt users 133728 Feb 24 22:51 2_tids_OEOS
-rw-r--r-- 1 andrewt users 14592 Feb 24 22:49 2_tids_OEOS.DB
-rwxr-xr-x 1 andrewt users 13294592 Feb 24 22:51 2_tids_OEOS.ML*

```

ch5/2_tids/2_tids_OG:

```

total 15131
drwxr-xr-x 2 andrewt users 1024 Feb 24 23:15 ./
drwxr-xr-x 4 andrewt users 1024 Mar 3 00:18 ../
-rw-r--r-- 1 andrewt users 134760 Feb 24 22:30 2_tids_OG
-rw-r--r-- 1 andrewt users 14592 Feb 24 22:30 2_tids_OG.DB
-rwxr-xr-x 1 andrewt users 15269888 Feb 24 22:30 2_tids_OG.ML*
-rw-r--r-- 1 andrewt users 665 Feb 24 22:00 4944.node
-rw-r--r-- 1 andrewt users 665 Feb 24 22:00 4945.node
-rw-r--r-- 1 andrewt users 654 Feb 24 22:00 5060.node
-rw-r--r-- 1 andrewt users 654 Feb 24 22:00 5061.node
-rw-r--r-- 1 andrewt users 652 Feb 24 22:00 5070.node
-rw-r--r-- 1 andrewt users 652 Feb 24 22:00 5088.node
-rw-r--r-- 1 andrewt users 648 Feb 24 22:00 5151.node
-rw-r--r-- 1 andrewt users 648 Feb 24 22:00 5152.node

```

ch5/3_tids_OEOS:

```

total 10705
drwxr-xr-x 2 andrewt users 1024 Mar 3 00:21 ./
drwxr-xr-x 5 andrewt users 1024 Mar 2 23:17 ../
-rw-r--r-- 1 andrewt users 665 Feb 24 22:05 2525.node
-rw-r--r-- 1 andrewt users 9 Feb 24 22:05 2684.node
-rw-r--r-- 1 andrewt users 654 Feb 24 22:05 2685.node
-rw-r--r-- 1 andrewt users 648 Feb 24 22:05 2704.node
-rw-r--r-- 1 andrewt users 134908 Feb 24 21:58 3_tids_OEOS
-rw-r--r-- 1 andrewt users 14592 Feb 24 21:58 3_tids_OEOS.DB
-rwxr-xr-x 1 andrewt users 10760192 Feb 24 21:58 3_tids_OEOS.ML*

```

ch5/5_tids_OEOS:

```

total 10703
drwxr-xr-x 2 andrewt users 1024 Feb 24 23:39 ./
drwxr-xr-x 5 andrewt users 1024 Mar 2 23:17 ../
-rw-r--r-- 1 andrewt users 134932 Feb 24 23:39 5_tids_OEOS
-rw-r--r-- 1 andrewt users 14600 Feb 24 23:38 5_tids_OEOS.DB
-rwxr-xr-x 1 andrewt users 10761216 Feb 24 23:39 5_tids_OEOS.ML*

```

ch7:

```

total 52
drwxr-xr-x 9 andrewt users 1024 Mar 30 07:34 ./
drwxr-xr-x 5 andrewt users 1024 Mar 30 07:34 ../
drwxr-xr-x 2 andrewt users 1024 Mar 30 01:41 combined/

```

```

drwxr-xr-x 2 andrewt users 1024 Mar 29 20:06 deterministic_traversal/
-rw-r--r-- 1 andrewt users 11682 Mar 2 23:14 equiv_be.sml
-rw-r--r-- 1 andrewt users 6313 Mar 30 07:34 equiv_mark.sml
-rw-r--r-- 1 andrewt users 23220 Mar 3 00:11 gdn.sml
drwxr-xr-x 2 andrewt users 1024 Mar 29 20:41 imp_exp/
drwxr-xr-x 7 andrewt users 1024 Feb 24 23:43 multiple/
drwxr-xr-x 11 andrewt users 1024 Feb 22 17:59 single/
drwxr-xr-x 7 andrewt users 1024 Feb 24 23:32 single_concurrent/
drwxr-xr-x 8 andrewt users 1024 Feb 24 23:45 stopping/

```

ch7/combined:

total 26043

```

drwxr-xr-x 2 andrewt users 1024 Mar 30 01:41 ./
drwxr-xr-x 9 andrewt users 1024 Mar 30 07:34 ../
-rw-r--r-- 1 andrewt users 31874 Mar 30 01:39 593.node
-rw-r--r-- 1 andrewt users 64654 Mar 3 00:09 combined.rep
-rw-r--r-- 1 andrewt users 810552 Mar 30 01:41 combined_OG
-rw-r--r-- 1 andrewt users 152612 Mar 30 01:39 combined_OG.DB
-rwxr-xr-x 1 andrewt users 25493504 Mar 30 01:41 combined_OG.ML*

```

ch7/deterministic_traversal:

total 41341

```

drwxr-xr-x 2 andrewt users 1024 Mar 29 20:06 ./
drwxr-xr-x 9 andrewt users 1024 Mar 30 07:34 ../
-rw-r--r-- 1 andrewt users 11040 Feb 22 23:05 4346.node
-rw-r--r-- 1 andrewt users 11040 Feb 22 23:03 9636.node
-rw-r--r-- 1 andrewt users 13987 Feb 22 22:35 determin.rep
-rw-r--r-- 1 andrewt users 768996 Feb 22 23:06 determin_oeos
-rw-r--r-- 1 andrewt users 130332 Feb 22 23:05 determin_oeos.DB
-rwxr-xr-x 1 andrewt users 18506752 Feb 22 23:06 determin_oeos.ML*
-rw-r--r-- 1 andrewt users 768996 Feb 22 23:04 determin_og
-rw-r--r-- 1 andrewt users 130332 Feb 22 23:03 determin_og.DB
-rwxr-xr-x 1 andrewt users 21815296 Feb 22 23:04 determin_og.ML*

```

ch7/imp_exp:

total 30848

```

drwxr-xr-x 2 andrewt users 1024 Mar 29 20:41 ./
drwxr-xr-x 9 andrewt users 1024 Mar 30 07:34 ../
-rw-r--r-- 1 andrewt users 4871 Mar 29 20:39 366.node
-rw-r--r-- 1 andrewt users 5014 Mar 29 20:39 376.node
-rw-r--r-- 1 andrewt users 4838 Feb 22 15:59 602.node
-rw-r--r-- 1 andrewt users 4981 Feb 22 15:59 609.node
-rw-r--r-- 1 andrewt users 499 Mar 29 20:40 diff_oeos.out
-rw-r--r-- 1 andrewt users 491 Mar 29 20:41 diff_oga.out
-rw-r--r-- 1 andrewt users 6591 Feb 22 15:57 imp_exp.rep
-rw-r--r-- 1 andrewt users 763556 Mar 29 20:40 imp_exp_oeos
-rw-r--r-- 1 andrewt users 126036 Mar 29 20:39 imp_exp_oeos.DB
-rwxr-xr-x 1 andrewt users 14748672 Mar 29 20:39 imp_exp_oeos.ML*
-rw-r--r-- 1 andrewt users 756916 Feb 22 15:59 imp_exp_og
-rw-r--r-- 1 andrewt users 125468 Feb 22 15:59 imp_exp_og.DB
-rwxr-xr-x 1 andrewt users 14904320 Feb 22 15:59 imp_exp_og.ML*

```

ch7/multiple:

total 7

```

drwxr-xr-x 7 andrewt users 1024 Feb 24 23:43 ./
drwxr-xr-x 9 andrewt users 1024 Mar 30 07:34 ../
drwxr-xr-x 2 andrewt users 1024 Feb 24 23:42 AA/
drwxr-xr-x 2 andrewt users 1024 Feb 24 23:43 AB/
drwxr-xr-x 2 andrewt users 1024 Feb 24 23:43 AC/
drwxr-xr-x 2 andrewt users 1024 Feb 24 23:43 BB/
drwxr-xr-x 2 andrewt users 1024 Feb 24 23:43 CC/

```

ch7/multiple/AA:

total 15090

```

drwxr-xr-x 2 andrewt users 1024 Feb 24 23:42 ./
drwxr-xr-x 7 andrewt users 1024 Feb 24 23:43 ../
-rw-r--r-- 1 andrewt users 8622 Feb 24 23:42 437.node

```

```

-rw-r--r-- 1 andrewt users      8429 Feb 24 23:42 AA.rep
-rw-r--r-- 1 andrewt users    763452 Feb 24 23:42 AA_og
-rw-r--r-- 1 andrewt users    129604 Feb 24 23:42 AA_og.DB
-rwxr-xr-x 1 andrewt users   14474240 Feb 24 23:42 AA_og.ML*

```

ch7/multiple/AB:

total 15928

```

drwxr-xr-x 2 andrewt users      1024 Feb 24 23:43 ./
drwxr-xr-x 7 andrewt users      1024 Feb 24 23:43 ../
-rw-r--r-- 1 andrewt users     8419 Feb 24 23:43 1547.txt
-rw-r--r-- 1 andrewt users     9281 Feb 24 23:43 AB.rep
-rw-r--r-- 1 andrewt users    762872 Feb 24 23:43 AB_og
-rw-r--r-- 1 andrewt users    129428 Feb 24 23:43 AB_og.DB
-rwxr-xr-x 1 andrewt users   15329280 Feb 24 23:43 AB_og.ML*

```

ch7/multiple/AC:

total 16096

```

drwxr-xr-x 2 andrewt users      1024 Feb 24 23:43 ./
drwxr-xr-x 7 andrewt users      1024 Feb 24 23:43 ../
-rw-r--r-- 1 andrewt users     8682 Feb 24 23:43 1753.node
-rw-r--r-- 1 andrewt users    10867 Feb 24 23:43 AC.rep
-rw-r--r-- 1 andrewt users    763520 Feb 24 23:43 AC_og
-rw-r--r-- 1 andrewt users    129604 Feb 24 23:43 AC_og.DB
-rwxr-xr-x 1 andrewt users   15498240 Feb 24 23:43 AC_og.ML*

```

ch7/multiple/BB:

total 19116

```

drwxr-xr-x 2 andrewt users      1024 Feb 24 23:43 ./
drwxr-xr-x 7 andrewt users      1024 Feb 24 23:43 ../
-rw-r--r-- 1 andrewt users     8419 Feb 24 23:43 5429.node
-rw-r--r-- 1 andrewt users    10133 Feb 24 23:43 BB.rep
-rw-r--r-- 1 andrewt users    763020 Feb 24 23:43 BB_og
-rw-r--r-- 1 andrewt users    129428 Feb 24 23:43 BB_og.DB
-rwxr-xr-x 1 andrewt users   18580480 Feb 24 23:43 BB_og.ML*

```

ch7/multiple/CC:

total 20433

```

drwxr-xr-x 2 andrewt users      1024 Feb 24 23:43 ./
drwxr-xr-x 7 andrewt users      1024 Feb 24 23:43 ../
-rw-r--r-- 1 andrewt users     8681 Feb 24 23:43 6969.txt
-rw-r--r-- 1 andrewt users    12789 Feb 24 23:43 CC.rep
-rw-r--r-- 1 andrewt users    763516 Feb 24 23:43 CC_og
-rw-r--r-- 1 andrewt users    129604 Feb 24 23:43 CC_og.DB
-rwxr-xr-x 1 andrewt users   19919872 Feb 24 23:44 CC_og.ML*

```

ch7/single:

total 11

```

drwxr-xr-x 11 andrewt users      1024 Feb 22 17:59 ./
drwxr-xr-x 9 andrewt users      1024 Mar 30 07:34 ../
drwxr-xr-x 2 andrewt users      1024 Mar 29 09:38 scen1/
drwxr-xr-x 2 andrewt users      1024 Mar 29 13:22 scen2/
drwxr-xr-x 2 andrewt users      1024 Feb 21 23:39 scen3/
drwxr-xr-x 2 andrewt users      1024 Feb 22 12:07 scen4/
drwxr-xr-x 2 andrewt users      1024 Feb 22 12:06 scen5/
drwxr-xr-x 2 andrewt users      1024 Feb 22 12:07 scen6/
drwxr-xr-x 2 andrewt users      1024 Feb 22 12:05 scen7/
drwxr-xr-x 2 andrewt users      1024 Feb 22 12:07 scen8/
drwxr-xr-x 2 andrewt users      1024 Feb 22 12:07 scen9/

```

ch7/single/scen1:

total 15052

```

drwxr-xr-x 2 andrewt users      1024 Mar 29 09:38 ./
drwxr-xr-x 11 andrewt users      1024 Feb 22 17:59 ../
-rw-r--r-- 1 andrewt users     4807 Feb 24 21:48 75.node
-rw-r--r-- 1 andrewt users    755580 Feb 24 21:49 single_1
-rw-r--r-- 1 andrewt users    126800 Feb 24 21:48 single_1.DB
-rwxr-xr-x 1 andrewt users   14453760 Feb 24 21:49 single_1.ML*

```

```

-rw-r--r-- 1 andrewt users 5324 Feb 21 18:22 single_1.rep

ch7/single/scen2:
total 14559
drwxr-xr-x 2 andrewt users 1024 Mar 29 13:22 ./
drwxr-xr-x 11 andrewt users 1024 Feb 22 17:59 ../
-rw-r--r-- 1 andrewt users 4774 Feb 21 18:42 21.node
-rw-r--r-- 1 andrewt users 730792 Feb 21 18:42 single_2
-rw-r--r-- 1 andrewt users 125156 Feb 21 18:42 single_2.DB
-rwxr-xr-x 1 andrewt users 13977600 Feb 21 18:42 single_2.ML*
-rw-r--r-- 1 andrewt users 4472 Feb 21 18:35 single_2.rep

ch7/single/scen3:
total 14585
drwxr-xr-x 2 andrewt users 1024 Feb 21 23:39 ./
drwxr-xr-x 11 andrewt users 1024 Feb 22 17:59 ../
-rw-r--r-- 1 andrewt users 4779 Feb 21 18:52 75.node
-rw-r--r-- 1 andrewt users 730804 Feb 21 18:53 single_3
-rw-r--r-- 1 andrewt users 125160 Feb 21 18:53 single_3.DB
-rwxr-xr-x 1 andrewt users 14003200 Feb 21 18:53 single_3.ML*
-rw-r--r-- 1 andrewt users 5384 Feb 21 18:59 single_3.rep

ch7/single/scen4:
total 14598
drwxr-xr-x 2 andrewt users 1024 Feb 22 12:07 ./
drwxr-xr-x 11 andrewt users 1024 Feb 22 17:59 ../
-rw-r--r-- 1 andrewt users 4720 Feb 21 19:03 75.node
-rw-r--r-- 1 andrewt users 730740 Feb 21 19:03 single_4
-rw-r--r-- 1 andrewt users 125156 Feb 21 19:03 single_4.DB
-rwxr-xr-x 1 andrewt users 14017536 Feb 21 19:03 single_4.ML*
-rw-r--r-- 1 andrewt users 4877 Feb 21 18:57 single_4.rep

ch7/single/scen5:
total 14560
drwxr-xr-x 2 andrewt users 1024 Feb 22 12:06 ./
drwxr-xr-x 11 andrewt users 1024 Feb 22 17:59 ../
-rw-r--r-- 1 andrewt users 4720 Feb 21 19:28 21.node
-rw-r--r-- 1 andrewt users 730540 Feb 21 19:28 single_5
-rw-r--r-- 1 andrewt users 125156 Feb 21 19:28 single_5.DB
-rwxr-xr-x 1 andrewt users 13979648 Feb 21 19:28 single_5.ML*
-rw-r--r-- 1 andrewt users 4020 Feb 21 19:23 single_5.rep

ch7/single/scen6:
total 14560
drwxr-xr-x 2 andrewt users 1024 Feb 22 12:07 ./
drwxr-xr-x 11 andrewt users 1024 Feb 22 17:59 ../
-rw-r--r-- 1 andrewt users 4779 Feb 21 19:19 21.node
-rw-r--r-- 1 andrewt users 730604 Feb 21 19:19 single_6
-rw-r--r-- 1 andrewt users 125160 Feb 21 19:19 single_6.DB
-rwxr-xr-x 1 andrewt users 13978624 Feb 21 19:19 single_6.ML*
-rw-r--r-- 1 andrewt users 4527 Feb 21 19:14 single_6.rep

ch7/single/scen7:
total 14549
drwxr-xr-x 2 andrewt users 1024 Feb 22 12:05 ./
drwxr-xr-x 11 andrewt users 1024 Feb 22 17:59 ../
-rw-r--r-- 1 andrewt users 4778 Feb 21 19:39 21.node
-rw-r--r-- 1 andrewt users 730604 Feb 21 19:39 single_7
-rw-r--r-- 1 andrewt users 125160 Feb 21 19:39 single_7.DB
-rwxr-xr-x 1 andrewt users 13967360 Feb 21 19:39 single_7.ML*
-rw-r--r-- 1 andrewt users 4516 Feb 21 19:34 single_7.rep

ch7/single/scen8:
total 14560
drwxr-xr-x 2 andrewt users 1024 Feb 22 12:07 ./
drwxr-xr-x 11 andrewt users 1024 Feb 22 17:59 ../
-rw-r--r-- 1 andrewt users 4778 Feb 21 19:46 21.node

```

```

-rw-r--r-- 1 andrewt users 730604 Feb 21 19:46 single_8
-rw-r--r-- 1 andrewt users 125160 Feb 21 19:46 single_8.DB
-rwxr-xr-x 1 andrewt users 13978624 Feb 21 19:46 single_8.ML*
-rw-r--r-- 1 andrewt users 4516 Feb 21 19:41 single_8.rep

```

ch7/single/scen9:

```

total 14604
drwxr-xr-x 2 andrewt users 1024 Feb 22 12:07 ./
drwxr-xr-x 11 andrewt users 1024 Feb 22 17:59 ../
-rw-r--r-- 1 andrewt users 4847 Feb 21 21:15 85.node
-rw-r--r-- 1 andrewt users 731000 Feb 21 21:15 single_9
-rw-r--r-- 1 andrewt users 125220 Feb 21 21:15 single_9.DB
-rwxr-xr-x 1 andrewt users 14021632 Feb 21 21:15 single_9.ML*
-rw-r--r-- 1 andrewt users 6481 Feb 21 20:57 single_9.rep

```

ch7/single_concurrent:

```

total 7
drwxr-xr-x 7 andrewt users 1024 Feb 24 23:32 ./
drwxr-xr-x 9 andrewt users 1024 Mar 30 07:34 ../
drwxr-xr-x 2 andrewt users 1024 Mar 29 23:37 scen1/
drwxr-xr-x 2 andrewt users 1024 Mar 29 23:38 scen2/
drwxr-xr-x 2 andrewt users 1024 Mar 29 17:17 scen3/
drwxr-xr-x 2 andrewt users 1024 Mar 29 17:48 scen4/
drwxr-xr-x 2 andrewt users 1024 Mar 29 23:39 scen5/

```

ch7/single_concurrent/scen1:

```

total 32768
drwxr-xr-x 2 andrewt users 1024 Mar 29 23:37 ./
drwxr-xr-x 7 andrewt users 1024 Feb 24 23:32 ../
-rw-r--r-- 1 andrewt users 5058 Mar 29 14:27 399.node
-rw-r--r-- 1 andrewt users 5058 Feb 22 23:23 680.node
-rw-r--r-- 1 andrewt users 9043 Feb 28 15:15 AA.rep
-rw-r--r-- 1 andrewt users 756196 Mar 29 14:27 AA_oeos
-rw-r--r-- 1 andrewt users 127048 Mar 29 14:26 AA_oeos.DB
-rwxr-xr-x 1 andrewt users 16800768 Mar 29 14:27 AA_oeos.ML*
-rw-r--r-- 1 andrewt users 756152 Feb 22 23:24 AA_og
-rw-r--r-- 1 andrewt users 127048 Feb 22 23:23 AA_og.DB
-rwxr-xr-x 1 andrewt users 14825472 Feb 22 23:24 AA_og.ML*

```

ch7/single_concurrent/scen2:

```

total 32899
drwxr-xr-x 2 andrewt users 1024 Mar 29 23:38 ./
drwxr-xr-x 7 andrewt users 1024 Feb 24 23:32 ../
-rw-r--r-- 1 andrewt users 4935 Mar 29 16:13 1547.node
-rw-r--r-- 1 andrewt users 4991 Feb 22 23:38 2630.node
-rw-r--r-- 1 andrewt users 9895 Feb 22 23:31 AB.rep
-rw-r--r-- 1 andrewt users 756664 Mar 29 16:14 AB_oeos
-rw-r--r-- 1 andrewt users 125916 Mar 29 16:14 AB_oeos.DB
-rwxr-xr-x 1 andrewt users 15587328 Mar 29 16:14 AB_oeos.ML*
-rw-r--r-- 1 andrewt users 755960 Feb 22 23:39 AB_og
-rw-r--r-- 1 andrewt users 126988 Feb 22 23:39 AB_og.DB
-rwxr-xr-x 1 andrewt users 16174080 Feb 22 23:39 AB_og.ML*

```

ch7/single_concurrent/scen3:

```

total 40016
drwxr-xr-x 2 andrewt users 1024 Mar 29 17:17 ./
drwxr-xr-x 7 andrewt users 1024 Feb 24 23:32 ../
-rw-r--r-- 1 andrewt users 4992 Feb 23 00:06 10124.node
-rw-r--r-- 1 andrewt users 4935 Mar 29 17:11 4274.node
-rw-r--r-- 1 andrewt users 10747 Feb 22 23:48 BB.rep
-rw-r--r-- 1 andrewt users 755948 Mar 29 17:17 BB_oeos
-rw-r--r-- 1 andrewt users 125916 Mar 29 17:17 BB_oeos.DB
-rwxr-xr-x 1 andrewt users 17510400 Mar 29 17:17 BB_oeos.ML*
-rw-r--r-- 1 andrewt users 755964 Feb 23 00:06 BB_og
-rw-r--r-- 1 andrewt users 126988 Feb 23 00:06 BB_og.DB
-rwxr-xr-x 1 andrewt users 21509120 Feb 23 00:06 BB_og.ML*

```

```

ch7/single_concurrent/scen4:
total 33166
drwxr-xr-x  2 andrewt  users          1024 Mar 29 17:48 ./
drwxr-xr-x  7 andrewt  users          1024 Feb 24 23:32 ../
-rw-r--r--  1 andrewt  users          5008 Mar 29 17:44 1753.node
-rw-r--r--  1 andrewt  users          5064 Feb 23 00:30 2992.node
-rw-r--r--  1 andrewt  users         11052 Feb 23 00:16 AC.rep
-rw-r--r--  1 andrewt  users        756152 Mar 29 17:45 AC_oeos
-rw-r--r--  1 andrewt  users       125980 Mar 29 17:45 AC_oeos.DB
-rwxr-xr-x  1 andrewt  users     15624192 Mar 29 17:45 AC_oeos.ML*
-rw-r--r--  1 andrewt  users       756168 Feb 23 00:30 AC_og
-rw-r--r--  1 andrewt  users       127052 Feb 23 00:30 AC_og.DB
-rwxr-xr-x  1 andrewt  users     16407552 Feb 23 00:30 AC_og.ML*

ch7/single_concurrent/scen5:
total 42450
drwxr-xr-x  2 andrewt  users          1024 Mar 29 23:39 ./
drwxr-xr-x  7 andrewt  users          1024 Feb 24 23:32 ../
-rw-r--r--  1 andrewt  users          5070 Feb 23 00:56 13104.node
-rw-r--r--  1 andrewt  users          5069 Mar 29 18:50 4096.txt
-rw-r--r--  1 andrewt  users         13066 Feb 23 00:38 CC.rep
-rw-r--r--  1 andrewt  users        756228 Mar 29 18:50 CC_oeos
-rw-r--r--  1 andrewt  users       125988 Mar 29 18:50 CC_oeos.DB
-rwxr-xr-x  1 andrewt  users     17430528 Mar 29 18:50 CC_oeos.ML*
-rw-r--r--  1 andrewt  users       756184 Feb 23 00:57 CC_og
-rw-r--r--  1 andrewt  users       127056 Feb 23 00:56 CC_og.DB
-rwxr-xr-x  1 andrewt  users     24068096 Feb 23 00:57 CC_og.ML*

ch7/stopping:
total 8
drwxr-xr-x  8 andrewt  users          1024 Feb 24 23:45 ./
drwxr-xr-x  9 andrewt  users          1024 Mar 30 07:34 ../
drwxr-xr-x  2 andrewt  users          1024 Feb 24 23:45 stop1/
drwxr-xr-x  2 andrewt  users          1024 Feb 24 23:46 stop2/
drwxr-xr-x  2 andrewt  users          1024 Feb 24 23:46 stop3/
drwxr-xr-x  2 andrewt  users          1024 Feb 24 23:46 stop4/
drwxr-xr-x  2 andrewt  users          1024 Feb 24 23:48 stop5/
drwxr-xr-x  2 andrewt  users          1024 Feb 24 23:48 stop6/

ch7/stopping/stop1:
total 17263
drwxr-xr-x  2 andrewt  users          1024 Feb 24 23:45 ./
drwxr-xr-x  8 andrewt  users          1024 Feb 24 23:45 ../
-rw-r--r--  1 andrewt  users       12257 Feb 24 23:45 2628.node
-rw-r--r--  1 andrewt  users       17622 Feb 24 23:45 stop1.rep
-rw-r--r--  1 andrewt  users       770412 Feb 24 23:45 stop1_og
-rw-r--r--  1 andrewt  users       132924 Feb 24 23:45 stop1_og.DB
-rwxr-xr-x  1 andrewt  users     16667648 Feb 24 23:45 stop1_og.ML*

ch7/stopping/stop2:
total 15754
drwxr-xr-x  2 andrewt  users          1024 Feb 24 23:46 ./
drwxr-xr-x  8 andrewt  users          1024 Feb 24 23:45 ../
-rw-r--r--  1 andrewt  users       12022 Feb 24 23:45 758.node
-rw-r--r--  1 andrewt  users       11384 Feb 24 23:45 stop2.rep
-rw-r--r--  1 andrewt  users       770044 Feb 24 23:46 stop2_og
-rw-r--r--  1 andrewt  users       132864 Feb 24 23:46 stop2_og.DB
-rwxr-xr-x  1 andrewt  users     15136768 Feb 24 23:46 stop2_og.ML*

ch7/stopping/stop3:
total 15429
drwxr-xr-x  2 andrewt  users          1024 Feb 24 23:46 ./
drwxr-xr-x  8 andrewt  users          1024 Feb 24 23:45 ../
-rw-r--r--  1 andrewt  users       12108 Feb 24 23:46 296.node
-rw-r--r--  1 andrewt  users       10745 Feb 24 23:46 stop3.rep
-rw-r--r--  1 andrewt  users       770256 Feb 24 23:46 stop3_og
-rw-r--r--  1 andrewt  users       132920 Feb 24 23:46 stop3_og.DB

```

```

-rwxr-xr-x 1 andrewt users 14804992 Feb 24 23:46 stop3_og.ML*

ch7/stopping/stop4:
total 15867
drwxr-xr-x 2 andrewt users 1024 Feb 24 23:46 ./
drwxr-xr-x 8 andrewt users 1024 Feb 24 23:45 ../
-rw-r--r-- 1 andrewt users 12113 Feb 24 23:46 844.node
-rw-r--r-- 1 andrewt users 12689 Feb 24 23:46 stop4.rep
-rw-r--r-- 1 andrewt users 770264 Feb 24 23:46 stop4_og
-rw-r--r-- 1 andrewt users 132920 Feb 24 23:46 stop4_og.DB
-rwxr-xr-x 1 andrewt users 15248384 Feb 24 23:46 stop4_og.ML*

ch7/stopping/stop5:
total 15747
drwxr-xr-x 2 andrewt users 1024 Feb 24 23:48 ./
drwxr-xr-x 8 andrewt users 1024 Feb 24 23:45 ../
-rw-r--r-- 1 andrewt users 12090 Feb 24 23:48 758.node
-rw-r--r-- 1 andrewt users 11724 Feb 24 23:48 stop5.rep
-rw-r--r-- 1 andrewt users 770240 Feb 24 23:48 stop5_og
-rw-r--r-- 1 andrewt users 132924 Feb 24 23:48 stop5_og.DB
-rwxr-xr-x 1 andrewt users 15128576 Feb 24 23:48 stop5_og.ML*

ch7/stopping/stop6:
total 29277
drwxr-xr-x 2 andrewt users 1024 Feb 24 23:48 ./
drwxr-xr-x 8 andrewt users 1024 Feb 24 23:45 ../
-rw-r--r-- 1 andrewt users 12258 Feb 24 23:48 16005.node
-rw-r--r-- 1 andrewt users 770416 Feb 24 23:48 stop6_og
-rw-r--r-- 1 andrewt users 132924 Feb 24 23:48 stop6_og.DB
-rwxr-xr-x 1 andrewt users 28941312 Feb 24 23:48 stop6_og.ML*

java:
total 9012
drwxr-xr-x 2 andrewt users 1024 Mar 22 20:42 ./
drwxr-xr-x 5 andrewt users 1024 Mar 30 07:34 ../
-rwxr-xr-x 1 andrewt users 9089308 Feb 24 23:50 jdk1_1_1_linux.tar.gz*
-rw-r--r-- 1 andrewt users 2194 Mar 22 20:41 tr_trace.tar.gz
-rw-r--r-- 1 andrewt users 94286 Mar 22 20:42 trader_100.tar.gz

```