

Bridging the Specification Protocol Gap in Argumentation

Ashwag Omar Magraby



Doctor of Philosophy

Centre for Intelligent Systems and their Applications

School of Informatics

University of Edinburgh

2013

Abstract

As a multi-agent system (MAS) has become more mature and systems in general have become more distributed, it is necessary for those who want to build large scale systems to consider, in some computational depth, how agents can communicate in large scale, complex and distributed systems. Currently, some MAS systems have been developed to use an abstract specification language for argumentation. This as a basis for agent communication; to provide effective decision support for agents and yield better agreements. However, as we build complete MAS that involve argumentation, there is a need to produce concrete implementations in which these abstract specifications are realised via protocols coordinating agent behaviour. This creates a gap between standard argument specification and deployment of protocols.

This thesis attempts to close this gap by using a combination of automated synthesis and verification methods. More precisely, this thesis proposes a means of moving rapidly from argument specification to protocol implementation using an extension of the Argument Interchange Format (AIF is a generic specification language for argument structure) called a Dialogue Interaction Diagram (DID) as the specification language and the Lightweight Coordination Calculus (LCC is an executable specification language used for coordinating agents in open systems) as an implementation language.

The main contribution of this research is to provide approaches for enabling developers of argumentation systems to use specification languages (in our case AIF/DID) to generate agent protocol systems that are capable of direct implementation on open infrastructures (in our case LCC).

Acknowledgements

I wish to thank, first and foremost, my God (Allah) for given me the power to believe in myself and giving me the strength to complete this work.

Thank you to everyone who has helped me in completing this work. Above all, I am eternally grateful to my beloved family. My parents Omar Maghraby and Khadiyja Alsolimani for their unconditional love, their faith in me, endurance and encouragement. I express my deep gratitude for the support they have provided to me over the years without which this work would not have been completed. Also, a special gratitude and love goes to my sister and brother for their concern, their advice and their unfailing support. Thank you for believing in me.

I would like to express my special appreciation and thanks to my primary supervisor, Prof. David Robertson, for his guidance, motivation, support and encouragement throughout the course of my research. His positive attitude and confidence in my research inspired me and gave me confidence.

I would also like to thank my second supervisor, Dr. Michael Rovatsos, for his sound advice and encouragement. Additionally, I would like to express my deepest gratitude to Adela Grando for her suggestions, understanding, encouragement and personal attention.

I cannot forget to express my deep and sincere thanks to Umm Al-Qura University for their financial support.

Last but not least, special thanks should be given to my friends (too many to list here but you know who you are!) for providing support and friendship that I needed during my research.

(Ashwag Omar Maghraby)

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Ashwag Omar Maghraby)

Table of Contents

CHAPTER 1: INTRODUCTION	1
1.1 THE CHALLENGE	2
1.2 THE PROPOSED APPROACH	3
1.3 APPROACH	4
1.4 CLAIMS OF NOVELTY	4
1.5 THESIS STRUCTURE.....	5
CHAPTER 2: BACKGROUND AND LITERATURE REVIEW	7
2.1 AGENT PROTOCOL DEVELOPMENT LANGUAGE	7
2.1.1 LCC SYNTAX.....	8
2.1.2 LCC EXAMPLE	10
2.2 DESIGN PATTERN.....	13
2.2.1 <i>SOFTWARE ENGINEERING DESIGN PATTERN.....</i>	14
2.2.2 <i>AGENT PROTOCOL DESIGN PATTERN.....</i>	14
2.2.3 <i>LOGIC PROGRAMMING PATTERNS (LOGIC PROGRAMMING TECHNIQUES)</i>	17
2.2.3.1 <i>Prolog Programming Techniques.....</i>	17
2.2.3.2 <i>Techniques editing.....</i>	19
2.2.3.3 <i>A Structural Synthesis System for LCC Protocols.....</i>	20
2.2.3.4 <i>Comparing LCC-Argument Patterns with Logic Programming Techniques</i>	22
2.3 VERIFICATION METHOD BASED ON SML AND COLOURED PETRI NET	23
2.3.1 <i>COLOURED PETRI NETS (CPNs).....</i>	24
2.3.1.1 <i>CPNs Model Elements.....</i>	24
2.3.1.2 <i>CPNs Hierarchical Structure</i>	28
2.3.1.3 <i>CPN Tool Components</i>	30
2.3.2 <i>COMPARING OUR APPROACH WITH VERIFICATION APPROACHES BASED ON SML AND CPN MODEL.....</i>	36
2.3.2.1 <i>A Transformational Approach to CPN Model</i>	366
2.3.2.2 <i>A Verification Method based on SML</i>	37
2.3.2.3 <i>LCC Verification Approaches based on Model Checking.....</i>	38
2.4 SUMMARY.....	39
CHAPTER 3: ARGUMENTATION, DIALOGUE GAMES AND MULTI-AGENT SYSTEMS.....	40

3.1 ARGUMENT AND ARGUMENTATION	40
3.2 DIALOGUE GAMES (ARGUMENTATION-BASED DIALOGUES).....	41
3.3 ARGUMENTATION FOR AGENT COMMUNICATION	45
3.4 DIALOGUES GAMES TERMINOLOGY	46
3.5 TYPES OF DIALOGUES	49
3.6 EMBEDDED DIALOGUES	53
3.6.1 FIRST TYPE: SHIFT FROM ONE TYPE TO ANOTHER TYPE.....	544
3.6.2 SECOND TYPE: INTERNAL EMBEDDED.....	54
3.7 ARGUMENTATION SHARING PROBLEM AND ARGUMENT INTERCHANGE FORMAT	54
3.7.1 AIF DEFINITION.....	555
3.7.2 AIF ELEMENTS	55
3.7.3 AIF EXAMPLE.....	56
3.7.4 AIF IMPLEMENTATION PROBLEM.....	57
3.7.5 AIF EXTENSION	58
3.8 SUMMARY.....	64
CHAPTER 4: ARGUMENT SPECIFICATION LANGUAGE	65
4.1 AGENT PROTOCOL CONCEPTS FOR ARGUMENTATION BETWEEN TWO AGENTS	66
4.2 DIALOGUE INTERACTION DIAGRAM (AN EXTENSION OF AIF)	67
4.2.1 DID ELEMENTS	68
4.2.2 HOW TO DRAW A DID DIAGRAM	70
4.2.3 EXAMPLE (PERSUASION DIALOGUE)	73
4.2.4 DID FOR TWO AGENTS FORMAL DEFINITION	78
4.3 DIALOGUE INTERACTION DIAGRAM FOR EMBEDDING DIALOGUE.....	88
4.3.1 DID FOR EMBEDDING DIALOGUE	88
4.3.2 DFSL FOR EMBEDDING DIALOGUE	88
4.3.3 EXAMPLE	89
4.4 DIALOGUE INTERACTION DIAGRAM FOR ARGUMENTATION BETWEEN N-AGENT	102
4.4.1 NEED FOR DIALOGUE GAMES AMONG N-AGENT.....	1022
4.4.2 ISSUES OF DIALOGUE GAMES AMONG N-AGENT	102
4.4.3 METHOD FOR DIALOGUE GAMES AMONG N-AGENT.....	105
4.4.4 DID FOR N-AGENT.....	109
4.4.5 PROBLEMS AND SOLUTIONS OF DID FOR N-AGENT.....	110

4.5 SUMMARY.....	112
CHAPTER 5 : SYNTHESIS OF CONCRETE PROTOCOLS.....	114
5.1 LCC-ARGUMENT PATTERNS	115
5.2 AGENT PROTOCOL AUTOMATED SYNTHESIS TOOL.....	143
5.2.1 <i>AUTOMATED SYNTHESIS STEPS FOR GENERATING AGENT PROTOCOL BETWEEN TWO AGENTS</i>	<i>144</i>
5.2.2 <i>AUTOMATED SYNTHESIS STEPS FOR GENERATING AGENT PROTOCOL FOR N-AGENT.....</i>	<i>144</i>
5.3 SUMMARY.....	148
CHAPTER 6: VERIFICATION METHOD BASED ON COLOURED PETRI NETS AND SML	150
6.1 STEP ONE: AUTOMATED TRANSFORMATION FROM LCC TO CPNXML.....	151
6.1.1 <i>DECLARATION OF COLOUR SETS AND FUNCTIONS</i>	<i>152</i>
6.1.2 <i>GENERATION OF A CPN SUBPAGE.....</i>	<i>152</i>
6.1.3 <i>GENERATION OF A CPN SUPERPAGE.....</i>	<i>169</i>
6.2 STEP TWO: CONSTRUCTION OF STATE SPACE	172
6.3 STEP THREE: AUTOMATED CREATION OF DID PROPERTIES FILES.....	174
6.4 STEP FOUR: APPLYING VERIFICATION MODEL	175
6.5 SUMMARY.....	187
CHPATER 7: DESIGN AND IMPLEMENTAION	189
7.1 ARCHITECTURE	189
7.1.1 <i>PART ONE: SYNTHESIS OF CONCRETE PROTOCOLS ARCHITECTURE</i>	<i>189</i>
7.1.2 <i>PART TWO: VERIFICATION MODEL ARCHITECTURE</i>	<i>189</i>
7.2 AN EXAMPLE SCENARIO	191
7.3 SUMMARY.....	201
CHPATER 8: EVALUATION AND DISCUSSION.....	202
8.1 SYNTHESIS OF CONCRETE PROTOCOLS	202
8.1.1 <i>RELATION BETWEEN DID AND AIF</i>	<i>202</i>
8.1.2 <i>THE DIFFERENCE BETWEEN DID AND AIF EXTENSION</i>	<i>208</i>
8.1.3 <i>DID LIMITATION.....</i>	<i>215</i>
8.1.4 <i>LCC-ARGUMENT PATTERNS LIMITATIONS.....</i>	<i>217</i>
8.2 VERIFICATION METHOD BASED ON COLOURED PETRI NET AND SML	220

8.2.1 AUTOMATICALLY TRANSFORMING THE LCC SPECIFICATION INTO AN EQUIVALENT CPNXML FILE LIMITATION	221
8.2.2 CONSTRUCTION OF THE STATE SPACE LIMITATION	221
8.2.3 AUTOMATICALLY VERIFYING METHOD LIMITATION	222
8.3 GENERATE LCC PROTOCOL TOOL	222
8.3.1 TASK ONE: SYNTHESIS OF CONCRETE PROTOCOLS	222
8.3.2 TASK TWO: MODEL VERIFICATION	223
8.4 SUMMARY	225
CHAPTER 9: CONCLUSION AND FUTURE WORK	20226
9.1 SUMMARY OF CONTRIBUTIONS	22026
9.2 IMPROVEMENTS AND FUTURE WORK	2208
9.2.1 DID FUTURE WORK	228
9.2.2 AUTOMATED SYNTHESIS METHOD FUTURE WORK	229
9.2.3 SEMI-AUTOMATED VERIFICATION METHOD FUTURE WORK	230
9.2.4 OTHER FUTURE WORK	231
APPENDIX A : NEGOTIATION DIALOGUE	ERROR! BOOKMARK NOT DEFINED.
A.1 NEGOTIATION DIALOGUE EXAMPLE	ERROR! BOOKMARK NOT DEFINED.
A.2 DFSL OF THE NEGOTIATION DIALOGUE	ERROR! BOOKMARK NOT DEFINED.
A.3 DID OF THE NEGOTIATION DIALOGUE	ERROR! BOOKMARK NOT DEFINED.
A.4 THE PICTURE HANGING EXAMPLE	ERROR! BOOKMARK NOT DEFINED.
A.5 LCC SYNTHESIS PROTOCOL OF THE NEGOTIATION DIALOGUE	ERROR! BOOKMARK NOT DEFINED.
A.6 VERIFICATION MODEL OF THE LCC SYNTHESIS PROTOCOL OF THE NEGOTIATION DIALOGUE	ERROR! BOOKMARK NOT DEFINED.
APPENDIX B: N-AGENT DIALOGUE	266
B.1 DID FOR N-AGENT FORMAL DEFINITION	266
B.2 DID FOR N-AGENT EXAMPLE	274
B.3 GENERAL N-AGENT PATTERNS	288
B.3.1 GENERAL LCC-ARGUMENT N-AGENT PATTERNS	288
B.3.2 AUTOMATED SYNTHESIS STEPS FOR GENERATING AGENT PROTOCOL FOR GENERAL N-AGENT AUTOMATICALLY	306
B.3.3 AN EXAMPLE OF AN LCC PROTOCOL BEGIN GENERATED FOR GENERAL N-AGENT DIALOGUE	309

APPENDIX C: PERSUASION DIALOGUE.....	315
C.1 AN EXAMPLE OF AN LCC PROTOCOL BEGIN GENERATED FOR TWO AGENTS	315
C.2 AN EXAMPLE OF AN LCC PROTOCOL BEGIN GENERATED FOR N-AGENT	315
C.3 VERIFICATION MODEL OF THE PERSUASION DIALOGUE	330
APPENDIX D: CPN FUNCTIONS	354
APPENDIX E: GENERATELCCPROTOCOL TOOL GRAPHICAL USER INTERFACE	354
E.1 GRAPHICAL USER INTERFACE FOR SYNTHESIS OF CONCRETE PROTOCOLS (PART ONE)	357
<i>E.1.1 DIALOGUE INTERACTION DIAGRAM</i>	<i>357</i>
<i>E.1.2 SYNTHESISING CONCRETE LCC PROTOCOLS FROM DID SPECIFICATIONS.....</i>	<i>370</i>
E.2 A GRAPHICAL USER INTERFACE FOR PART TWO: VERIFICATION MODEL SCREENS.....	371
APPENDIX F: PUBLISHED PAPERS	354
BIBLIOGRAPHY	376

List of Figures

1.1: SYSTEM ARCHITECTURE	5
2.1: THE SLAVE CLASS.....	16
2.2: CPNS MODEL ELEMENTS EXAMPLE.....	25
2.3: A HIERACHICAL CPN	29
2.4: CPN TOOL.....	32
2.5 (A): CPNXMK FILE STRUCTURE EXAMPLE	33
2.5 (B): CPNXMK FILE STRUCTURE EXAMPLE	34
2.6: STATE SPACE GRAPH	35
3.1: PERSUASION DIALOGUE EXAMPLE (CAR SAFETY CASE)	51
3.2: DETERMINING THE TYPE OF DIALOGUE	53
3.3: SPECIFICATION IN AIF OF THE ARGUMENTS EXCHANGED BY AGENTS DISCUSSING THE FLYING ABILITIES OF THE "P" BIRD	56
3.4: A DIALOGUE GRAPH REPRESENTED IN THE AIF.....	59
3.5: ILLUSTRATING THE LINK BETWEEN ARGUMENT (AIF NODES) AND DIALOGUE GAMES (AIF ⁺ NODES)	62
4.1: MISSING CONCEPTS BETWEEN AIF AND AGENT PROTOCOL	65
4.2: LOCUTION ICON	69
4.3 DID STRUCTURE OF A PERSUASION DIALOGUE.....	75
4.4: THE COMPLEX CAR SAFETY EXAMPLE	77
4.5: THE PERSUASION DIALOGUE LEGAL MOVES.....	87
4.8: THE INQUIRY DIALOGUE LEGAL MOVES	96
4.9: DID STRUCTURE OF AN INQUIRY DIALOGUE.....	98
4.10(A): INQUIRY DIALOGUE LOCUTIONS PRE-CONDITIONS AND POST-CONDITIONS	99
4.10 (B): INQUIRY DIALOGUE LOCUTIONS PRE-CONDITIONS AND POST-CONDITIONS.....	100
4.10 (C): INQUIRY DIALOGUE LOCUTIONS PRE-CONDITIONS AND POST-CONDITIONS.....	101
4.11: EMBEDDED INQUIRY DIALOGUE EXAMPLE	103
4.12: DIALOGUE AMONG N-AGENT	106

4.13: EXAMPLE TWO OF DIALOGUE AMONG N-AGENT	107
4.14: EXAMPLE THREE OF DIALOGUE AMONG N-AGENT	108
4.15: LOCUTION ICON FOR N-AGENT	111
4.16: BLACK BOX OF DID FOR N-AGENT.....	112
5.1: BROADCASTING PATTERN SOLUTION (STEP ONE)	125
5.2: BROADCASTING PATTERN SOLUTION (STEP TWO).....	127
5.3: BROADCASTING PATTERN SOLUTION (STEP THREE: DIVIDE).....	128
5.4: BROADCASTING PATTERN SOLUTION (STEP THREE: TERMINATION)	129
5.5 : STRUCTURE (PROPOSALSENDER _{PROPOSAL} AND PROPSALRECEIVER _{RECEIVER} ROLES)	131
5.6 : STRUCTURE (REPLYTOPROPSALSENDER AND REPLYTOPRPOSALRECEIVER _{PROPOSAL} ROLES) ...	132
5.7 : STRUCTURE (<i>RESULTSENDER</i> _{PROPOSAL} , <i>SENDREACHAGREEMENT</i> _{PROPOSAL} , <i>DIVIDEGROUP</i> _{PROPOSAL} AND <i>RESULTRECEIVER</i> ROLES)	133
5.8: SOLUTION OF RECURS-TO-N-DIALOGUE PATTERN.....	141
5.9: AGENT PROTOCOL AUTOMATED SYNTHESIS TOOL.....	144
5.10: TWO AGENTS PROTOCOL AUTOMATED SYNTHESIS ALGORITHM	145
5.11: N-AGENTS' PROTOCOL AUTOMATED SYNTHESIS ALGORITHM	147
6.1: VERIFICATION PROCESS.....	151
6.2: STATE SPACE TOOL PALETTE.....	173
6.3: PROPERTY 1 AS AN STANDARD ML FUNCTION	177
6.4: PROPERTY 2 AS AN STANDARD ML FUNCTION	179
6.5: PROPERTY 3 AS AN STANDARD ML FUNCTION	182
6.6: PROPERTY 4 AS AN STANDARD ML FUNCTION	185
6.7: PROPERTY 5 AS AN STANDARD ML FUNCTION	186
7.1: GENERATELCCPROTOCOL TOOL.....	188
7.2: OVERALL ARCHITECTURE.....	190
7.3: AN EXAMPLE SCENARIO OF <i>GENERATELCCPROTOCOL</i> TOOL	192
7.4: CREATE NEW DIALOGUE INTERACTION DIAGRAM EXAMPLE (CLAIM LOCUTION ICON)	193

7.5: CREATE NEW DIALOGUE INTERACTION DIAGRAM EXAMPLE (ADD LOCUTION FORMAL DEFINITION TO DID)	193
7.6: OPEN DID FILE DIALOGUE BOX	194
7.7: THE DID TEXTUAL REPRESENTATION OF THE PERSUASION DIALOGUE	195
7.8: SYNTHESISES OF LCC PROTOCOL OF THE PERSUASION DIALOGUE.....	196
7.9: SPECIFYING AGENTS KNOWLEDGE BASE SCREENS.....	197
7.10: TRANSFORMING LCC PROTOCOL INTO AN EQUIVALENT CPN MODEL SCREENS.....	199
7.11: INSTRUCTION SCREEN	200
7.12: VERIFICATION MODEL RESULT SCREEN.....	201
8.1:THE RELATIONSHIP BETWEEN AIF AND DID LOCUTIONS ICON.....	204
8.2 (A) : ILLUSTRATING THE LINK BETWEEN ARGUMENT (AIF NODES) AND DID LOCUTIONS.....	205
8.2 (B): ILLUSTRATING THE LINK BETWEEN ARGUMENT (AIF NODES) AND DID LOCUTIONS.....	206
8.3: LOCUTION CONCEPTS.....	209
8.4 (A): DIALOGUE GAMES CONCEPTS.....	211
8.4 (B): DIALOGUE GAMES CONCEPTS	212
8.5: MODGIL AND MCGINNIS EXAMPLE OF DIALOGUE GAMES CONCEPTS.....	212
8.6: AIF+ DESCRIPTION OF PERSUASION DIALOGUE GAMES.....	213
8.7: DID PROTOCOL IMPLEMENTATION CONCEPTS	214
8.8: PARTIAL DID DIAGRAM	218
8.9: <i>POSSIBLE SEQUENCE OF REPLY MOVES</i>	218
A.1: THE NEGOTIATION DIALOGUE LEGAL MOVES	238
A.2: DID STRUCTURE OF A NEGOTIATION DIALOGUE	240
A.3(A): NEGOTIATION DIALOGUE LOCUTIONS PRECONDITIONS AND POSTCONDITONS	241
A.3 (B): NEGOTIATION DIALOGUE LOCUTIONS PRECONDITIONS AND POSTCONDITONS.....	242
A.3 (C): NEGOTIATION DIALOGUE LOCUTIONS PRECONDITIONS AND POSTCONDITONS.....	243
A.3 (D): NEGOTIATION DIALOGUE LOCUTIONS PRECONDITIONS AND POSTCONDITONS	244
A.4: THE PICTURE HANGING EXAMPLE.....	245
A.5(A): GENERATED LCC PROTOCOL FOR NEGOTIATION DIALOUGE	247

A.5(B): GENERATED LCC PROTOCOL FOR NEGOTIATION DIALOUGE	248
A.6: THE REQUESTSENDERA CPN SUBPAGE	251
A.7:THE REQUESTRECEIVERB CPN SUBPAGE	251
A.8: THE REPLYTOREQUESTSENDERB CPN SUBPAGE	252
A.9: THE REPLYTOREQUESTSENDERB CPN SUBPAGE	253
A.10: THE REPLYTOCHALLENGESENDERA CPN SUBPAGE	248
A.11: <i>THE REPLYTOCHALLENGERECEIVERB CPN SUBPAGE</i>	253
A.12: THE REPLYTOJUSTIFYSENDERB CPN SUBPAGE.....	254
A.13: THE REPLYTOJUSTIFYRECEIVERA CPN SUBPAGE	254
A.14: THE REPLYTOPROMISESENDERA CPN.....	255
A.15:THE REPLYTOPROMISERECEIVERB CPN	255
A.16: THE PROTOCOL CPN SUPERPAGE	256
A.17: THE STATE SPACE GRAPH	257
A.18: POSSIBLE LOCUTIONS FILE	257
A.19: REPLY LOCUTIONS FILE.....	257
A.20: STARTING LOCUTIONS FILE	257
A.21: INTERMEDIATE LOCUTIONS FILE	258
A.22: TER,OMATOPM LOCUTIONS FILE	258
A.23: TERMINATION LOCUTIONS EFFECT CS AND EFFECTIVE CS FILES	258
A.24: PLAYER TYPES FILE	258
A.25: PLAYER IDS FILE.....	258
A.26: TERMINATION ROLE NAMES FILE.....	259
A.27: THE VERIFICATION RESULT OF THE FIVE BASIC PROPERTIES.....	259
2.8: SUCCESSFUL AND UNSUCCESSFUL DIALOGUE EXAMPLES.....	261
2.9: C-SUCCESSFUL DIALOGUE EXAMPLE.....	262
A.30: PROPERTY 6 (SUCCESSFUL DIALOGUE) AS AN STANDARD ML FUNCTION.....	262
A.31: PROPERTY 7 (C-SUCCESSFUL DIALOGUE) AS AN STANDARD ML FUNCTION	264

A.32: PROPERTY 6 (SUCCESSFUL DIALOGUE) VERIFICATION RESULT	265
A.33: PROPERTY 7 (C-SUCCESSFUL DIALOGUE) VERIFICATION RESULT	265
B.1: PERSUASION DIALOGUE BETWEEN N-AGENT	275
B.2:THE PERSUASION DIALOUGE BETWEEN N-AGENT LEGAL MOVES.....	280
B.3: DIALOGUE INTERACTION DIAGRAM FOR N-AGENT (DIDN).....	282
B.4(A): DIDN LOCUTIONS PRE-CONDITIONS AND POST-CONDITIONS.....	283
B.4(B): DIDN LOCUTIONS PRE-CONDITIONS AND POST-CONDITIONS.....	284
B.4(C): DIDN LOCUTIONS PRE-CONDITIONS AND POST-CONDITONS	285
B.5: THE COMPLEX CAR SAFETY EXAMPLE AMONG N-AGENT	287
B.6: RECURSIVE STARTING(SENDING) PATTERN SOLUTION	290
B.7: RECURSIVE TERMINATION-RECUR PATTERN (TERMINATION) SOLUTION	297
B.8: RECURSIVE TERMINATION-DIVIDED PATTERN STRUCTURE	303
B.9: N-AGENT PROTOCOL AUTOMATED SYNTHESIS ALGORITHM	307
B.10(A): GENERATED LCC PROTOCOL FOR N-AGENT DIALOGUE	310
B.10(B): GENERATED LCC PROTOCOL FOR N-AGENT DIALOGUE	311
B.10(C): GENERATED LCC PROTOCOL FOR N-AGENT DIALOGUE	312
B.10(D): GENERATED LCC PROTOCOL FOR N-AGENT DIALOGUE	313
C.1(A): GENERATED LCC PROTOCOL FOR PERSUSION DIALOGUE (PART 1)	319
C.1(B): GENERATED LCC PROTOCOL FOR PERSUSION DIALOGUE (PART 2)	320
C.2 (A): STEP 3 OF PROTOCOL GENERATION (MATCHING THE STARTING PATTERN).....	321
C.2 (B): STEP 3 OF PROTOCOL GENERATION (COMPLETING THE RECURSIVE ROLES)	322
C.3 (A): STEP 5 AND 6 OF PROTOCOL GENERATION	323
C.3 (B): STEP 7 OF PROTOCOL GENERATION (MATCHING THE TERMINATION-INTERMEDIATE PATTERN).....	324
C.3 (C): STEP 7 OF PROTOCOL GENERATION (COMPLETE THE RECURSIVE ROLES).....	325
C.3 (D): STEP 8 OF PROTOCOL GENERATION (MATCHING THE REWRITING METHODS OF THE TERMINATION-INTERMEDIATE PATTERN	326

C.8(A): GENERATED LCC PROTOCOL FOR PERSUSION DIALGOUE (PART 1)	331
C.8(B): GENERATED LCC PROTOCOL FOR PERSUSION DIALGOUE (PART 2)	332
C.8(C): GENERATED LCC PROTOCOL FOR PERSUSION DIALGOUE (PART 3)	333
C.8(D): GENERATED LCC PROTOCOL FOR PERSUSION DIALGOUE (PART 4)	334
C.8(E): GENERATED LCC PROTOCOL FOR PERSUSION DIALGOUE (PART 5).....	335
C.8(F): GENERATED LCC PROTOCOL FOR PERSUSION DIALGOUE (PART 6)	336
C.9 (A): STEP 2 OF PROTOCOL GENERATION (MATCHING THE MOVE-TO-DIALOGUE PATTERN) ..	337
C.9 (B): STEP 2 OF PROTOCOL GENERATION (MATCHING THE MOVE-TO-DIALOGUE PATTERN) ...	338
C.9 (C): STEP 3 (PART 1) OF PROTOCOL GENERATION (MATCHING THE REWRITING METHODS OF THE RECURS-TO-N-DIALOGUE PATTERN).....	339
C.9 (D): STEP 3 (PART 2) OF PROTOCOL GENERATION (MATCHING THE REWRITING METHODS OF THE RECURS-TO-N-DIALOGUE PATTERN).....	340
C.9 (E): STEP 3 (PART 3) OF PROTOCOL GENERATION (MATCHING THE REWRITING METHODS OF THE RECURS-TO-N-DIALOGUE PATTERN).....	341
C.10: THE CLAIMSENDER _p CPN SUBPAGE	342
C.11: THE CLAIMRECEIVER _o CPN SUBPAGE.....	343
C.12: THE REPLYTOCLAIMSENDER _o CPN SUBPAGE	343
C.13: THE REPLYTOCLAIMRECEIVER _p CPN SUBPAGE	344
C.14: THE REPLYTOWHYSENDER _p CPN SUBPAGE	344
C.15: THE REPLYTOWHYRECEIVER _o CPN SUBPAGE	345
C.16: THE REPLYTOARGUESENDER _o CPN SUBPAGE	345
C.17: THE REPLYTOARGUERECEIVER _p CPN SUBPAGE	346
C.18: THE PROTOCOL CPN SUPERPAGE	347
C.19: THE STATE SPACE GRAPH	348
C.20: DIALOGUE OPENING PROPERTY PAGE.....	351
C.21: TERMINATION OF A DIALOGUE PROPERTY PAGE	352
C.22: TURN TAKING BETWEEN AGENTS PROPERTY PAGE	352
C.23:MESSAGE SEQUENCING PROPERTY PAGE.....	352
C.24: RECURSIVE MESSAGE PROPERTY PAGE	353

C.25: THE VERIFICATION RESULT OF THE FIVE BASIC PROPERTIES	353
E.1: GENERATE LCC PROTOCOL TOOL MAIN SCREEN	358
E.2: DIALOGUE INTERACTION DIAGRAM LIBRARY SCREEN	358
E.3: CREATE NEW DIALOGUE INTERACTION DIAGRAM SCREEN.....	358
E.4: SIMPLE DID GRAPHICAL REPRESENTATION OF A PERSUASION DIALOGUE.....	360
E.5: DID FORMAL REPRESENTATION OF AN INQUIRY DIALOGUE.....	360
E.6: FULL DID GRAPHICAL REPRESENTATION OF A PERSUASION DIALOGUE.....	362
E.7 (A): HOW TO READ DID.....	364
E.7 (B): HOW TO READ DID.....	365
E.8: ADD NEW ARGUMENT SUBSCREEN	366
E.9: ADD NEW CONDITION SUBSCREEN.....	366
E.10: DID TEXTUAL REPRESENTATION	367
E.11: DID TEXTUAL REPRESENTATION OF CLAIM LOCUTION.....	369
E.12: CREATE NEW DIALOGUE INTERACTION DIAGRAM SCREEN	370
E.13: OPEN DID FILE DIALOG BOX.....	370
E.14: GENERATES CONCRETE LCC PROTOCOLS FOR PERSUASION DIALOGUE	372
E.15: GENERATES CONCRETE LCC PROTOCOLS FOR PERSUASION DIALOGUE AMONG N-AGENT ..	373
E.16: SHOW GENERATED LCC PROTOCOLS SCREEN	374

List of Tables

2.1: THE ABSTRACT SYNTAX OF LCC	9
3.1: DIALOGUE TYPES	50
5.1 : BROADCASTING PATTERN ROLES ARGUMENTS	135
5.2 (A): BROADCASTING PATTERN FUNCTIONS	136
5.2 (B): BROADCASTING PATTERN ROLES FUNCTIONS.....	137
5.3: RELATIONSHIP BETWEEN LOCUTION TYPE AND PATTERNS	144
6.1: LCC-CPNXML AUTOMATIC TRANSFORMATION TABLE (ROLE).....	156
6.2:LCC-CPNXML AUTOMATICALLY TRANSFORMATION TABLE (SEND A MESSAGE)	157
6.3: LCC-CPNXML AUTOMATICALLY TRANSFORMATION TABLE(RECEIVE A MESSAGE)	158
6.4: LCC-CPNXML AUTOMATICALLY TRANSFORMATION TABLE (THEN KEYWORD AND CHANGE ROLE)	159
6.5:LCC-CPNXML AUTOMATICALLY TRANSFORMATION TABLE (OR KEYWORD)	161
6.6: LCC-CPNXML AUTOMATICALLY TRANSFORMATION TABLE (DIALOGUE TOPIC)	162
6.7: LCC-CPNXML AUTOMATICALLY TRANSFORMATION TABLE (STARTER ROLE ARGUMENTS) ...	163
6.8: LCC-CPNXML AUTOMATICALLY TRANSFORMATION TABLE (END STATEMENT)	165
6.9 (A):LCC-CPNXML AUTOMATICALLY TRANSFORMATION TABLE (GET AN ARGUMENT CONDITION)	166
6.9 (B):LCC-CPNXML AUTOMATICALLY TRANSFORMATION TABLE (GET AN ARGUMENT CONDITION)	167
6.9 (C):LCC-CPNXML AUTOMATICALLY TRANSFORMATION TABLE (GET AN ARGUMENT CONDITION)	168
6.10: LCC-CPNXML AUTOMATICALLY TRANSFORMATION TABLE (ROLE IN THE CPN SUPERPAGE).170	
6.11: LCC-CPNXML AUTOMATICALLY TRANSFORMATION TABLE (DIALOGUE TOPIC IN THE SUPERPAGE).....	171
6.12: LCC-CPNXML AUTOMATICALLY TRANSFORMATION TABLE (AGENT'S STARTER ROLE ARGUMENTS IN SUPERPAGE).....	172
8.1 DIFFERENCES BETWEEN MODGIL AND MCGINNIS [MODGIL AND MCGINNIS, 2007], REED ET AL. [REED ET AL., 2008] AND DID.....	209
B.1 RELATIONSHIP BETWEEN LOCUTION TYPE AND PATTERNS	307

Chapter 1

Introduction

An argument offers a reason for believing a statement, taking an action, changing a goal, etc. Recently, argumentation has been an important area of research in natural language processing, knowledge representation, and construction of automated reasoning systems [Maudet et al., 2007]. It also has merged with multi-agent systems (MAS), in particular for modelling the communication between agents, where it supports mechanisms for designing, implementing, and analyzing models of the interaction among agents. However, a wide ranging approach of this kind carries with it various challenges. An important challenge is to ensure that agent arguments can be communicated in a reliable way by using argument-based protocols.

The Argument Interchange Format (AIF) [Chesnevar et al.,2007;Willmott et al., 2006] is an approach that has been used successfully to tackle this challenge. Recognizing that no single style of argumentation fits all circumstances, the AIF stipulates a layered style of specification in which a high-level language is used to specify the argument which is then implemented as a protocol.

Interaction protocols in AIF [Chesnevar et al.,2007;Willmott et al., 2006] can be represented using a protocol language called the Lightweight Coordination Calculus (LCC) [Robertson, 2004; Hassan et.al., 2005], an executable specification language which is at the core of an overall architecture for coordination of MAS.

The goal of this research is to develop a useful tool that can enable designers to build an efficient LCC program in the easiest and quickest manner. The aim is to propose a high-level control flow specification language, called a Dialogue Interaction Diagram (DID) between AIF and LCC, for designers to build an agent by reusing common LCC argumentation patterns. The selection and instantiation of these patterns is performed automatically given a high-level specification ideally written in the AIF.

1.1 The Challenge

Today, argumentation [Rahwan and Moraitis, 2009] is gaining more prominence because it is being used as a key form of interaction among agents in MAS. However, the argumentation community encounters various problems, such as the lack of a shared interchange format for arguments. Arguments [Rahwan and Moraitis, 2009] are represented in many different ways depending on the particular approach people used. To solve this problem, the argumentation community developed the AIF [Chesnevar et al.,2007;Willmott et al., 2006], which provides an abstract language to exchange argumentation concepts among agents in a MAS.

However, AIF [Chesnevar et al.,2007;Willmott et al., 2006] is an abstract language that does not capture some concepts that are needed to support the interchange of arguments between agents (e.g. sequence of argument, locutions and pre- and post-conditions for each argument). Rather, AIF only specifies the properties that define an argument without prescribing how that argument may be made operational. An example of this problem occurs in one of the basic dialogue games stereotypes: *A1* and *A2* are reasoning about whether a particular penguin, Tweety, can fly:

A1) Tweety flies. (*making a claim*);

A2) Why does Tweety fly? (*asking for grounds for a claim*);

A1) Tweety is a bird, birds generally fly. (*arguing: offering grounds for a claim*);

A2) Tweety does not fly because Tweety is a penguin, penguins do not fly. (*stating a counterargument*);

A1) You are right.(*conceding an argument*).

In this dialogue game each agent responds in turn to the argument made by other agent. This flow of the dialogue is not captured by AIF (e.g. AIF does not record that a given argument has been made in response to an earlier argument). AIF only captures argument structures (e.g. it connect "Tweety flies" with its premises "Tweety is a bird, birds generally fly"). (see chapter 3 for a detailed discussion of

this problem). This means that there is a gap between argument specification languages and multi-agent implementation languages. The objective of this thesis is to fill this gap using a combination of automated synthesis and verification methods. The following sections provide an introduction to these methods.

1.2 The Proposed Approach

The main research question is:

"Can we automatically synthesise multi-agent protocols (LCC as an operational specification language) from a high-level argument specification languages (AIF as a high-level specification language)?"

This research presents an approach to solve the described argument implementation challenge. It demonstrates how a generic argumentation representation (acting as a high-level specification language) can be used to automate the synthesis of executable specifications in a protocol language capable of expressing a class of multi-agent social norms. As our argumentation language we have chosen the AIF. As our protocol language we have chosen LCC.

This approach has two main tasks (parts):

- (1) Bridging the gap between AIF and LCC by using transformational synthesis methods:
 - a) Extending the AIF diagrammatic notation (since AIF is an abstract language and fully automated synthesis starting only from the AIF is not possible) to give a new, intermediate recursive visual high-level language between the AIF and LCC. The new high-level specification language remedies the AIF problem and represents the dialogue game protocol rules in abstract way.
 - b) Implementing a tool which automatically synthesise concrete LCC protocols from the new high-level specification language using a new pattern-based synthesis method.

- (2) Checking the semantics of the new high-level specification language, used as a starting point, against the semantics of the synthesised LCC protocol.

1.3 Approach

Our approach attempts to close the gap between standard argument specification and deployable protocols by automating the synthesis of protocols, in LCC, from argument specifications, ideally written in the AIF. It consists of two parts (as shown in Figure 1.1):

Part one which is used to bridge the gap between AIF and LCC by using a transformational synthesis. Part one was built in two stages:

- (1) Proposing a new high-level specification language, between the AIF and LCC, for multi-agent protocols called a DID;
- (2) Synthesising concrete LCC protocols from DID specifications (automatically synthesising LCC protocols from DID specifications by recursively applying LCC-Argument patterns).

Part two provides a verification methodology based on Standard functional programming language (SML) and Colored Petri Net (CPNs) to verify the semantics of the original DID specification against the semantics of the synthesised LCC protocol.

1.4 Claims of Novelty

This thesis contributes to the area of multi-agent argumentation protocol implementation. Firstly, it extends the AIF diagrammatic notation to give a new, intermediate recursive visual high-level language between the AIF and LCC called a DID. It does this to remedy the AIF obstacle (AIF is not an executable language). The goal is to be able to represent, in an abstract way, dialogue game protocol rules. Second, it introduces LCC-Argument patterns. It uses LCC-Argument patterns with DID to fully automated synthesis multi-agent protocols. Finally, it introduces

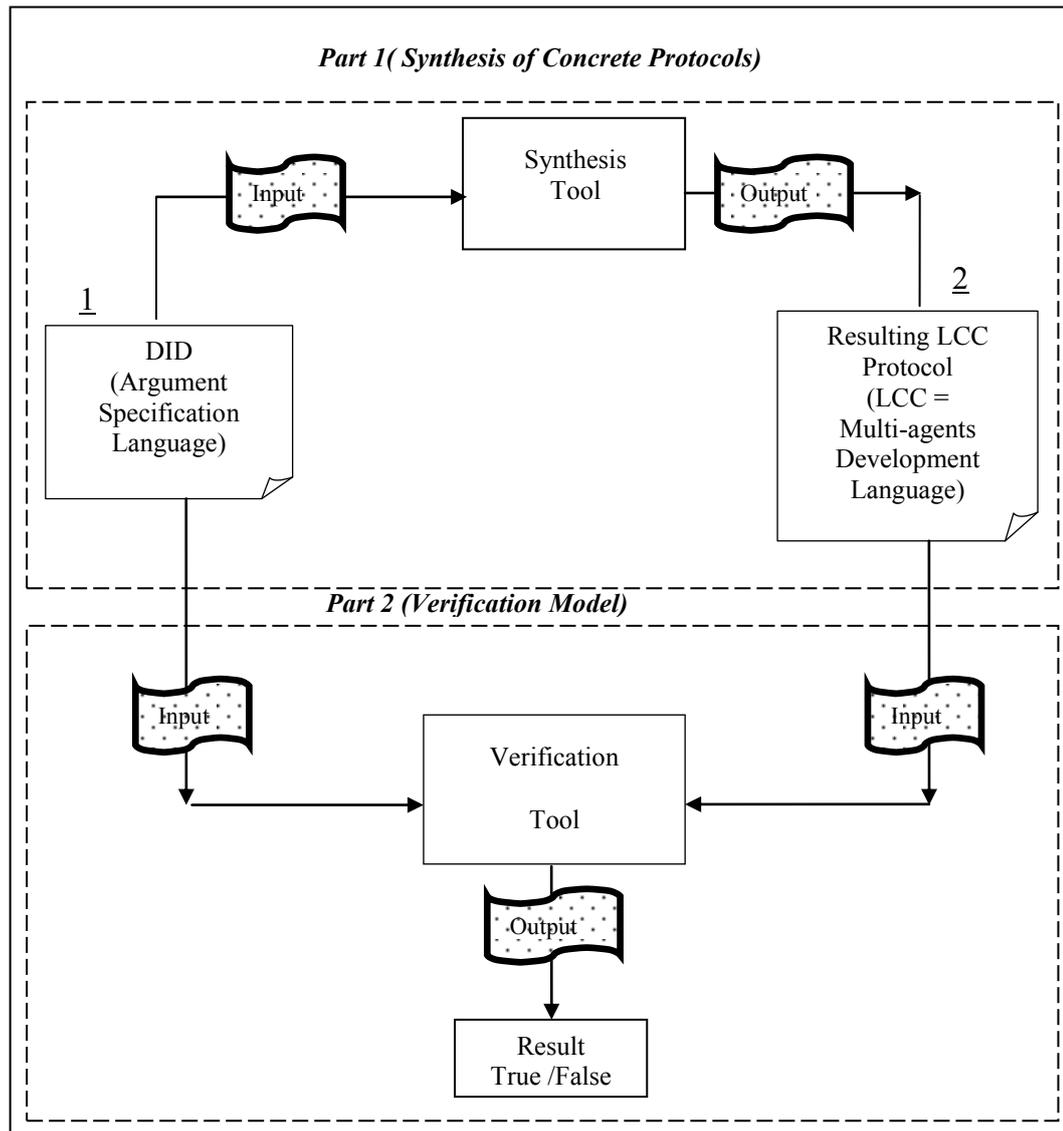


Figure 1.1: System Architecture

verification methods to verify the semantics of the DID specification, used as a starting point, against the semantics of the synthesised LCC protocol. The remaining chapters of this thesis illustrate how this may be accomplished.

1.5 Thesis Structure

The rest of this thesis is structured as follows:

- Chapter 2: Background and Literature Review. This chapter reviews research related to our representation approach.

- Chapter 3: Argumentation, Dialogue Games and MAS. This chapter introduces the basic concepts of arguments, argumentation, dialogue games and AIF. It also summarises the advantages of using argumentation for modelling agent communication, as well as the shared and the implementation problems faced by argumentation community and the requirements we need in order to solve these problems.
- Chapter 4: Argument Specification Language. This chapter proposes a new high-level specification language, between the AIF and LCC, for multi-agent protocols called a DID, which is used to specify the dialogue game protocol in an abstract way.
- Chapter 5: Synthesis of Concrete Protocols. This chapter proposes a set of LCC–Argument patterns and describes a fully automated synthesis method which can automatically synthesise LCC protocols from DID specifications by recursively applying LCC-Argument patterns.
- Chapter 6: Verification Method based on Standard functional programming language (SML) and Colored Petri Net (CPNs). This chapter proposes a verification methodology based on SML and CPNs used to evaluate the research hypothesis.
- Chapter 7: Design and Implementation. This chapter presents the architecture and the prototype implementation of the represented approach, that is used to synthesise concrete LCC protocols from DID specifications by recursively applying LCC-Argument patterns.
- Chapter 8: Evaluation and Discussion. This chapter discusses and summarises the main contributions of this thesis. It also points out limitations of the thesis.
- Chapter 9: Conclusions and Future work. This chapter summarises the thesis and discusses the main significance, contribution and limitations of the current work. It also outlines future research work.

Chapter 2

Background and Literature Review

This chapter provides an overview and background of previous work on topics related to this thesis. Given the extensive literature on these topics, we limit the discussion to areas that are most relevant to later chapters.

We open this chapter with a summary of agent protocol development language related works in Section 2.1. This is followed by a description of research on design patterns in Section 2.2. Section 2.3 introduces research on relevant verification methods. Lastly, Section 2.4 summarises this chapter.

2.1 Agent Protocol Development Language

The approaches presented in this thesis began with Argument Interchange Format (AIF) which provides a common language to exchange argumentation concepts among agents in a MAS.

To support formal analysis and verification, the AIF community [Willmott et al., 2006; Modgil and McGinnis, 2007] (see chapter 3 for more information about AIF) suggests using a process¹ and declarative² language to implement the dialogue games protocol. For this reason we chose the Lightweight Coordination Calculus (LCC) [Robertson, 2004; Hassan et.al., 2005], a declarative, process calculus-based,

¹ Process language: Process calculi [Baeten,2005] provide a tool to describe the behaviour of agents or processes interactions or communications by algebraic means in a high-level way. It allows formal reasoning and process verification.

² Declarative language: According to Lloyd [Lloyd, 1994] "declarative programming involves stating what is to be computed, but not necessarily how it is to be computed".

executable specification language for choreography³ which is based on logic programming and is used for specifying the message-passing behaviour of MAS interaction protocols.

LCC is based on process calculus, in the sense that it determines when and what actions the agent can perform and under what circumstances these actions may be carried out. In other words, LCC restricts each agent's behaviour in the dialogue by specifying the rules of the dialogue game. It controls what messages can be received or sent, in what order these messages may be received or sent, and under what pre-conditions and post-conditions these messages may be sent or received [Grivas, 2005].

In addition, LCC is a declarative language, in the sense that it only describes the interaction between agents (what to do, not how to do it) and can be understood independently from any specific execution architecture. It also contains few operators, which make LCC a compact language for agent interaction [Willmott et al., 2006; Modgil and McGinnis, 2007].

LCC is also an executable specification language (a very high-level executable programming language) in the sense that there is a deployment mechanism for LCC agent protocols [Grivas, 2005].

2.1.1 LCC Syntax

The abstract syntax of an LCC clause [Robertson, 2004; Hassan et.al., 2005] is shown in Table 2.1. In an LCC framework each of the $N \geq 2$ agents is defined with a unique identifier *Id* and plays a *Role*. Each agent, depending on its *Role*, is assigned an LCC protocol.

³ Choreography: According to Dijkman and Dumas [Dijkman and Dumas, 2004] "Choreography is collaboration between some service providers and their users to achieve a certain goal. It only describes tasks that involve communication between the parties involved, and not tasks performed internally."

	Meaning
Framework :=	{Clause,...}
Clause :=	Agent ::= Dn
Agent :=	a(Role, Id)
Dn :=	Agent Message null \leftarrow Constraint Dn then Dn Dn or Dn
Message :=	M \Rightarrow Agent M \Rightarrow Agent \leftarrow Constraint M \Leftarrow Agent Constraint \leftarrow M \Leftarrow Agent
Constraint :=	Term Constraint and Constraint Constraint or Constraint
Role :=	Term
M :=	Term
Term:=	Constant (Argument,.....)
Id	Constant Variable
Constant	Character sequence made up of letters or numbers beginning with a lower case letter
Variable	Character sequence made up of letters or numbers beginning with an upper case character
Argument	Term Constant Variable

Table 2.1: The abstract Syntax of LCC

AN LCC protocol can be recursively defined as a sequential composition (denoted as *then*) or choice (denoted as *or*) of LCC protocols. In an LCC protocol, agents can change roles, exchange (receive or send) messages and exit the dialogue under certain constraints C ($\text{null} \leftarrow C$). Null represents an event (a do-nothing event) that does not involve role changing or message exchanging. A constraint is defined as a propositional formula specified over *terms* connected by *or* and *and* operators.

Messages M are the only way to exchange information between agents. An agent can send a message M to another agent ($M \Rightarrow \text{Agent}$), and receive a message from another agent ($M \Leftarrow \text{Agent}$). There are two types of constraints over the messages exchanged: pre-condition and post-condition. Pre-conditions ($M \Rightarrow \text{Agent} \leftarrow C$) specify the required conditions for an agent to send a message. Post-conditions ($C \leftarrow M \Leftarrow \text{Agent}$) explain the states of the receiver after receiving a message. An agent can test the satisfaction of the constraints either privately (by using the internal agent's mechanism) or by using shared knowledge transferred via messages.

An agent can play more than one role during several interactions. In LCC, recursion can be achieved by repeating the same role either to process a list or to loop it until the recursive condition fails.

LCC has a Prolog like syntax [Besana, 2009]:

- (1) Constraint name are character sequence made up of letters or numbers beginning with a lower case letter;
- (2) Variable are character sequence made up of letters or numbers beginning with an upper case character;
- (3) Constraints are analogues to Prolog queries (Although LCC itself does not assume that the constraint solver must be a Prolog system);
- (4) Some of the role parameters are input and others are output parameters. The values of output parameters are set when the role ends;
- (5) The semantics of the assignment and the comparison of variables is taken from Prolog: an assignment to an un-instantiated variable always succeeds by putting the value in the variable (simple assignment action), whereas an assignment to an instantiated variable succeeds if, and only if, the values of the two variables are the same (comparison action).

2.1.2 LCC Examples

This section illustrates three simple and complex examples, which demonstrate the use of LCC as a specification language for specifying the message-passing behaviour of MAS interaction protocols:

Example 1: Simple Persuasion Protocol

This is the simplest example of a persuasion protocol between two agents *P* and *O*. *P* and *O* have arguments for and against *Topic*. Agent *P* sends a *claim* message *Topic* and agent *O* receives this *claim* message *Topic*. A fragment of LCC protocol for the interchange in this argument is:

```

a(R1,P)::=
  claim(Topic) => a(R2, O)
  then
    a(R3,P).

a(R2,O)::=
  claim(Topic) <= a(R1, P)
  then
    a(R4,O).

```

This is read as: role $R1$ of agent P sends a claim message to the role $R2$ of agent O and then role $R2$ of agent O receives the claim message from role $R1$ of agent P . Then P changes its role to $R3$ and O changes its role to $R4$.

Example 2: Buying and Selling

In this example (adapted from [Besana, 2009]), there are two parties: buyer and seller. The buyer wants to buy an item R .

```

a(buyer, A)::=
  request(R) => a(seller, B) ← need(R)
  then
    price(Y) <= a(seller, B)
    or
    failure <= a(seller, B).

a(seller, B)::=
  request(R) <= a(buyer, A)
  then
    price(Y) => a(buyer, A) ← find(R,Y)
    or
    failure => a(buyer, A).

```

This is read as: the buyer role of agent A satisfies the constraint $need(R)$ (the request for the item that the seller needs to provide), and then sends the request message with the needed item to the seller of agent B and waits for agent A to reply (the buyer waits for one of the two messages: $price(Y)$ or $failure$). Then, the seller, receives the request message, tries to satisfy the constraint $find(R,Y)$ (finds the item), and then either replies with the item price or sends a failure message if the constraint $find(R,Y)$ cannot be satisfied.

Example 3: An Auction

In this example (adapted from Besana and Barker works [Besana and Barker, 2009]), there are N agents: A which is considered to be an auctioneer and more than one agent B , which are considered as bidders.

```

a(auctioneer(Product,Bidders), A) ::=
a(caller(Product, Bidders),A)
then
  a(waiter(Bidders, Bids, curwinner(nul, 0),A)
  then
    sold(Product,Price) => a(bidder,WB) ← curwinner(WB, Price) = Winner.

a(caller(Product,Bidders), A) ::=
null ← Bidders = [ ] %no bidders left
or
  [ invite_bid(Product) => a(bidder, BH) ← Bidders = [BH|BT]
    then
      a(caller(Product, BT), A). %recursion ]

a(waiter(Bidders, Bids, curwinner(WinBidder, WinBid), A) ::=
null ← allarrived(Bids, Bidders) and Winner = curwinner(WinBidder, WinBid)
or
null ← timeout( ) and Winner = curwinner(WinBidder, WinBid)
or
  [ bid(Product,Offer) <= a(bidder, B)
    then
      [ a(waiter(Bidders, [B|Bids], curwinner(B, Offer), A) ← Offer > WinBid
        or
        a(waiter(Bidders, [B|Bids], curwinner(WinBidder, WinBid), A) ]
    ]

or
a(waiter(Bidders, Bids, curwinner(WinBidder, WinBid), A) ← sleep(1000).

a(bidder, B) ::=
invite_bid(Product) <= a(caller, A)
then
bid(Product, Offer) => a(waiter, A) ← bid_at(Product, Offer)
then
sold(Product, Price) <= a(auctioneer(Product,Bidders), A).
    
```

The *auctioneer* role of agent A has two input parameters: *Product* to sell and the list of *Bidders*. The *auctioneer* role starts by changing its role to *caller*. The *caller* role of agent A recurses over the *Bidders* list. If the list is empty, it returns null, otherwise, it

sends the *invite_bid* message to one bidder (at the head of the Bidders' list) and then it recurses over the remaining bidders. The *caller* role ends once the *invite_bid* message is sent to all the bidders (*Bidders* = []).

Afterwards, the control changes to the *auctioneer* role which then changes its role to *waiter*. The *waiter* role of agent *A* has one input parameter: *Bidders*, and two output parameters: (1) *Bids* (*Bids* represents the list of replied bidders); (2) *Winner* (*Winner*=*curwinner*(*WinBidder*,*WinBid*) where *WinBidder* represents bidder's ID and *WinBid* represents bidder's offer). The values of output parameters are set when the role *waiter* ends. The *waiter* role begins by checking if all the replies have arrived (all the bidders have replied to the *invite_bid* message) or if the period has expired (*timeout*() = true). If either condition is true, then the *waiter* role assigns the current winner as the final winner. Otherwise, the *waiter* role receives a message from a bidder (there is a message in the receiving message queue) and checks if the bidder's offer is higher than the current highest offer. If this condition is true, the *waiter* role recurses to make the current bidder the current winner, otherwise it simply recurses. The *waiter* role then waits for a second (*sleep*(1000)) and recurses, if there is no message in the receiving message queue.

At the same time, the bidder role of agent *B* receives the request to bid, and sends the offer to the *waiter* role of agent *A*. Then, if the offer is successful (the current bidder is the final winner), the bidder role receives a *sold* message from the *auctioneer* role of agent *A*. If the offer is unsuccessful, then the interaction between agent *A* and *B* will end.

2.2 Design Pattern

To support agent protocol development activities, this thesis proposes LCC-Argument design patterns. Design patterns, which are common and recurring code patterns of a specific programming language [Gamma et.al, 1995], have been extensively studied within the object-oriented and logic programming community. This section summarises the software engineering, the agent protocol and the logic programming community view of design patterns and how they have been used in

software development. It also compares our LCC-Argument patterns with the literature.

2.2.1 Software Engineering Design Pattern

Object-oriented software engineering [Gamma et.al, 1995] uses the definition of patterns as proposed by the architect Christopher Alexander [Alexander et.al, 1977] to define the design pattern:

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".

In a practical sense, design patterns are generic and recurring solutions to common problems. However, they are not finished code that can be used directly. In essence, design patterns describe how to solve some problems that are dependent on a particular language, such as Java, but are independent of any particular algorithm or problem domain, and can be reused in many different situations. These patterns can help to speed up the development process by allowing a set of tested and proven patterns to be reused in order to solve a given problem.

2.2.2 Agent Protocol Design Pattern

Patterns for agent protocols are different from traditional object-oriented design patterns because objects and agents do not relate to the same logical and conceptual levels [Sauvage, 2004; Odell, 2002]:

- (1) Agents are dynamic, autonomous and intelligence whereas objects are conventionally passive.
- (2) Agents communicate in a different way than object. They have the ability to communicate with their environment and other entities.

However, most of the implemented agent protocols [Deugo and Weiss, 1999] are implemented using object-oriented languages (such as Aglets⁴ and Voyager⁵ frameworks which are implemented using Java). Consequently, the structure of most agent protocol patterns [Deugo and Weiss, 1999; Aridor and Lange, 1998; Tolksdorf, 199; Paschke et.al, 2006] are similar to the structure of object-oriented design patterns.

Object-oriented design patterns usually describe relationships and interactions between objects and classes to solve general object-oriented design problems without identifying the software classes or objects involved.

An Example

An example of an agent design pattern (similar to Object-Oriented pattern) is from Aridor and Lange [Aridor and Lange, 1998] work. Aridor and Lange [Aridor and Lange, 1998] represent a set of new different mobile agent design patterns, which can be used to generate mobile agent applications. They classify patterns into three types: travelling, task, and interaction patterns.

One example of Aridor and Lange [Aridor and Lange, 1998] patterns is *Master-Slave* pattern (see Figure 2.1) from the group of task patterns. This pattern implemented as an aglet. It defines how master agent can assign a task to a slave agent. It has two abstract classes:

- (1) *Master* class, which has one abstract method *getResult*. The *getResult* method defines how to handle the task's result.
- (2) *Slave* class, which has two abstract methods:

⁴ <http://aglets.sourceforge.net/>

⁵ <http://www.pegacat.com/vcf/>

```

public abstract class Master extends Aglet
{
    public void onCreation(Object obj)
    {
        // Called when the master is created.
    } //end of onCreation function

    public void run ( )
    {
        getResult( )
    } // end of run function

} //end of Master class

public abstract class Slave extends Aglet
{
    Object result = null

    public void onCreation(Object obj)
    {
        // Called when the slave is created. Gets the remote destination, a reference to
        // the master agent, and other specific parameters.
    } //end of onCreation function

    public void run ( )
    {
        // At the origin:
        initializeJob( );
        dispatch(destination); // Goes to destination
        // At the remote destination:
        doJob( ); // Starts on the task.
        result=...;
        // Returns to the origin.
        // Back at the origin.
        // Delivers the result to the master and dies.
        dispose( );
    } // end of run function

} //end of Slave class

```

Figure 2.1: The Slave Class

- i. *initializeJob* method, which defines the initialization steps to be performed before the agent travels to a remote destination;
- ii. *doJob* method, which defines the concrete task to be performed at the remote destination.

A second example of an agent design pattern (similar to Object-Oriented pattern) is from Tolksdorf [Tolksdorf,1998] work. Tolksdorf [Tolksdorf,1998] describes five patterns which rely on some mobility mechanism of information (which are used to manage the exchanging -accessibility dependencies- of knowledge between users, systems and agents). These patterns, called "coordination patterns", can be used to generate agent protocols that can manage dependence in organisation, economic, and computing systems.

Both Aridor and Lange [Aridor and Lange, 1998] and Tolksdorf [Tolksdorf,1998] patterns are expressed in terms of classes and objects. However, our solutions (LCC-Argument patterns) are expressed in terms of roles. Our proposal can be interpreted as an adaptation of object-oriented design patterns in order to capture the different relationships and interactions between agents' roles.

Essentially, agent role design patterns (LCC-Argument patterns) are similar to object-oriented design patterns. The only difference between them is the structure of an agent role pattern which is described by using the notions of roles instead of the notions of classes and objects. In fact, we use the notation of the roles since our protocol language is LCC which is not considered to be an object-oriented language and uses roles (instead of classes and objects) to describe agent protocols (see section 2.1 for more details).

2.2.3 Logic Programming Patterns (Logic Programming Techniques)

Since LCC has a Prolog like syntax (see section 2.1.1), in this section, we give a summary of Prolog programming techniques (logic programming patterns), Techniques editing and Grivas structured design methods. The general idea of logic programming techniques is analogous to that used in Techniques editing [Bowles et.al, 1994], to synthesise Prolog clauses, as summarised below.

2.2.3.1 Prolog Programming Techniques

Programming Techniques [Bowles et.al, 1994] uses common code patterns (loosely called techniques), which depend of a particular language such as Prolog but are

independent to any particular algorithm or problem domain. It provides generalised pieces of code, which can be used by software engineers to implement part of a specification.

An Example

An example of a technique taken from [Bowles et.al, 1994], is to consider the standard implementation of reverse in Prolog:

```
rev( [ ] , R , R ).  
rev( [H|T] , R0 , R ) :-  
  rev( T , [H|R0] , R ).
```

This predicate consists of two parts:

- (1) A part which performs the recursion down the list:

```
rev([ ],...)  
rev([H|T],... ) :-  
  rev(T,...).
```

- (2) An accumulator pair [O'Keefe,1990] part which builds a list during the recursion and passes the result to the top of the recursion:

```
rev([ ],R,R)  
rev(...,R0,R) :-  
  rev(...,[H|R0],R).
```

These two parts are considered to be Prolog techniques because they are general common patterns, which can be used in a wide range of domains irrespective of the algorithm being implemented.

A summary of methodology for building programs using techniques is given in [Kirschenbaum at.al, 1989]:

- (1) This methodology constructs a program by using a set of syntactic entities (skeletons), which describe the common control flow pattern of the program.

- (2) This methodology also constructs a set of syntactic methods (techniques or additions), which perform simple tasks such as adding parameters.
- (3) Additions and techniques can be applied to the skeletons yielding extensions (extra parameters, goals or clauses).
- (4) The final program is obtained by composing extensions.

The idea of building programs is to define the set of suitable skeletons to solve the problem. In this way, the software engineer can choose one skeleton from this set that suits his needs. Next, the software engineer can apply additions (or techniques) to the skeleton. Finally, the software engineer can repeat the process of applying additions (or techniques) until the final program is obtained.

The concept of Prolog programming techniques has been developed and applied in a variety of contexts. The most interesting context is techniques editing.

2.2.3.2 Techniques editing

Techniques editors can speed up the program-building process by reusing a set of skeletons that solves a given problem. The idea of techniques editors has been proposed in two editors: Robertson's editor [Robertson, 1991] and Ted [Bowles, 1994].

Robertson's editor is based directly on methodology that is given in [Kirschenbaum et.al, 1989] as illustrated above. The editor aims to support primary novice users. It provides a set of Prolog skeletons, additions, and other information that allows the editor to guide and judge the user. The user can construct the program by selecting a skeleton and then apply additions onto the selected skeleton. This editor is limited by a small set of skeletons and additions. Its interface is not sophisticated but it provides a basic set of editing operations and some basic guidance in the editing process.

The second editor, Ted, also aims to support novice users, but its technique is different from the skeleton-addition approach. Ted common patterns capture the relationships between the head and recursive arguments in the recursive clauses of a

program. An example of Ted patterns is *Same Technique* [Bowles et.al, 1994], which passes the same value between two argument positions: the head of a clause and a recursive subgoal in the clause. (Note: in this example the technique appears underlined).

```
rev([],R,R).  
rev([H|T],R0,R):-  
rev(T,[H|R0],R).
```

The Ted editor has a number of limitations in the patterns. Most notably, that it does not support both mutually recursive predicates⁶ and doubly recursive clauses⁷ a long with the fact that its data-structures are limited to three types: lists, atoms, and numbers.

The Ted editor has a graphical interface (point and click interface) and provides the same amount of information as that provided by Robertson's editor. It also has the ability to map the arguments and check their suitability. However, it does not have the ability to guide the user through the editing process.

Despite the limitations in both editors, they were tested on user groups. Ted in particular was used in controlled experiments with novice programmers (those using Ted tended to build programs faster and with fewer errors).

2.2.3.3 A Structural Synthesis System for LCC Protocols

Grivas' project [Grivas, 2005] developed a structured design editor for LCC protocol (SDE). It aims to define a set of common LCC patterns, which can be reused to make the LCC protocol-building process faster and easier by requiring less knowledge and effort from the software engineer. In particular, Grivas' project attempts to use similar techniques to Prolog techniques editing.

⁶ Mutually recursive predicate: [Krauss,2008] "If two or more functions call one another mutually, they have to be defined in one step". An example of mutually recursive predicate is: $p \rightarrow *q$, $q \rightarrow *p$.

⁷ Double recursion: Double recursion [Odifreddi and Cooper, 2012] "allows the recursion to happen on two variables instead of only one".

Grivas' project found that a direct use of Prolog technique editing approaches in the LCC case is not easy because of the differences between Prolog and LCC languages (LCC syntax is similar to Prolog but LCC tackles different problems from those of Prolog). The idea is to come up with a set of skeletons by using process-oriented methods and then extend the design using similar techniques to those employed in Prolog.

Three different types of patterns were identified in this project. The first type of pattern, called *Skeletal*, describes the general structure of the clause where the details of the clause can be specified later either manually or by applying another pattern.

An example of this pattern follows:

```
a(R,X) ::
  ( <def>
    then a(R,X) )
  or
  null ← <con>
```

This example represents a general recursive clause that can be applied to different clauses. R represents role name, X represents agent identifier, $\langle def \rangle$ represents unspecified definition, and $\langle con \rangle$ represents unspecified conditions.

The second type of pattern, called *Role Refinement*, describes the clause in more detail and is used to refine the clause.

An example of this pattern is as follows:

```
a(F(A1...An),X) ::
  (
    <def>
    then a(F(A1...An-1,An'),X)
  )
  or
  null ← <con>
```

This example represents a recursive clause in more detail than the *Skeletal* example. F represents the role name and $A1 \dots An$ represents role arguments.

The third type of pattern, called *Clause Interaction*, describes the interaction between two clauses. It is a message passing specification pattern.

An example of this pattern is as follows:

```

a(R1,X) ::
    <def>
    then M=>R2

a(R2,Y) ::
    M<=R1
    then <def>
    
```

This example represents a message passing clause where one role sends a message and another role receives the message. $R1$ and $R2$ represent roles name, and X and Y represent the agent's identifiers while $<def>$ represents an unspecified definition.

There are two main differences between Robertson's editor and Ted, and the SDE editor. Firstly, Roberson's and Ted's editors focus on helping a Prolog learner whereas SDE aims to help software engineers by giving them a quick and easy way to build the LCC protocol. Secondly, SDE considers patterns as reusable LCC code, which can be useful when building protocols because that it saves effort. Conversely, Robertson's and Ted editors consider patterns as primitive operations where the combinations of these patterns can produce a wide range of Prolog programs.

2.2.3.4 Comparing LCC-Argument Patterns with Logic Programming Techniques

The most notable differences between our LCC-Argument patterns and Grivas' [Grivas, 2005] patterns are:

- (1) Grivas did not base his system on a high-level language, while we used as a high-level language DID. DID provides mechanisms to represent, in an abstract

way, the dialogue game protocol rules by giving an overview of the permitted moves and their relationship to each other (see chapter 4 for more details).

- (2) Grivas describes very small scale patterns of LCC protocol systems (operating at individual clause level) which required quite a lot of expertise from the user (engineers) in order to put them together, while our patterns are large scale patterns which bring more structure at one time (across entire LCC protocols) and specific to argumentation. Our patterns allow larger LCC components to be synthesised from smaller specification and do not require extensive low-level (coding) skill;
- (3) Grivas' patterns are inspired by Prolog Techniques editing, while our patterns have their origins in object-oriented patterns. We do not claim that our approach is better but we prefer to use the object-oriented approach over the Prolog Techniques editing approach. Essentially, we choose to work with the object-oriented patterns approach because it allows us to build the LCC roles in one step, whereas Prolog Techniques editing (Grivas' patterns) solve the problem (build LCC roles) by using an incremental approach in which missing parts of an LCC clause can be filled in (refined by) with another pattern or LCC statements (see [Grivas, 2005] chapter 4, page 22-29).

2.3 Verification Method based on SML and Coloured Petri Net

This thesis presents a verification method based on Standard functional programming language⁸ (SML) and Coloured Petri Net (CPNs). This method transforms the generated LCC protocol to CPNs models. The generated CPNs models which can then be used to check the validity of various concurrent behaviour properties of the resulting LCC protocol by using state space techniques and CPN SML language. This section gives an introduction of CPNs model models, explains a tool to specify

⁸ SML[Milner *et al.*, 1997] "SML is a general-purpose, modular, functional programming language with compile-time type checking and type inference."

and simulate CPNs models called CPN Tool, and roughly summarizes some related work which use SML and CPNs model to simulate, analyse the dynamic behavior and verify the semantics of their system.

2.3.1 Coloured Petri Nets (CPNs)

CPNs [Jensen, 1992; Jensen *et al.*, 2007; Kristensen *et al.*, 1998] is a high-level formal modelling language which can be used to model concurrent, distributed and complex systems such as communication protocols [Suriadi *et al.*, 2009; Floreani *et al.*, 1996]. An example of such systems are multi-agents interaction protocols.

A CPN model has a graphical representation as well as mathematical (formal) definition [Jensen, 1992] which is defines in mathematical way what will happen and when a specific event occurs in the model. The user does not need to know about the formal definition of CPN. The formal definition is used by the CPN editor (such as CPN Tool [Westergaard and Verbeek, 2002; Aalst and Stahl, 2011; Jensen *et al.*, 2007]) to check the syntax and the semantics of the CPN model, simulate, execute the CPN and to do the formal verification methods [Balbo *et al.*, 2000].

2.3.1.1 CPNs Model Elements

CPNs are Petri Nets⁹ (PNs) which have been extended with the notion of colors or types. As a variant of PN, the CPN model consists of four elements [Jensen and Kristensen, 2009; Eunice, 2005; Jensen *et al.*, 2007] (as shown in Figure 2.2): data, place, transition, and arc which describe the net structure of the CPN model. Places and transitions are called nodes. An arc is used to connect a place and a transition and to specify the data flow (the pre- and post- condition relation between transitions).

⁹ Petri Nets [Murata, 1989] is a mathematical, executable and graphical high level modelling language that is used for the description and analysis of concurrent distributed systems.

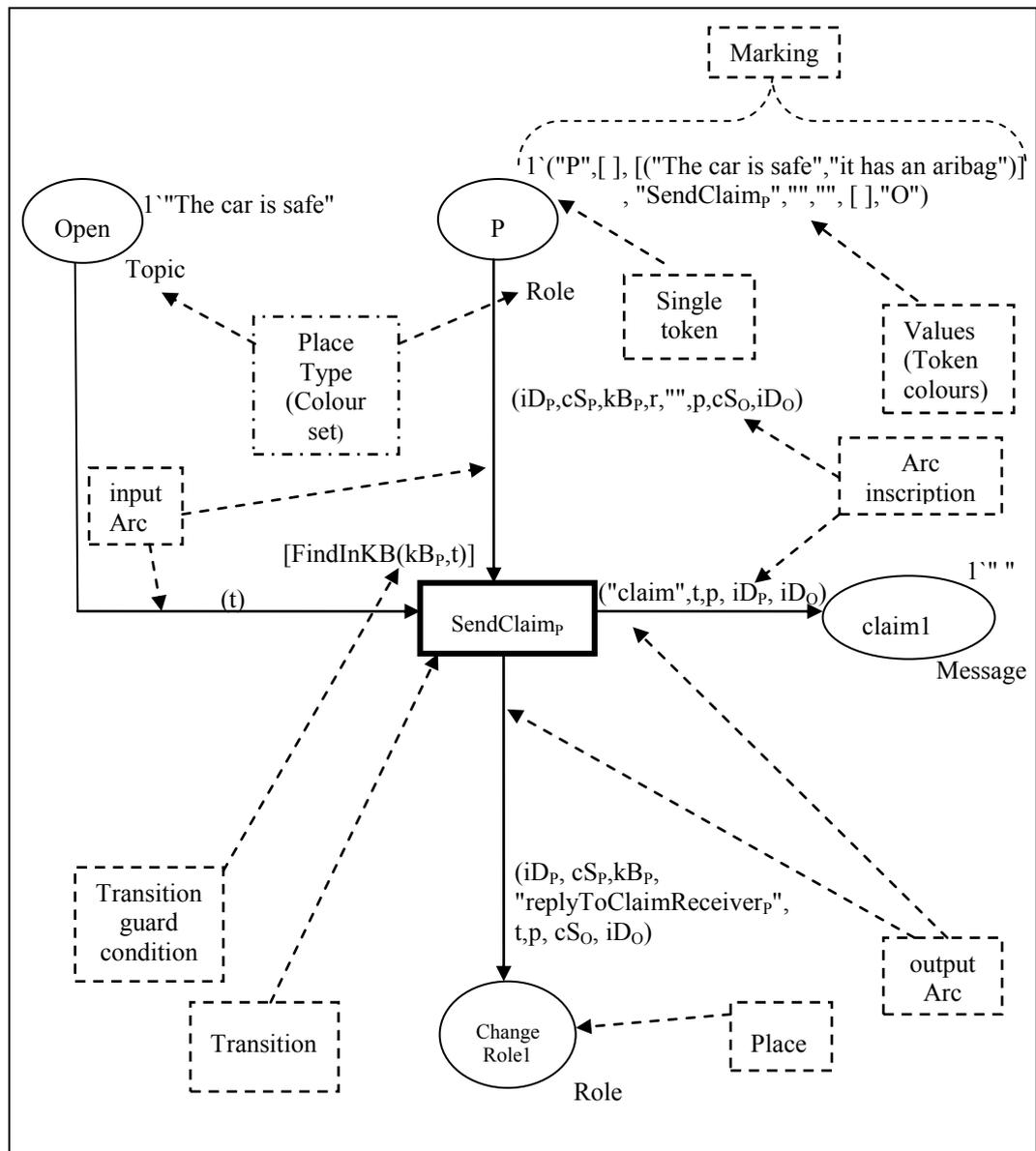


Figure 2.2 CPNs Model Elements Example

Data represents data types (colour sets), data objects (tokens) and variables. A colour set [Jensen, 1992] can be a basic colour set (integer, string, real and Boolean) or a product of colour sets or a combination of other colour sets (a declared colour set from already declared colour sets). Colour sets are used to declare variables, other colour sets, functions, operations, constants and a place's inscription. A token is associated with a colour set and has data values (token colours) attached to it.

A place is a location (drawn as ellipse). It is used to hold data items (tokens). Tokens must match the place type (colour set). A place is associated with a marking, which indicates the number of stored tokens and the value (token colours) of these tokens.

The state of the CPN model, at a particular moment, is represented by the set of markings of all the places.

A transition is an activity which represents an event and is drawn as a rectangle. It is used to transform data between places. In practice, transition receives data from one or more places, checks its guard condition, executes its associated code segment, and sends the result to other places. A guard condition is a Boolean expression enclosed in square brackets that appears above the transition rectangle. A code segment which is a computer program written in the CPN SML language (in the CPN Tool) or in the other kinds of notations which has a well-defined syntax and semantic [Jensen, 1992].

An arc is used to connect a place and a transition. It has two directions: 1) an output arc from a transition (input transition) to a place (output place); 2) an input arc from a place (input place) to a transition (output transition). An arc is associated with inscription (input inscription in an input arc or output inscription in an output arc) which is used to describe how the state of the modelled system changes. In the CPN Tool, an arc inscription is an expression that consists of CPN SML variables, constants and functions.

An example of a CPN modelled in the CPN tool is depicted in Figure 2.2. This model has:

- 1) Three colour sets (see chapter 6, section 6.1.1 for more details):
 - i. Topic colour is string data type;
 - ii. Message colour is a product type (comprising of locution, topic, premise, sender identifier and receiver identifier) used to represent message exchanges between agents;

- iii. Role colour is used to represent the agent's profile (played role, agent's identifier, agent's commitment store, agent's private knowledge based, agent's role name, topic, premise, other agent's commitment store and other agent's identifiers).
- 2) Two input places (*Open* and *P*) and two output places (*claim1* and *ChangeRole1*):
- i. The names of the places are written inside the ellipses. The place's name has no formal meaning. It has an important impact on the readability of a CPN model.
 - ii. At the bottom right hand side of each place, the colour set is written. The place *Open* has the colour set *Topic*. *P* and *ChangeRole1* places have the colour set *Role*. The place *claim1* has the colour set *Message*.
 - iii. At the upper right side of each place, the *initial marking* of the place is written. For example, the inscription at the upper right side of the place *Open* indicates that the *initial marking* of this place consists of a single token with the token colour (value) "*The car is safe*". The place *claim1* has an *initial marking* which consists of a single token with the token colour (value) "" (the empty text string) and indicates that the *initial marking* of this place has no data.
- 3) One transition called *SendClaim_P*:
- i. The name of the transition is written inside the rectangle. The transition's name as the name of the place has no formal meaning. It has an important impact on the readability of a CPN model.
 - ii. In the upper left side of the transition, the *guard condition* is written. The transition *SendClaim_P* has the *guard condition* *FindInKB(KB_P,t)*. In the CPN Tool, this condition is written in the CPN SML programming language.

- iii. When a transition *occurs* (a transition is enabled or activated when *its input places* are active and all the variables in the all surrounding *input arcs* are bound to values), the guard condition can be checked. If the condition is true, the transition removes tokens from its *input places* (which are connected to the transition by the *input arc*) and it adds tokens to its *output places* (which are connected to the transition by the *output arc*). Note that the removed tokens are determined by means of the *arc inscription*. For the example depicted in Figure 2.2, an agent can send a claim (*SendClaim_P* occurs) if an *open* place is active (there is a token in *Topic* state) and an agent playing role *SendClaim_P* is active (there is a token in state *P*).
- 4) Two input arcs and two output arcs. Each arc has an inscription (variables, constants and functions). If an inscription has variables, these variables (or functions variables) are bound to values (when the connected transition occurs) and the inscription can then be evaluated. The bounded values must have the same type as the connected place colour set. For example, the input arc, which connects the place *Open* to the transition *SendClaim_P*, has (*t*) as its inscription. This inscription (*t*) must be bound to a value of type *Topic* (string) because the *Open* place has the colour set *Topic*. For this example, the arc inscription evaluated to the "*The car is safe*" (the place token colour or value).

2.3.1.2 CPNs Hierarchical Structure

One of the key features of the CPN is its ability to construct large models in a hierarchical manner [Jensen *et al.*, 2007] by using subpages (submodules, subnets or child CPN model) to build superpages (parent model, complex model). The pages interact with each other and with the superpages through a set of substitution transitions and a set of interfaces (fusion places).

A substitution transition is a transition (drawn as rectangular double lined boxes in Figure 2.3) which is located in a superpage and refined by a subpage. A fusion place is composed of one socket and one port. In practice, sockets and ports represent the

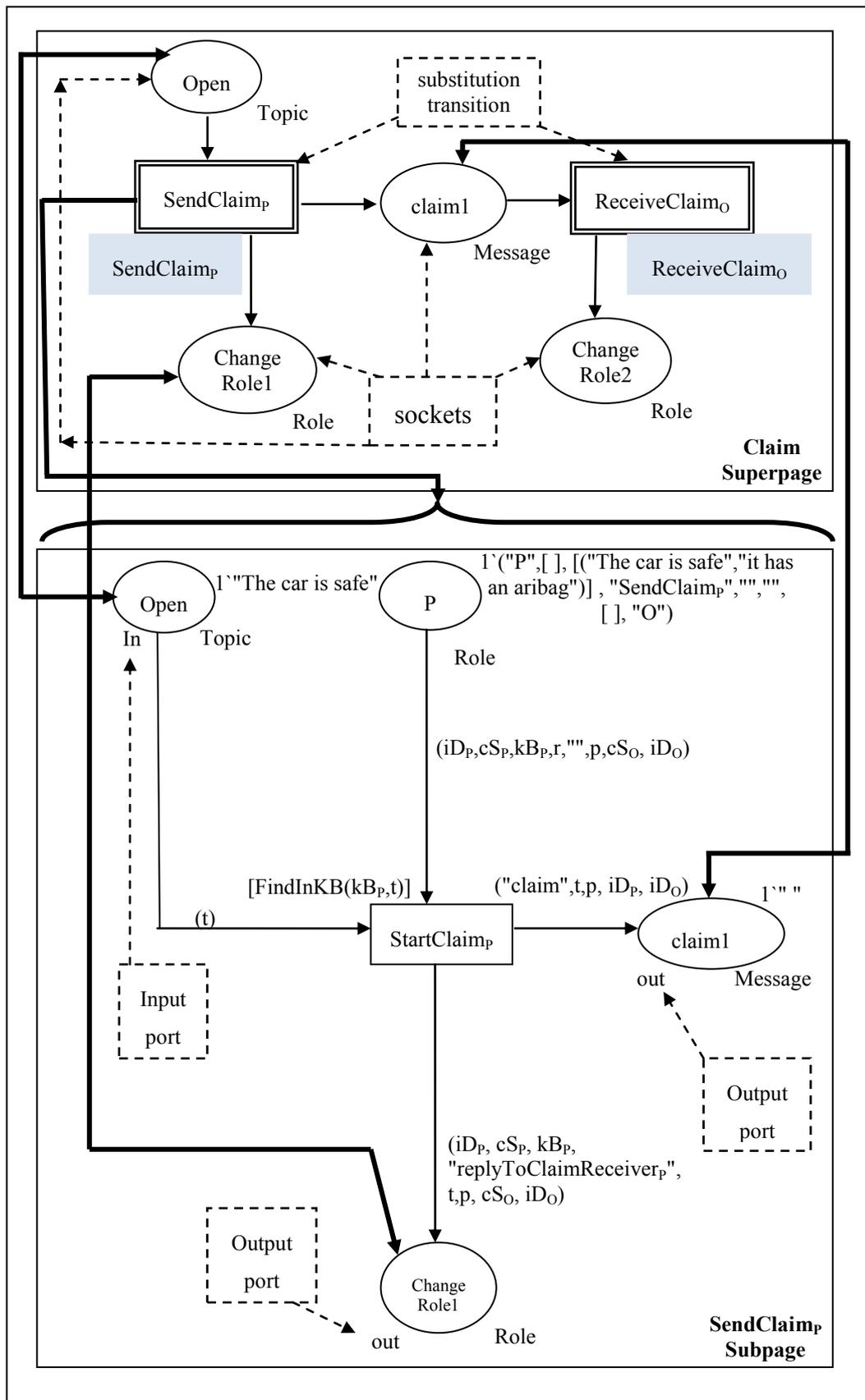


Figure 2.3: A Hierarchical CPN

same places and store the same information, but the sockets are located in the superpages whereas the ports are located in the subpages. There are three different types of sockets/ports: (1) input sockets which are assigned to input ports and which receive data from other CPNs models; (2) output sockets which are assigned to output ports and send data to other CPNs models; (3) input/output sockets which are assigned to input/output ports and receive/send data from/to other CPNs models.

Each related port and socket always has the same marking. Figure 2.3 illustrates the hierarchical specification of CPNs supported by the CPN tool. Note that in the CPN Tools (see section 2.3.1.3 for more information about the CPN Tool), below each substitution transition there is a blue rectangular subpages tag which contains the name of the subpages related to the substitution transition. In practice, the blue rectangle means that the subpage has more detailed information (information about the model behaviour) than the one represented in the superpage [Jensen *et al.*, 2007].

The claim superpage in Figure 2.3 has two *substitution transitions* (*SendClaim_P* and *ReceiveClaim_O*) and four sockets (*Open*, *claim1*, *ChangeRole1* and *ChangeRole2*). The *SendClaim_P* subpage in Figure 2.3 has an input port *Open*, two output ports *claim1* and *ChangeRole1* and an internal place *P*. The *open port* place of the *SendClaim_P* subpage is assigned to the *open* socket of claim superpage.

The *claim1 port* place of the *SendClaim_P* subpage is assigned to the *claim1* socket of claim superpage. The *ChangeRole1 port* place of the *SendClaim_P* subpage is assigned to the *ChangeRole1* socket of claim superpage. Note that in Figure 2.3, each port in the *SendClaim_P* subpage has the same name as the socket in the claim superpage to which it is assigned, but this is not essential.

2.3.1.3 CPN Tool Components

The CPN Tool¹⁰ "is a tool for editing, simulating, and analyzing Colored Petri nets." [Westergaard and Verbeek, 2002; Aalst and Stahl, 2011; Jensen *et al.*, 2007].

¹⁰ <http://cpntools.org/>

The CPN Tool supports graphical representations which makes it easy for the user to understand the structure of a CPN model and helps him/her to understand how the individual subsystems interact with each other. It also allows the user to execute the CPN model with data and analyse the model.

The CPN Tool uses the CPN SML language for declaration of variables, constants, functions, arc inscription and transition's guard condition [Jensen and Kristensen, 2009; Ullman, 1998]. It is an extension of SML (see [Jensen, 1992] chapter 6 for more information about the difference between the SML and the CPN SML language) which can be used with the state-space technique¹¹ to analyse the behaviours of communication systems [Jensen *et al.*, 2006].

The CPN tool is composed of three integrated tools which interact with a CPN model:

- (1) The CPN editor which is used to construct, edit and check the syntax of a CPN diagram;
- (2) The CPN simulator which is used to execute a CPN model;
- (3) The CPN state space tool which is used to generate the state space of a CPN model and to analyse the dynamic behaviour of a CPN model.

Figure 2.4 shows a screenshot of the CPN Tool. The area to the left is the *index* which has the Tool box with various tools that are available for the user to constitute, edit and simulate the CPN model. The remaining part of the screen is the *CPN workspace*. For more information about the CPN Tool and the construction of the CPN model see [Jensen *et al.*, 2007; Kristensen *et al.*, 1998].

¹¹ State-space technique: state-space technique [Jensen *et al.*, 2006] is used to compute all reachable states and state changes of the modeling system. See section 6.3 for more details.

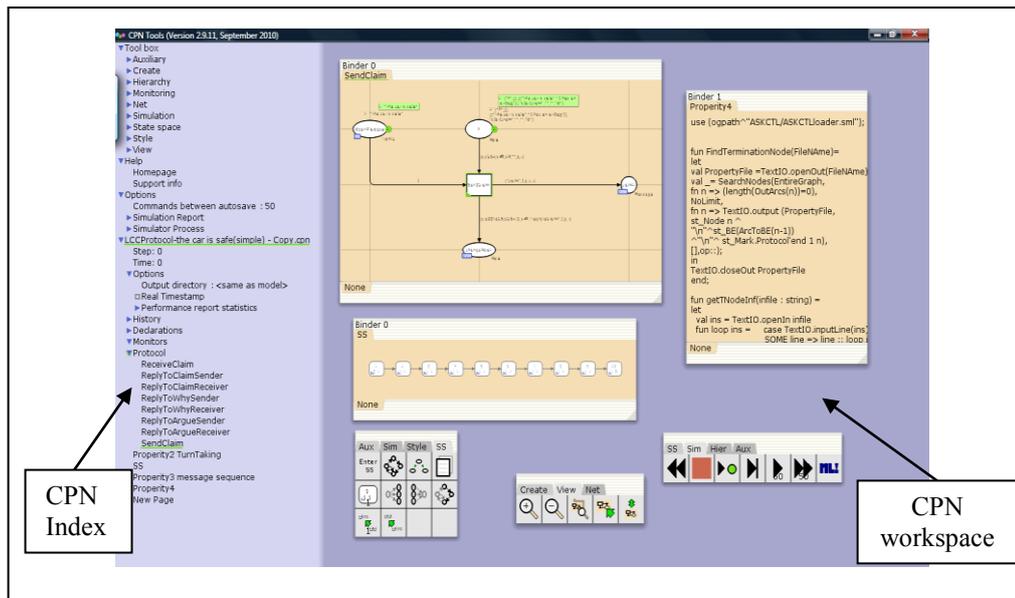


Figure 2.4: CPN Tool

CPNXML File

The CPN Tool generates for each CPN model a CPNXML file [Billington *et al.*, 2003], which is an extended markup language (XML) document [Goldfarb and Prescod, 2003] that describes the modelling elements of the CPN model. The structure of a CPNXML file is determined by the CPN Tool version [Eunice, 2005]. In this thesis, we used CPN Tools version 2.9.11.

In general, a CPNXML file is organised using pages, where each *page* represents one CPN model. In the CPNXML file, there are two types of pages [Eunice, 2005]:

- (1) Global declaration page: there is only one global declaration page in a CPN model which is used to declare colour sets and variables;
- (2) Subpage: contains the information about place, transition and arc elements of a CPN model. There is more than one subpage in a CPN model. Note that in this thesis, the number of subpages is dependent on the number of LCC roles.

Figure 2.5(a) and Figure 2.5(b) show a simple CPN diagram with one input place, one output place and one transition as well as the CPNXML description of the same CPN diagram (note that to make CPNXML file easier to read the CPNXML description in this chapter is slightly edited as compared to the CPNXML generated

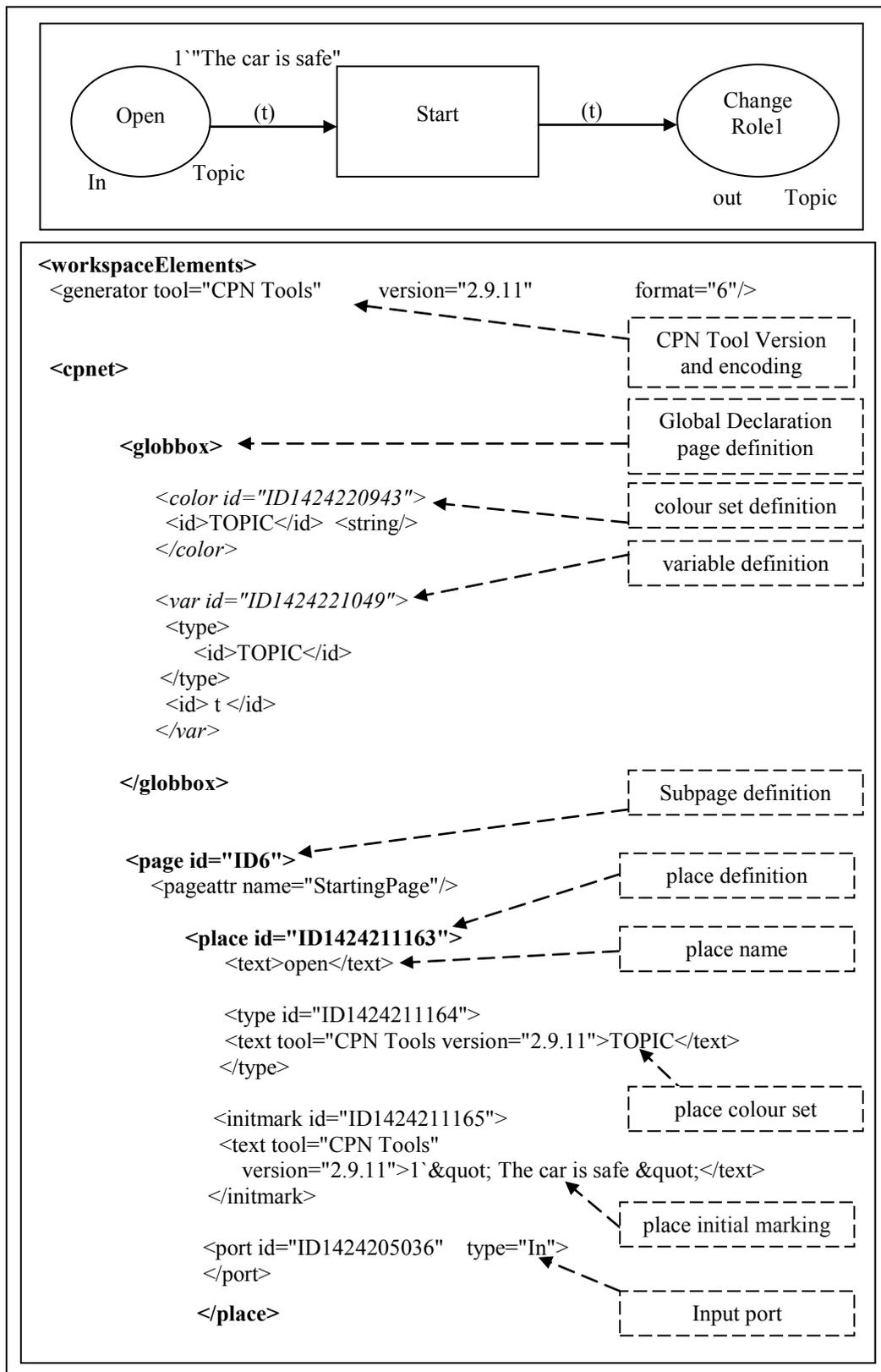


Figure 2.5 (a): CPNXML File Structure Example

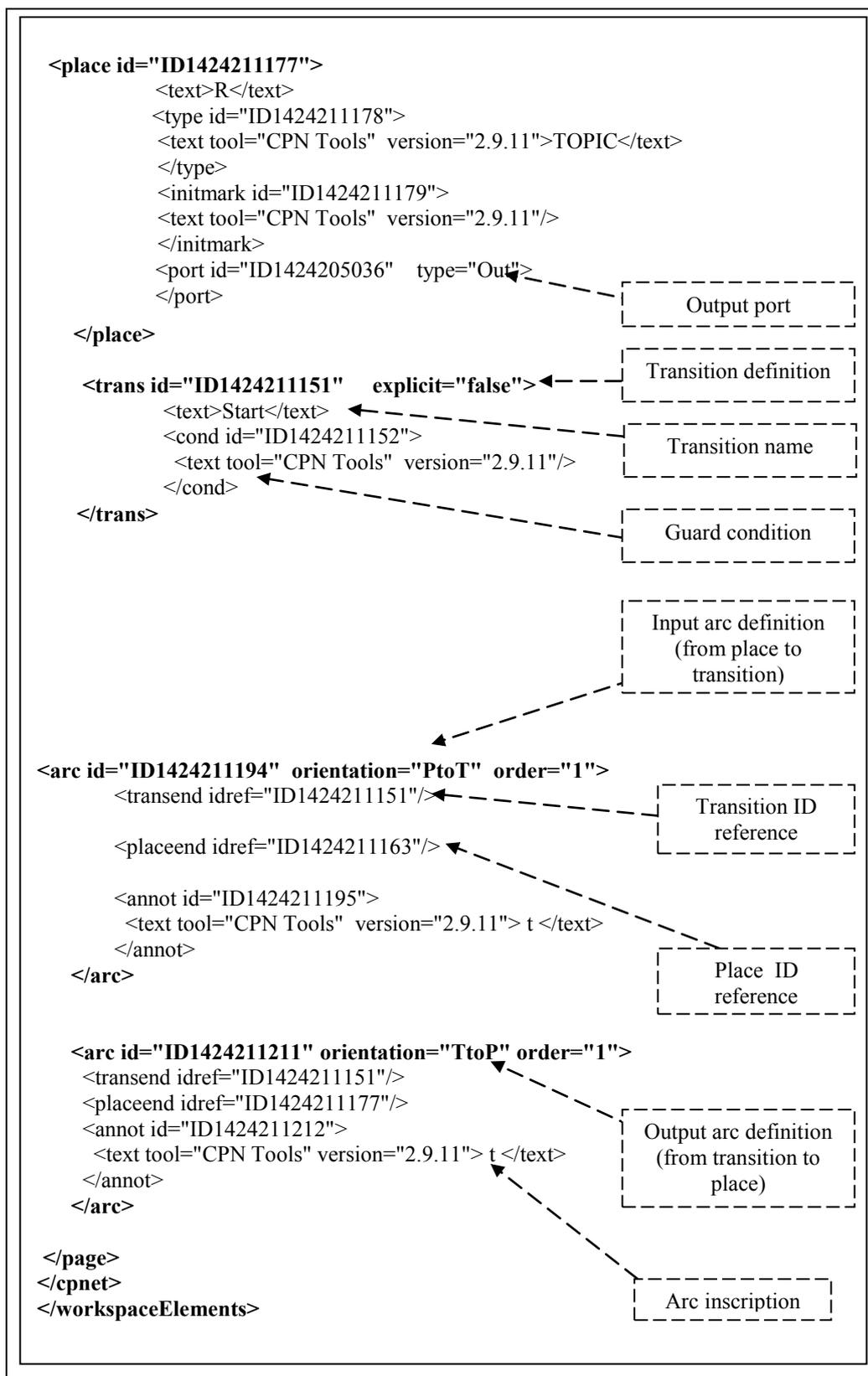


Figure 2.5 (b): CPNXML File Structure Example

- (1) Ten nodes (with rounded boxes). Each of these nodes represents a reachable marking. The marking (the token values of all places in the CPN model) of each node is described in the rectangle box next to the node.
- (2) Nine arcs. Each arc represents the occurrence of one or more binding elements that leads to the occurrence of the next node and leads us from the marking of the starting node to the marking of the termination node.

2.3.2 Comparing our Approach with Verification Approaches based on SML and CPN Model

2.3.2.1 A Transformational Approach to CPN Model

Calderon [Eunice, 2005] developed a tool to transform UML-based systems [Bauer et.al., 2001] to CPN models (Design/CPN XML¹² file) [Jensen, 1992; Jensen *et al.*, 2007; Kristensen *et al.*, 1998]. The tool was tested by running the Design/CPN tool¹³ simulator for analysing the dynamic behavior of two large-scale UML systems:

- (1) The stop and wait protocol system [Kristensen et. al., 1998]: This system has two actors: a sender and a receiver. The sender actor sends data packets to the receiver actor using a synchronous message communication protocol. Then, the system allows the sender to send another message only when this actor has received an acknowledgement message from the receiver which indicates that the receiver received the previous message;
- (2) The gas station system [Shin et. al., 2003; Shin et. al., 2005]: This is a system that allows drivers to purchase petrol (gas) and to pay the bill by credit card, debit card or Fast Pass card.

¹² <http://www.tcs.hut.fi/Software/maria/tools/cpn2maria/cpn2maria.html>

¹³ <http://www.daimi.au.dk/designCPN/>

But the CPN models generated by the tool are not ready for analysis. The user needs to perform some manual work to get an executable CPN model and to be able to verify the correctness of the generated CPN.

This work demonstrates that the development of a software tool that used to transform UML-based systems into a CPN models automatically is possible.

The most notable differences between our verification tool and Calderon's [Eunice, 2005] tool are:

- (1) Calderon's [Eunice, 2005] approach transforms data types of the UML-based system model to the colour sets types of the CPN model automatically, while our approach is not able to transform LCC parameters to the colour sets types of the CPN model automatically because LCC is an untyped language (see chapter 6 for more information).
- (2) In the Calderons' [Eunice , 2005] approach, the dynamic behaviour of the system is analysing by running the Design/CPN tool simulator, while in our approach, the dynamic behaviour of the system is analysing by using state space techniques and the CPN SML language.

2.3.2.2 A Verification Method based on SML

Paper [Suriadi *et al.*,2009] used the CPN Tool to model one case study of the Privacy Enhancing Protocols (PEPs) called the Private Information Escrow Bound to Multiple Conditions Protocol (PIEMCP)¹⁴ manually. Then, this paper used the state space techniques, CPN SML language and session-data files (these files are used by SML function to verify if some security properties are achieved) to perform:

¹⁴ Privacy enhancing protocols (PEPs): "are a family of protocols that allow secure exchange and management of sensitive user information"[Suriadi *et al.*,2009].

- (1) Model validation of the PIEMCP: to check various properties of the generated CPN model to ensure that the generated CPN model is a reliable representation of the PIEMCP protocol specification model.
- (2) Verification of the PIEMCP: this is a two stage verification.
 - a) The basic behaviour verification: to analyse the termination of session, deadlock freedom, livelock freedom and absence of unexpected dead transitions.
 - b) The Security behaviour verification: to check that the various security properties of PIEMCP model are holding and to prove the correctness of the security protocols.

The similarity between our verification approach and the approach from [Suriadi *et al.*,2009] paper is that both use the state space techniques, CPN SML language and files (the session-data file in the [Suriadi *et al.*,2009] paper and the DID properties file in our work). However, the main difference between our verification approach and the paper approach [Suriadi *et al.*,2009] are:

- (1) Suriadi's et al. [Suriadi *et al.*,2009] approach generates a CPN model from a PIEMCP system model manually, while our approach generates a hierarchical CPN model from an LCC protocol by using a set of transformational rules automatically.
- (2) Suriadi's et al. [Suriadi *et al.*,2009] approach is used to check the behaviour properties of PIEMCP system while our approach is used to check the semantics of the DID specification used against the semantics of the synthesised LCC protocol.

2.3.2.3 LCC Verification Approaches based on Model Checking

Osman's [Osman, 2007; Osman et al., 2006] approach describes a small sized and dynamic local model checker for checking the deontic model (a list of agent constraints) and trust model of MAS interactions. This model checker is a fully

automatic process, which helps agents at run-time to decide whether or not the given interaction scenarios are trustworthy to join.

This model checker is implemented in XSB tabled Prolog [Sagonas et al., 1994]. It gets as input:

- (1) LCC and deontic constraints that model MAS scenarios.
- (2) Desirable properties of the system expressed in model μ -calculus [Bradfield and Stirling, 2006].

Then, the local model checker generates the state space, one step at a time, automatically to verify whether or not MAS scenarios satisfy the desirable properties.

While Osman's approach [Osman, 2007; Osman et al., 2006] is based on process calculus model checking, our approach is based on CPN and SML language. We do not claim that our approach is better but we prefer to use a CPN-based approach over a process calculus approach because:

- (1) CPN are reasonably simpler modeling techniques in comparison with process calculus [Aalst, 2005];
- (2) CPN-based tools are easier to use since they have a graphical interface as well as a formal semantics;
- (3) CPNs modelled with the CPN tool are integrated with SML, which can be used to capture and analyse the behaviour of the CPN.

2.4 Summary

This chapter has described the background of the related topics to this thesis. It also compared the thesis with relevant related work. The background review was narrowed down to the concepts of agent protocol development language, design patterns and verification methods. The motivations of this research as well as the description of the basic concepts of argument, argumentation and dialogue games are presented in chapter 3.

Chapter 3

Argumentation, Dialogue Games and Multi-Agent Systems

Argumentation has for some time been an important area of research in natural language processing, knowledge representation, and construction of automated reasoning systems. It also has importance in Multi-Agent Systems (MAS), in particular, to the design, implementation, and analysis of models of communication between agents. In fact, argumentation-based communication not only allows agents to exchange messages but also allows agents to support their messages by giving reasons why those messages are appropriate. Commonly, argumentation-based communication is based on systems of specification that use commitment and dialogue games.

This chapter is an introduction to the basic concepts of argument, argumentation and dialogue games. It begins by defining the meaning of an argument and argumentation in Section 3.1. Section 3.2 provides a simple definition and examples of dialogue games (argumentation-based dialogue). Section 3.3 explains the advantages of using dialogue games for agent communication. The standard terminology of dialogue games is given in Section 3.4. Section 3.5 describes six basic types of dialogue. Section 3.6 stresses the importance of embedding more than one type of dialogue game within another game. Finally, Section 3.7 summarises the Argument Interchange Format work, which has been proposed to tackle the argumentation sharing problem.

3.1 Argument and Argumentation

A simple definition of argument [Besnard and Hunter, 2008] is:

"An argument is a set of assumptions (i.e., information from which conclusions can be drawn), together with a conclusion that can be obtained by one or more reasoning steps (i.e., steps of deduction). The

assumptions used are called the support (or, equivalently, the premises) of the argument, and its conclusion (singled out from many possible ones) is called the claim (or, equivalently, the consequent or the conclusion) of the argument. The support of an argument provides the reason (or, equivalently, justification) for the claim of the argument."

Argumentation [Besnard and Hunter, 2008; Eemeren et al., 1987] is the act or process of constructing arguments and counterarguments with the intention of finding conclusions for a given problem. It normally involves handling conflicts. Handling conflicts may involve comparing and evaluating arguments along with looking for pros and cons for conclusions.

In particular, according to [Maudet et al., 2007] argumentation systems can be used by:

- (1) Logicians, computer scientists and autonomous agents for forming beliefs, desires, intentions and obligations along with making decisions in the face of uncertainty and non-standard, incomplete and conflicting information. This is for the reason that argumentation offers formal systems that can be used for resolving conflicts between different arguers by constricting and comparing arguments for and against certain conclusions and finding consistent, well-supported conclusions;
- (2) Artificial intelligence (AI) and MAS designers for designing, modelling, implementing and analysing multi-agent communication. This is for the reason that argumentation offers structure and reasons for the exchange of information related to an argumentation topic.

This thesis focuses on the use of argumentation in multi-agent communication.

3.2 Dialogue Games (Argumentation-Based Dialogues)

Dialogue games (argumentation-based dialogues) are a dynamic form of argumentation which capture the intermediate stages of argument exchanges or the process of building up the set of arguments between two or more participants until the participants, as a group, reach a conclusion. Normally, dialogue games involve:

- (1) a proponent (one or more participants) which is intended as the speaker(s) of the argument,
- (2) an audience (one or more participants) which is intended as the receiver(s) of the argument.

According to Walton [Walton, 1990] dialogue games are defined as follows:

"Argument is a social and verbal means of trying to resolve, or at least to contend with, a conflict or difference that has arisen or exists between two (or more) parties. An argument necessarily involves a claim that is advanced by at least one of the parties. In an asymmetrical case, one party puts forward a claim, and the other party questions it. In a symmetrical case, each party has a claim that clashes with the other party's claim. The claim is very often an opinion, or claim that a view is right, but it need not be. In a negotiation argument, the claim could be to goods or to financial assets."

The following five cases are examples of dialogue games that we will use throughout this thesis:

- (1) Simple car safety case (adapted from [Prakken, 2006]):

***P:** My car is safe. (Making a claim)*

***O:** Why is your car safe? (Asking grounds for a claim)*

***P:** Since it has an airbag. (Arguing: offering grounds for a claim)*

***O:** OK, your car is safe. (Conceding)*

In this case, there are two parties: *P* and *O*. *P* claims that his car is safe and *O* claims that *P*'s car is not safe. At the end, *P* succeeds in persuading *O* that his car is safe by offering grounds for his claim.

- (2) Complex car safety case ([Prakken, 2006]):

***P:** My car is safe. (Making a claim)*

***O:** Why is your car safe? (Asking grounds for a claim)*

***P:** Since it has an airbag. (Arguing: offering grounds for a claim)*

***O:** Your car is not safe since the newspapers recently reported on airbags expanding without cause. (Stating a counterargument)*

*P: Newspaper reports are very unreliable sources of technological information.
(Counterattack)*

O: Still your car is not safe, since its maximum speed is very high. (Alternative counterargument)

P: OK, I was wrong about my car being safe.

In this case, there are two parties: *P* and *O*. *P* claims that his car is safe and *O* claims that *P*'s car is not safe. At first, *P* tries to persuade *O* that his car is safe by offering grounds for his claim but *O* puts forward a counterargument. Then, *P* puts forward a strong counterattack on *O*'s counterargument. After that, *O* provides his second argument as to why *P*'s car is not safe and succeeds in persuading *P* that *P*'s car is not safe

(3)The picture hanging case (adapted from [Parsons et al., 1998; Maudet et al., 2007]):

A: Can you please give me a nail? (Making a request)

B: Why do you need a nail? (Challenging)

A: Because I want to hang a picture up and to do this I need a nail. (Justifying a request)

B: But you can use a screw and a screw driver to hang the picture up! And if you ask me I can provide you with these in exchange for a hammer. (Providing an alternative plan)

A: Really, I guess in that case, I do not need the hammer. Here you go. (Accepting the request)

In this case, there are two parties: *A* and *B*. *A* wants to hang a picture up and *B* wants to hang a mirror up. *A* has a screw, screw-driver and hammer. However, to hang the picture up *A* needs a nail in addition to the hammer. In contrast, *B* has a nail and needs a hammer in addition to the nail in order to hang the mirror up.

A knows that *B* has a nail and, in order to hang the picture up, *A* needs to get the nail from *B*. *B* knows that *A* has a hammer and, in order to hang the mirror up, *B* needs to get the hammer from *A*. At first, *A* asks *B* to give him the nail but since *B* needs the nail to hang the mirror up, *B* challenges *A* by asking *A* for grounds for his request. Then, *B* provides an alternative plan for *A* that allows both *A* and *B* to achieve their goals and succeeds in persuading *A* to give away the hammer.

(4) The buyer and seller car case (adapted from [Modgil and Amgoud, 2008]):

Buyer: *I want to buy a car. (Request)*

Seller : *Why don't you try a Renault. (Making an offer)*

Buyer: *I don't want to buy a Renault. (Refusing an offer)*

Seller: *Why? (Asking the reason for rejection)*

Buyer: *Renault is French, and French cars are unsafe. (Justifying the reason for rejection)*

Seller: *Renaults are not unsafe as they have been given the award of the safest car in Europe by the EU. (Offering grounds for an offer)*

Buyer: *Okay, I accept your offer. (Accepting an offer)*

In this case, there are two parties: buyer and seller. The buyer wants to buy a car and the seller offers a *Renault*. At first, the buyer does not accept the initial offer made by the seller. Therefore, the seller offers grounds for his offer and the buyer accepts it, and the dialogue ends.

(5) The flying abilities of birds and penguins case:

A1: *Tweety flies. (Making a claim)*

A2: *Why does Tweety fly? (Asking for grounds for a claim)*

A1: Tweety is a bird , birds generally fly. (Arguing: offering grounds for a claim)

A2: Tweety does not fly because Tweety is a penguin, penguins do not fly. (Starting a counterargument)

A1: You are right. Tweety does not fly. (Conceding an argument).

In this case, there are two parties: *A1* and *A2* reasoning about whether a particular penguin Tweety can fly. *A1* claims that Tweety can fly and *A2* claims that Tweety cannot fly. *A1* tries to persuade *A2* that Tweety can fly by offering grounds for his claim but *A2* puts forward a counterargument which persuades *A1* that Tweety cannot fly.

3.3 Argumentation for Agent Communication

An agent, according to Jennings et al. [Jennings et al.,1998] "is a computer system, situated in some environment, that is capable of flexible autonomous action in order to meet its design objectives".

Despite the fact that the agent is autonomous, in a MAS, each individual agent needs to consider its dependence on other agent(s), their role(s) in their environment, their commitments to other agent(s), and environment rules which control their behaviour. Agents need to communicate, cooperate, coordinate and negotiate with each other in order to achieve their individual or cooperative goals, resolve and manage conflicts or disagreements and differences of opinions, work together to resolve problem or to prove that specific information is either true or false, and inform each other of important facts. For example, for an agent to perform a new activity or to cancel or modify an existing activity, it needs to persuade other agents to act in the way required. To succeed in this, agents must be able to speak the same language with each other and must be able to construct a sequence of arguments for and against a particular claim and exchange these arguments with other agents [Norman et al.,2004].

This is exactly the type of communication which correlates to the interests of argumentation-based dialogue theory. In fact, communication with argumentation allows an agent to request a change to the arguments, to justify their attitude, and to provide reasons for their claims [Maudet et al., 2007]. As a result of this fact, there has been an increased interest in argumentation-based dialogue (dialogue games) as an alternative model of agent communication - for example, by Sycara [1989]; Reed [1998]; and Parsons et al. [2003].

3.4 Dialogues Games Terminology

We can view dialogue as a game which involves interactions between two or more participants. Each participant is considered as a player who tries to achieve its main goal (group goals) by making some finite set of moves. As in any game, players must speak a common communication language and abide by combination rules (e.g. rules which stipulate when a player(s) is allowed to make particular moves at a specific time in the game) [Parsons and McBurney, 2003; Maudet et al., 2007; Walton and Krabbe, 1995; Norman et al., 2004].

The standard terminology considered for the specification of protocols in dialogue games includes [Hamblin, 1970; Walton and Krabbe, 1995; Prakken, 2000; Mcburney et. al., 2003; Prakken, 2006]:

- (1) Locutions rules: represent the set of permitted moves;
- (2) One Commitment Store (CS) for each participant: the CSs of the participants reflect the state of the dialogue;
- (3) Commitment rules (effective rules): define the propositional commitments made by each participant with each move during the dialogue;
- (4) Pre-condition: rules define the conditions under which the move will be achieved;

- (5) Structural rules (reply rules or dialogue rules): define legal moves in terms of the available moves that a participant can select to follow on from the previous move;
- (6) Turn Taking (next player): specifies the next player [Prakken, 2006];
- (7) Starting rules (commencement rules) [Mcburney et. al., 2003]: define the conditions beginning the dialogue;
- (8) Termination rules [Mcburney et. al., 2003; Prakken, 2006]: define the conditions ending the dialogue.

Dialogues Games Example

There are many examples [Parkken, 2000; Parkken, 2005; McBurney and Parsons, 2002; Walton and Krabbe, 1995] from literature for a formal model of dialogue games. These examples include an abstract form (model) of dialogue games between two agents. The primary difference between these examples is the set of locutions.

One of these examples is a persuasion dialogue (adapted from [Parkken, 2000; Parkken, 2005]), where a dialogue is presented as a game in which one participant (proponent 'P') attempts to persuade another participant (opponent 'O') to change their point of view about a particular topic 'T'. We will describe this dialogue by using the standard terminology of dialogue games introduced above:

(1) ***Locutions:***

Locutions (speech acts)	Meaning of Locution
claim(T)	Making a claim
why(T)	Asking grounds for a claim
concede(T)	Conceding (accepting) a claim
argue(Pre, T)	Offering grounds for a claim
retract (T)	Retracting (withdrawing) a claim

(2) ***Commitment Store:*** There is one *CS* for each participant: { CS_P , CS_O }

(3) **Commitment rules:**

Locutions	Commitment rules	Meaning of Commitment rules
claim(T)	CS \cup {T}	The effect of a 'claim' move is always to add topic 'T' to the mover's commitments 'CS'
why(T)	CS	The mover's commitments remain unchanged
concede (T)	CS \cup {T}	The effect of a 'concede' move is always to add topic 'T' to the mover's commitments 'CS'
argue(Pre, T)	CS \cup {T} \cup {Pre}	The effect of an 'argue' move is always to add topic 'T' and premise 'Pre' to the mover's commitments 'CS'
retract (T)	CS - {T}	The effect of a 'retract' move is always to remove topic 'T' from the mover's commitments 'CS'

(4) **Pre-conditions**

Locutions	Pre-conditions
claim(T)	There are no special pre-conditions to starting a persuasion dialogue (for the utterance of 'claim' locution).
why(T)	In order for the speaker to ask grounds for a claim 'T', he must not be able to find 'T' in his 'KB' or 'CS' (he must not have committed to it).
concede(T)	In order for the speaker to concede a claim 'T', he must not have committed to it. He also must not have committed to the opposite of the claim ' \sim T'.
argue(Pre, T)	In order for the speaker to offer grounds for a claim 'T', he must not have committed to the claim 'T'. He also must be able to find promise 'Pre' in his 'KB' or 'CS' to support a claim 'T'.
retract(T)	In order for the speaker to retract a claim, he must have committed to it. He also must not be able to find a promise 'Pre' to support a claim 'T'.

(5) **Structural rules:**

Locutions	Structural rules	Meaning of Structural rules
claim(T)	why(T) or concede(T)}	After a 'claim' move, the Next player can select either 'why' or 'concede' locutions
why(T)	argue(Pre) or retract(T)	After an 'why' move, the Next player can select either 'argue' or 'retract' locutions
concede (T)	No reply	After a 'concede' move, the Next player cannot make a move.
argue(Pre, T)	why(Pre), argue(Def) or concede(T)	After an 'argue' move, the Next player can select 'why', 'argue' or 'concede' locutions
retract (T)	No reply	After a 'retract' move, the Next player cannot make a move

- (6) **Turn Taking:** The turn-taking between participants switches after each move.
- (7) **Starting rules:** dialogue is allowed to begin with claim locution.
- (8) **Termination rules:** dialogue is allowed to end when agents send either concede or retract locutions.

3.5 Types of Dialogues

Walton and Krabbe [Walton and Krabbe, 1995] identify six different general types of dialogue in AI and MAS: persuasion, inquiry, information-seeking, negotiation, deliberation and eristic. These dialogue types are classified based on:

- (1) Their pre-conditions of the dialogue;
- (2) Their Participant's goals for the dialogue;
- (3) The primary goal of the dialogue.

The definitions and properties [Walton and Krabbe, 1995] of these dialogue types are summarized in Table 3.1.

Persuasion [Parkken, 2000; Parkken, 2005] dialogue arises from an initial clash or conflict of opinion. Its primary goal is to resolve the initial clash or conflict. It usually takes the form of a sequence of questions (from the opponent) and the replies (from the proponent) or attacks (from the opponent) and defence of its position (from the proponent) [Walton and Krabbe, 1995]. An example of a persuasion dialogue is illustrated in Figure 3.1.

Inquiry [Black and Hunter, 2007; Black and Hunter, 2009] dialogue is similar to the persuasion dialogue since it aims at a stable agreement. However, it differs from a persuasion dialogue since it does not arise from a conflict but from a problem (something that is not proved to be true or false). To successfully end an inquiry dialogue, each participant must reach the same conclusion [Walton and Krabbe, 1995].

Bridging the Specification Protocol Gap in Argumentation

Type of Dialogue	General Definition	Pre-conditions	Participant's Goal	Primary Goal of the Dialogue
Persuasion	One participant (proponent) attempts to persuade another participant (opponent) to change their point of view about a particular topic. [Parkken, 2000; Parkken, 2005].	Clash or Conflict of opinions	Persuade other participant	Resolve the initial conflict, reach a stale agreement or clarity issue
Inquiry	"The participants collaborate to answer some question or questions whose answers are not known to any one participant" [Parsons et al., 2003]	Need to prove hypothesis to answer some questions	Find and verify evidence	Prove or disprove hypothesis
Negotiation	"The participants bargain over the division of some scarce resource in a way acceptable to all, with each individual party aiming to maximize his or her share"[Parsons et al., 2003]. The goal of the dialogue may be in conflict with the individual goals of each of the participants[Parsons et al., 2003]	Conflict of interests	Get what you most want	Find areasonable settlement or an attractive deal to all participant
Information seeking	One participant is seeking some information from another participant, who is believed by the first participant to know this information. [Parsons et al., 2003]	One participant lacks and needs information and other participant has this information	Obtain or give information	Exchange information
Deliberation	"Participants collaborate to decide what course of action to take in some situation. Participants share a responsibility to decide the course of action, and either share a common set of intentions or a willingness to discuss rationally whether they have shared intentions"[Parsons et al., 2003]	Practical problem that needs for action (decision to act)	Co-ordinate goals or actions	Decide best course of action
Eristic	"Participants quarrel verbally as a substitute for physical fighting, with each aiming to win the exchange" [Parsons et al., 2003]	Personal conflict	Participants are trying to win and verbally hit out opponents	Reveal deeper basis of conflict

Table 3.1: Dialogue Types

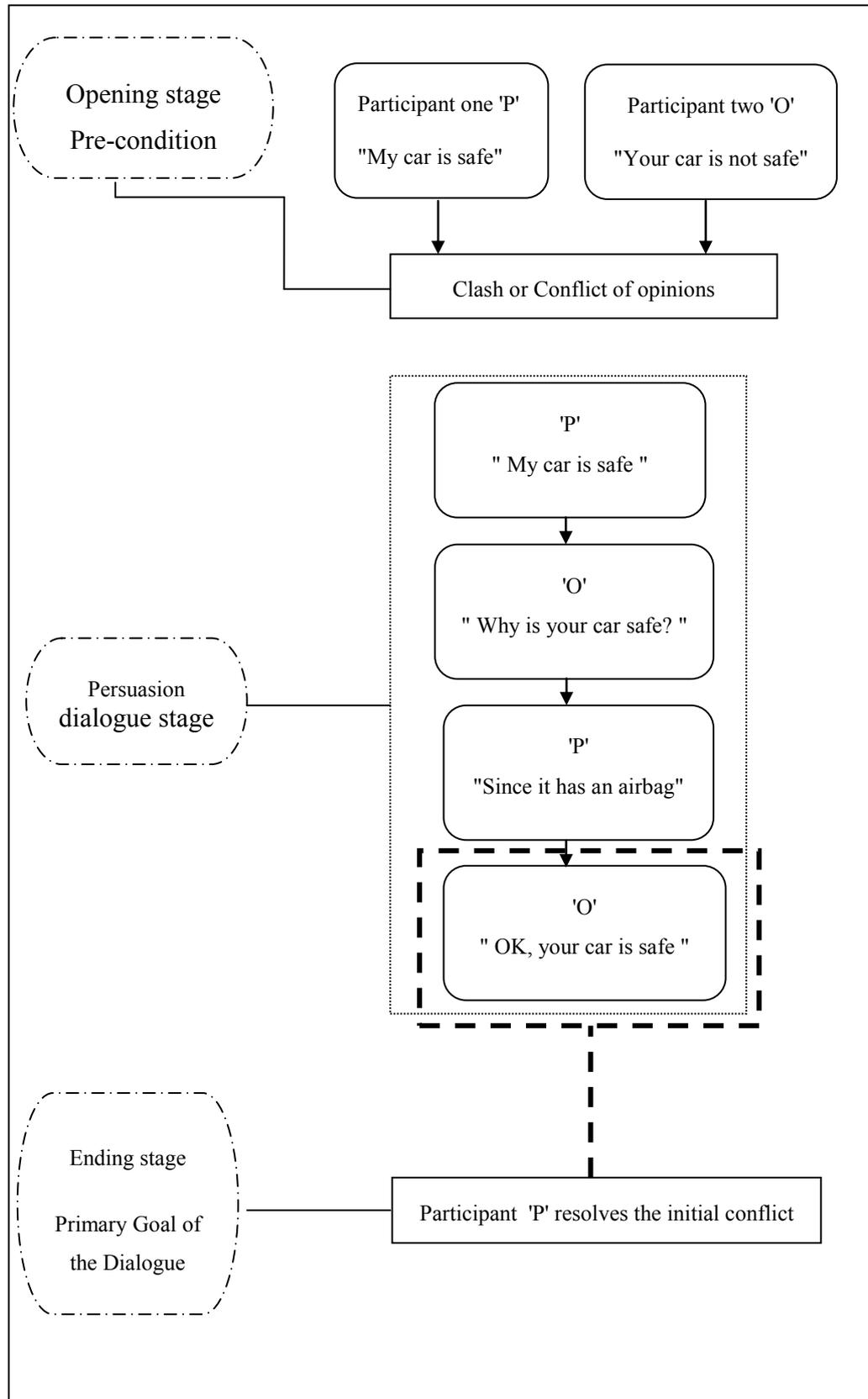


Figure 3.1: Persuasion Dialogue Example (Car Safety Case)

Negotiation [Parsons et. al., 1998; Sadri et. al., 2001; Luo et. al., 2001] dialogue is similar to the persuasion dialogue since it arises from a conflict. However, it differs from a persuasion dialogue since its goal is to make a deal that is attractive to all participants [Walton and Krabbe, 1995].

Information seeking [Doutre et. al.,2005; Walton, 1998] dialogue differs from the negotiation and persuasion dialogues since it does not arise from a conflict but arises from a situation where one participant lacks information and the other participant has this information. It also differs from an inquiry and a deliberation dialogue since these two arise from a lack of information, whereas, in an information-seeking dialogue, the information is already present and the problem is to find a way to obtain this information from the other participant (who obtains this information) [Walton and Krabbe, 1995].

Deliberation [Tang and Parsons, 2006; McBurney et.al., 2007] dialogue is similar to an inquiry dialogue but differs from a persuasion dialogue since it does not arise from a conflict but from an open problem. However, it differs from the inquiry dialogue since it has to proceed with some action. In practice, deliberation dialogue is considered as a practical type of dialogue since its goal is to perform an action (decides how to act to solve a practical problem) which enables the practical interaction of life and human business to go ahead [Walton and Krabbe, 1995].

Eristic [Walton, 1998] dialogue is similar to the persuasion and negotiation dialogues since it arises from conflict. However, in this dialogue each participant is trying to win and their main goal is to hit out at other participants (opponents). In this thesis, we will not consider the eristic type of dialogue since it is not expected to be useful in agent interactions. Rather, it involves venting grievances or serving primarily as a dialogue substitute for physical confrontation [Walton and Krabbe, 1995, page 76] .

Figure 3.2 (adapted from [Walton and Krabbe, 1995]) summarises the differences between these types of dialogue.

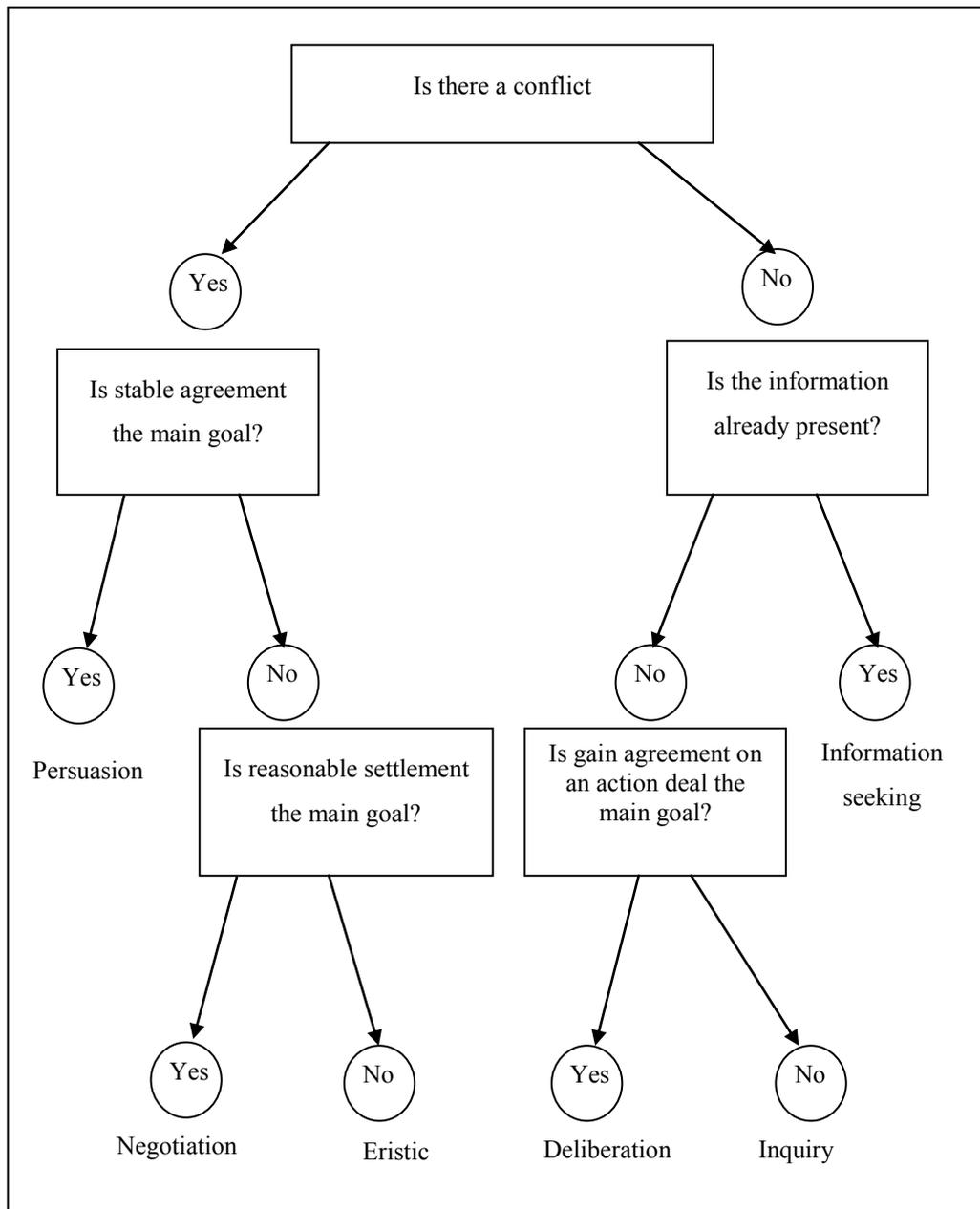


Figure 3.2: Determining the Type of Dialogue

3.6 Embedded Dialogues

Typically, agent interaction involves several dialogue types. Walton and Krabbe [Walton and Krabbe, 1995] stress the importance of embedding more than one type of dialogue game within another game, which allows complex interaction to occur (e.g. [Black and Anthony, 2007; Sadri et.al., 2001; Reed, 1998; McBurney and Parsons, 2002; Dimopoulos et.al., 2005]). There are two types of embedded dialogues:

3.6.1 First Type: Shift from One Type to Another Type

Embedded dialogues are different dialogues types, which occur during a specific type of dialogue between agents causing the dialogue to shift to another type. Some examples of different situations in which we may find embedded dialogue are:

- (1) One of the participants in an inquiry dialogue reaches a conclusion before the other participants, then it needs to persuade her fellow participants to reach the same conclusion since, to successfully end an inquiry dialogue, each participant must reach the same conclusion. Therefore, persuasion dialogue could be embedded as sub-dialogue in any given inquiry dialogue.
- (2) A persuasion dialogue may reach a point where the participants need to settle a fact before the discussion can continue, which means that the participants need to move to an inquiry dialogue to settle the fact. Therefore, inquiry dialogue could be embedded as sub-dialogue in any given persuasion dialogue.
- (3) A negotiation dialogue may well move through persuasion or information seeking dialogue in order to reach a decision.

3.6.2 Second Type: Internal Embedded

Embedding one type of the dialogue to the same type of the dialogue (change in the subject of dialogue) called internal embedded (shifts) [Walton and Krabbe, 1995]. One example of the internal embedded is that an inquiry dialogue may reach a point where the participants need to settle a sub-fact before settling the main fact [Black and Anthony, 2007]. Therefore, inquiry dialogues could be embedded as subdialogues in another inquiry dialogue.

3.7 Argumentation Sharing Problem and Argument Interchange Format

Today, argumentation [Maudet et al., 2007; Rahwan, 2006] is gaining more prominence since it is being used as part of the high-level specification of MAS. However, a wide ranging approach of this kind carries with it various challenges

such as the lack of shared and agreed notations for an interchange format concerning arguments and argumentation. To tackle this challenge, the argumentation community has developed the Argument Interchange Format (AIF) [Chesnevar et al.,2007;Willmott et al., 2006], which provides a common language to exchange argumentation concepts among agents in a MAS.

3.7.1 AIF Definition

AIF [Chesnevar et al.,2007; Willmott et al., 2006] is the result of an international effort which proposed a format for representation and communication of argument resources between agents, research groups, argumentation tools, and specific domains. It provides an ontology that can easily be extended to deal with different types of argumentation formalisms and schemes. It is used to represent argument entities and the relations between these entities.

3.7.2 AIF Elements

The AIF [Chesnevar et al.,2007; Willmott et al., 2006] provides an ontology which represents an argument as a network of linked nodes. This network consists of two types of nodes: *Information nodes (I-nodes)* that contain specific data (such as claims, proposition and premises) depending on the domain of discourse, and *Scheme Application nodes (S-nodes)* that describe the domain independent patterns of reasoning. *S-nodes* come in three different types; include the *Rule of Inference Application nodes (RA-nodes)* that define the support or inference of argument, *Preference Application nodes (PA-nodes)* that represent the value judgements or preference orderings of argument, and *Conflict Application nodes (CA-nodes)* that specify the conflict of argument.

There are various restrictions on how nodes are connected. For example, I-nodes cannot be connected to other I-nodes directly; they must be connected across S-nodes. On the other hand, S-nodes can be connected to other S-nodes directly. Basically, two types of edges can be added to connect any two nodes: scheme edges that support conclusions that start from S-nodes and end either in I-nodes or S-nodes,

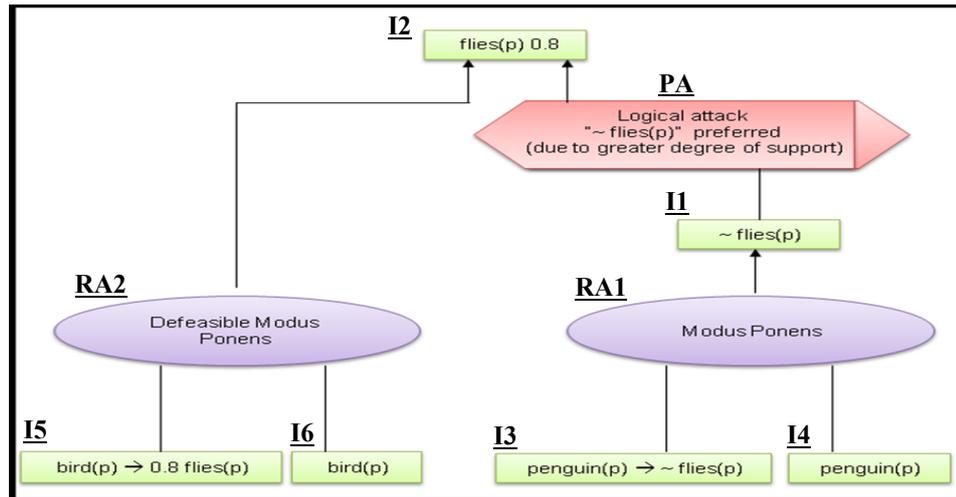


Figure 3.3: Specification in AIF of the Arguments Exchanged by Agents Discussing the Flying Abilities of the "P" Bird

and data edges that supply data and start from I-nodes and end in S-nodes. See [Chesnevar et al.,2007] for more details.

3.7.3 AIF Example

An example of AIF is shown in Figure 3.3 [Willmott et al., 2006; Modgil and McGinnis, 2007]. This concerns a multi-agent persuasion dialogue where N ($N \geq 2$ and unbounded) agents are involved in a discussion about the flying abilities of a bird called "P" (Note that *I-nodes* are shown as rectangles, *RA-nodes* as ellipses and *PA-nodes* as hexagon):

- (1) There are two arguments: one for $\sim flies(P)$ (*I1-node*) and one for $flies(P)$ (*I2-node*);
- (2) The argument for $\sim flies(P)$ is composed of one Rule of Inference Application node (*RA1-node* that defines the support or inference of argument), namely Modus Ponens and two child nodes (premises);
- (3) The argument for $flies(P)$ is composed of one *RA2-node*, namely defeasible Modus Ponens and two child nodes (premises);

- (4) AIF assumes that there is a way of ordering the support for premises. In this particular example, the choice was the justification through the probability. The argument for \sim *flies*(*P*) has a higher degree of support because the premises (*I3-node* and *I4-node*) support it with a higher degree of probability (1 degree). Conversely, the argument for *flies*(*P*) is weak because the premises (*I5-node* and *I6-node*) support it with only 0.8 degree (a low probability). So, \sim *flies*(*P*) is preferred to the argument for *flies*(*P*). That is why the intermediate Preference Application node (*PA-node* that defines the value judgments or preference orderings of argument), namely Logical attack, links \sim *flies*(*P*) (*I1-node*) to *flies*(*P*)(*I2-node*).

This example demonstrates that a persuasion dialogue can be specified abstractly by using arguments expressed in AIF. It describes the argument entities and relations between argument entities but it does not describe the items related to the interchange of arguments between agents (e.g. locutions and pre- and post-conditions for each argument). It also does not directly influence the specification of agent communication languages and interaction protocol standards.

3.7.4 AIF Implementation Problem

AIF enables users to structure arguments using diagrammatic linkage of natural language sentences. However, AIF does not model dialogue games (because it does not show the interchange of arguments between agents). Besides, it is not an executable specification language. It specifies the properties that define an argument without prescribing how that argument may be made operational. In fact, AIF is used to represent data (argumentation structure) not to process data (it does not represent or generate a dialogue games protocol). In other words, it lacks the ability to implement complex systems of arguments from high-level specifications.

Papers by [Chesnevar et al.,2007; Willmott et al., 2006] suggest a way to solve AIF problem by identifying two elements: (1) Locutions, which are particular words, phrases or forms of expressions which are used by agents, (2) Interaction Protocols, which define communication between agents via a set of rules governing how two or

more agents should interact in order to reach a specific goal. These papers also give the advantages of defining the interaction protocol language as part of AIF: (1) If we can find an interaction protocol language that can be used practically for computation then it will be easier to develop an associated computer program which is durable; (2) To support formal analysis and verification, we need to use a declarative language; (3) To facilitate human readability, we need to use a high-level language. These papers also suggest the use of patterns in the design of protocols. Therefore, these papers only provide some suggestions for solving the AIF deployment problem and demonstrate that it is difficult to solve it.

3.7.5 AIF Extension

AIF Extension by Modgil and McGinnis

Modgil and McGinnis' [Modgil and McGinnis, 2007] tried to solve the AIF dialogue problem by extending the AIF to represent characterised argumentation-based dialogues. The extensions are based on two types of nodes: Information nodes (I-nodes) whose content expands to represent locution, and Protocol Interaction Application nodes (PIA) that are created to represent interaction protocols and used to link I-nodes.

An example of this work is illustrated in Figure 3.4 (see persuasion dialogue game example in section 3.4):

- (1) *A1* opens the discussion by sending *claim(Tweety flies)* in *I1-node*.
- (2) *PIA1-node* specifies that *A2* can reply with *why(T)* or *concede(T)*.
- (3) *A2* sends *why(Why does Tweety fly?)* in *I2-node*.
- (4) *PIA2-node* specifies the legal replies *argue(Pre)* where *Pre*'s conclusion is *T*, or *retract(T)*.

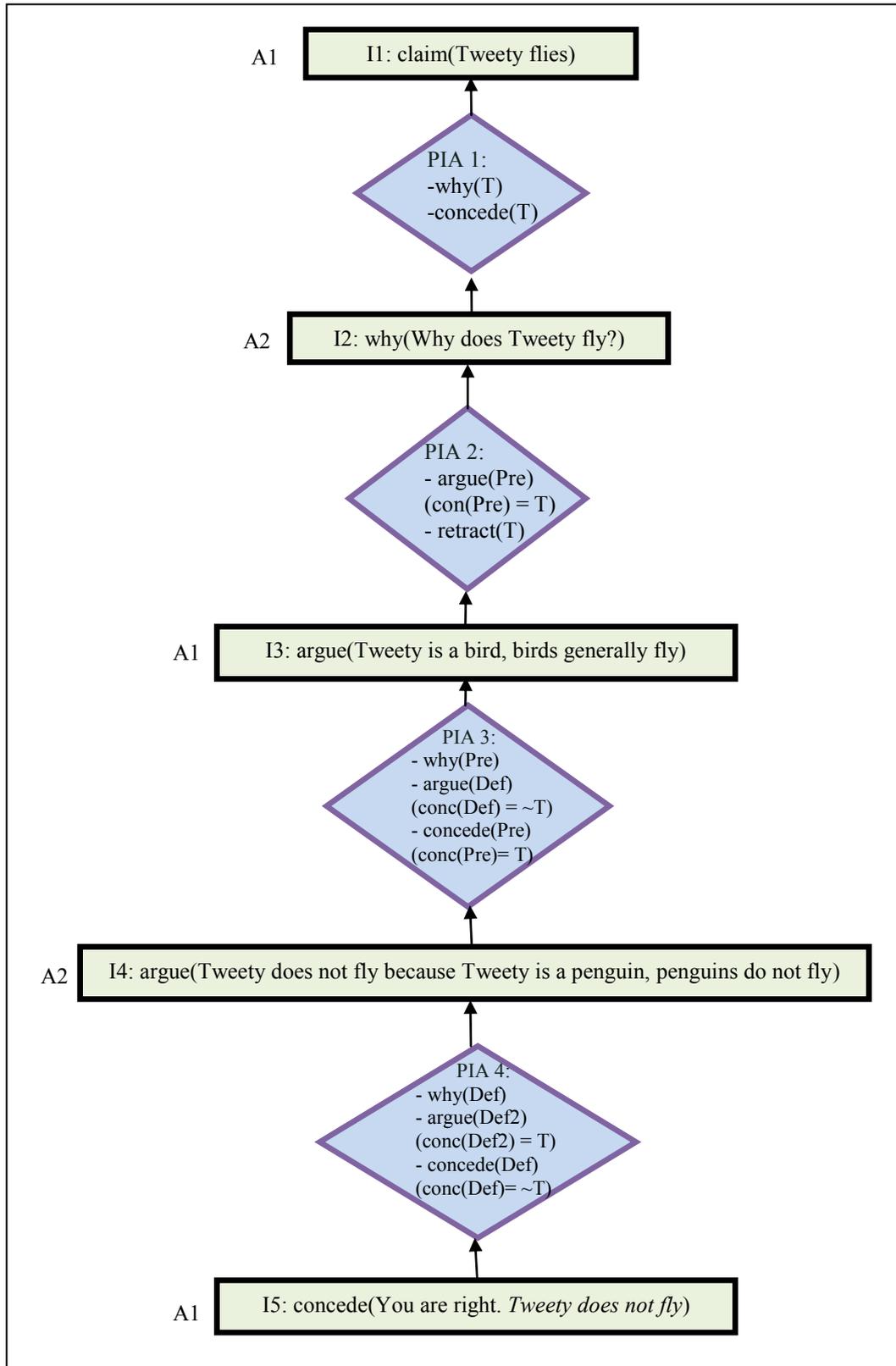


Figure 3.4: A Dialogue Graph Represented in the AIF

- (5) *A1* responds to the challenge by declaring the supporting premises "Tweety is a bird, birds generally fly" for "Tweety flies" [sends *argue(Tweety is a bird, birds generally fly)* in *I3-node*].
- (6) *PIA3-node* specifies the legal replies *why(Pre)*, *argue(Def)* where *Def*'s conclusion is $\sim T$, or *concede(Pre)* where *Pre*'s conclusion is *T*.
- (7) *A2* puts forward a strong counterargument "Tweety does not fly because Tweety is a penguin, penguins do not fly" [sends *argue("Tweety does not fly because Tweety is a penguin, penguins do not fly")* in *I4-node*].
- (8) *PIA4-node* specifies the legal replies *why(Def)*, *argue(Def2)* where *Def2*'s conclusion is *T*, or *concede(Def)* where *Def*'s conclusion is $\sim T$.
- (9) *A1* concedes to the *A2*'s argument that "Tweety does not fly" [sends *concede("You are right. Tweety does not fly")* in *I5-node*].

This study also represents agents interaction protocols by using a Lightweight Coordination Calculus language (LCC) [Robertson, 2004; Hassan et. al., 2005] (see chapter 5 for more details). To explain the use of LCC it uses as an example of an argumentation-based medical dialogue where arguments are specified and evaluated in the ASPIC¹⁵ (Argumentation Services Platform with Integrated Components) engine. The result of this study supports the idea that protocol rules could be represented as a part of the dialogue. However, this study was limited in three important ways. Firstly, it only shows how to implement a particular sort of argumentation in LCC. Secondly, it is limited to dialogues between only two agents. Finally, it does not explain how to synthesise protocols (semi-)automatically for any given argumentation.

¹⁵ ASPIC [Fox et.al, 2006] provides a general formal model for argumentation functions for individual agents and argumentation between agents in medical multi-agent systems. It enables agents to resolve conflicts of opinion in order to diagnose medical cases and find treatments.

AIF Extenuation by Reed et al.

Reed et al. [Reed et al., 2008] extended AIF to AIF⁺¹⁶ so that it could handle argumentation dialogue games as well as represent the relation between the locution and its propositional content. The extensions are based on three nodes:

- (1) Locution nodes (*L-nodes*) a subclass of *I-nodes* which are created to represent dialogue history (utterances of locutions);
- (2) Transition Application nodes (*TA-nodes*) a subclass of *RA-nodes* which are used to link two L-nodes and capture the flow of a dialogue (the sequence of connected locutions)
- (3) Illocutionary Application (*YA-nodes*). To handle natural arguments (to represent the relation between the locution and its propositional content), [Reed et al., 2010] extend AIF⁺ to represent the interaction between locutions uttered as part of an argumentation-based dialogue (AIF⁺ nodes) and the argument structures (AIF nodes) by creating a new node type called Illocutionary Application (*YA-nodes*). *YA-nodes* links *I-nodes* with *L-nodes*, and *RA-nodes* with *TA-nodes*.

An example of this work is illustrated in Figure 3.5 (Some detail is omitted from Figure 3.5 for clarity. Please see chapter 8, section 8.1.2 for more information):

- (1) In this dialogue between *A1* and *A2*, the dialogue game consists of seven *L-nodes* which are represented by *L1*, *L2*, *L3*, *L4*, *L5*, *L6* and *L7* nodes.
- (2) The argument consists of six propositions which are represented by *I1*, *I2*, *I3*, *I4*, *I5* and *I6* nodes.
- (3) The *L1*, *L3*, *L4*, *L5*, *L6* and *L7* have illocutionary nodes connecting them with propositional contents *I2*, *I5*, *I3*, *I4* and *I1*, respectively.

¹⁶ AIF+ [Reed et al., 2008 ; Reed et al., 2010]: the development of AIF+ still ongoing. See http://www.arg.dundee.ac.uk/?page_id=197

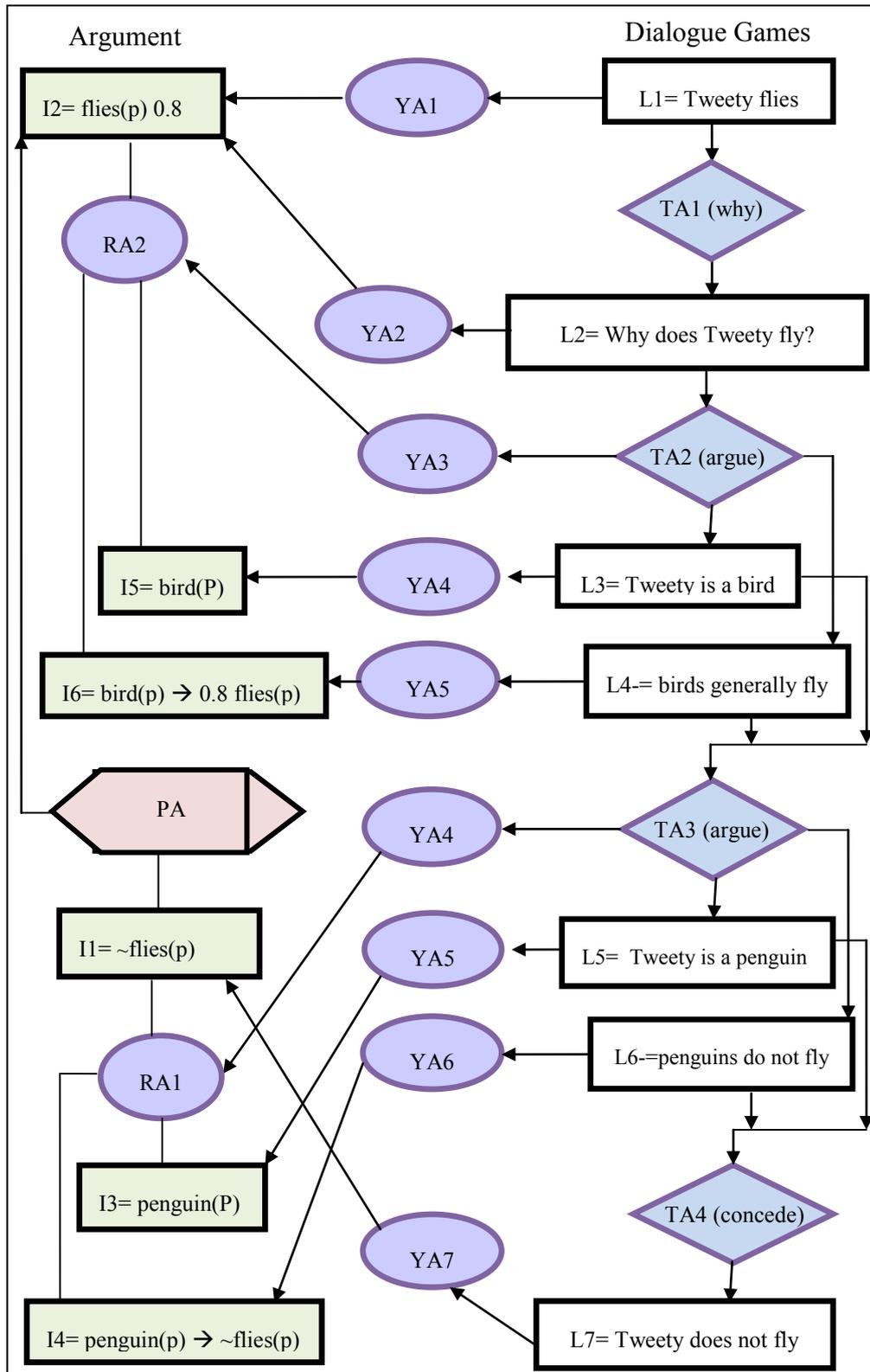


Figure 3.5 : Illustration of the Link between Argument (AIF Nodes) and Dialogue Games (AIF⁺ Nodes)

- (4) Locution nodes $L1$ and $L2$ have a transition node $TA1$ connecting them.
- (5) Locution nodes $L2$, $L3$ and $L4$ have a transition node $TA2$ connecting them.
- (6) Locution nodes $L2$, $L3$, $L4$ and $L5$ have a transition node $TA3$ connecting them.
- (7) Locution nodes $L5$, $L6$ and $L7$ have a transition node $TA4$ connecting them.
- (8) The interaction between the argument and the dialogue game is described by means of the YA-nodes:
 - The links between $L1$, $L3$ and $L4$ with $I2$, $I5$, $I6$ are represented by $YA1$, $YA4$ and $YA5$, respectively.
 - The illocutionary node $YA2$ links $L2$ and its propositional content $I2$.
 - The illocutionary node $YA3$ links $TA2$ and $RA2$.
 - The links between $L5$, $L6$ and $L7$ with $I1$, $I3$, $I4$ are represented by $YA5$, $YA6$ and $YA7$, respectively.
 - The illocutionary node $YA7$ links $L7$ and its propositional content $I1$.
 - The illocutionary node $YA4$ links $TA3$ and $RA1$.

In this example:

- (1) $A1$ opens the discussion by sending *claim(Tweety flies)* in $L1$ -node.
- (2) $A2$ sends *why(Why does Tweety fly?)* in $L2$ -node.
- (3) $A1$ responds to the challenge by sending *argue(Tweety is a bird, birds generally fly)* in $L3$ -node and $L4$ -nodes.
- (4) $A2$ puts forward a strong counterargument by sending *argue("Tweety does not fly because Tweety is a penguin, penguins do not fly")* in $L5$ -node and $L6$ -nodes.
- (5) $A1$ concedes to the $A2$'s argument by sending *concede("You are right. Tweety does not fly")* in $L7$ -node.

Like Modgil and McGinnis' [Modgil and McGinnis, 2007], the results of AIF⁺ support the idea that protocol rules could be represented as a part of the dialogue. However, similarly to AIF, AIF⁺ is used to represent data (describe the dialogue games' structure), not to process data (it does not generate dialogue games). It also does not explain how to synthesise protocols (semi-)automatically for any given argumentation.

In conclusion, both Modgil and McGinnis' [Modgil and McGinnis, 2007] and Reed et al. [Reed et al., 2010; Reed et al., 2008] attempted to solve the dialogue problem of AIF, but they did not try to solve the implementation problem.

In chapters 4 and 5 we will propose a new method to solve AIF dialogical and implementation problems. We will accomplish this by extending the AIF. Our extension will consist on adding more information to the AIF to represent interaction protocol information, as well as some implementation information, to allow the user to synthesise the multi-agent interaction protocol from it.

3.8 Summary

This chapter has presented some concepts of arguments and argumentation, summarising the advantages of using argumentation for agent communication, as well as the problems of argumentation.

In practice, the argumentation community faces various problems, such as the lack of a shared interchange format for arguments along with the lack of ability to implement complex systems of arguments from high-level specifications. The first problem is addressed by the AIF, which provides a common language to exchange argumentation concepts among agents in a MAS. However, AIF does not solve the implementation problem. The AIF language is abstract and solely concerned with the structure of argument, while implemented multi-agent systems are concrete and need social constraints via protocols. This means that there is a gap between argument specification languages and multi-agent systems implementation languages which we bridge in chapters 4 and 5.

Chapter 4

Argument Specification Language

Although, significant progress has been made in argumentation community for modelling agent communication in abstract way (using argument specification languages), there remain major barriers to make argumentation systems practical and to implement (deploy) argumentation systems. This means that there is a gap between argument specification languages and multi-agent deployment languages.

This thesis will attempt to close the gap between standard argument specification and deployable protocol by automating the synthesis of protocols (in LCC) from argument specifications (ideally written in the AIF). As we shall see later in the thesis, it is not possible to fully automate synthesis starting only from the AIF because it does not capture some concepts that are essential to the choice of protocol structure. Some of these missing concepts we need to obtain from the user and some of them from the development (implementation) language (see Figure 4.1).

This chapter proposes a mechanism by which the missing concepts might be obtained from the user. We will propose a new intermediate language between the AIF and LCC called a Dialogue Interaction Diagram (DID), which is an extension of AIF and used to specify the dialogue game agent protocol in an abstract way.

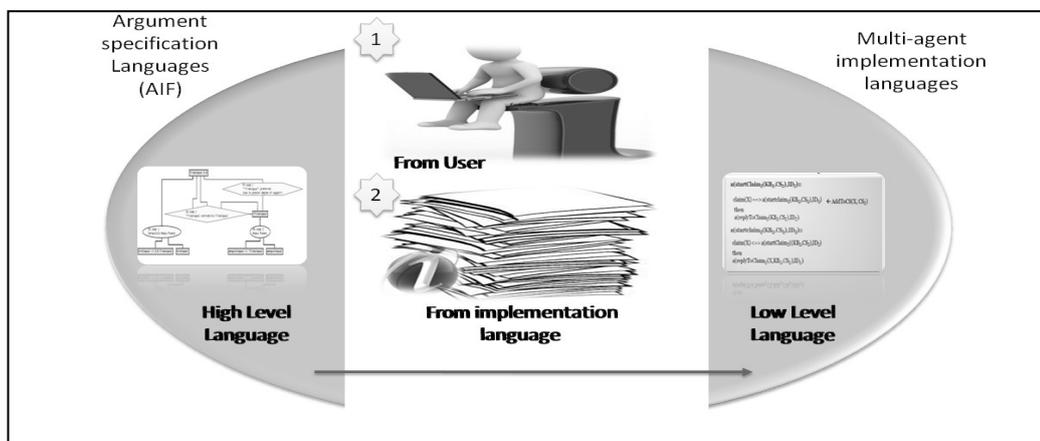


Figure 4.1: Missing Concepts between AIF and Agent Protocol

We open this chapter with a discussion of dialogue game agent protocol concepts (dialogue games and agent protocol implementation concepts) in Section 4.1. This is followed by a graphical and formal description of DID language in Section 4.2. DID for embedding dialogues is presented in Section 4.3. An extension of DID for modelling dialogue between $N > 2$ agents is presented in Section 4.4. Finally, Section 4.5 summarises the DID language, and justification is given for creating and using DID as a high-level dialogue game protocol language.

4.1 Agent Protocol Concepts for Argumentation between Two Agents

In order to represent an argument protocol in full, nine concepts are required:

- (1) Locutions;
- (2) Participants Commitment Store and Commitment rules;
- (3) Structural rules (reply rules or dialogue rules);
- (4) Turn Taking rules (Next player rules);
- (5) Starting rules (commencement rules);
- (6) Termination rules;
- (7) Post-condition rules define the conditions which must always be true just after the locution utterance;
- (8) Pre-condition rules;
- (9) Sender and receiver agents roles: a set of functions that an agent can use to interact with another agent. Each role identifies the messages that an agent can send or receive.

The first six concepts can be found in most of the existing dialogue games [Hamblin, 1970; Walton and Krabbe, 1995; Prakken, 2000 ; Mcburney et. al., 2003]. However,

the last three concepts are not found in most of the existing dialogue games, [Hamblin, 1970; Walton and Krabbe, 1995; Prakken, 2000] which makes it difficult to generate the multi-agent protocol automatically. Post-condition rules [Atkinson et al., 2005; Modgil and McGinnis, 2007] could refer to the effect of a locution utterance on the receiver agent commitment stores as well as the effect of a locution utterance on the agent's mental state structure; Pre-condition rules [Modgil and McGinnis, 2007] could refer to three different conditions: (1) sender agent commitment stores at a particular time; (2) agent internal reasoning states; or (3) a strategy that enables agents to select exactly one of the moves (locutions) from the legal moves. The concepts of the pre-condition and post-condition rules are imposed on utterance locutions and helps to control agent behaviour. Pre-condition allows an agent to utter a specific locution only when this agent has a prior argument or proof from its knowledge base or commitment stores. Sender and receiver agent roles [Willmott et al., 2006; Modgil and McGinnis, 2007] in relation to the dialogue help to control the way the dialogue proceeds.

All these concepts need to be presented in the AIF in order to perform the automated synthesis. Unfortunately, AIF does not possess the following nine concepts: Locutions; Participants Commitment Store and Commitment rules; Structural rules; Turn Taking rules; Starting rules; Termination rules; Post-condition rules; Pre-condition rules; and Sender and receiver agents roles. The next section extends the AIF to enable it to represent the dialogue game agent protocol concepts.

4.2 Dialogue Interaction Diagram (An Extension of AIF)

In this section, we propose a new language called Dialogue Interaction Diagram (DID) which is an extension of AIF. The extension of AIF to DID is not added automatically. In practice, DID is a new layer on top of AIF. DID is a new high-level specification language for multi-agent protocols, which allows to specify the dialogue game protocol in an abstract way. It has the nine concepts of the agent protocol [Locutions; Participants Commitment Store and Commitment rules; Structural rules; Turn Taking rules; Post-condition rules; Pre-condition rules; Locution types (Starting rules and Termination rules which are used to specify when

the dialogue starts and when the dialogue ends); and Sender and receiver agents roles]. It provides mechanisms to represent multi-agent interaction protocol rules between two agents by allowing the designer to specify the permitted moves and their relationship to each other.

DID is a recursive visual language which restricts agents moves to:

- (1) Unique-moves: agents can make just one move before the turn-taking shifts, and agents can reply just once to the other agent's move;
- (2) Immediate-reply moves: the turn-taking between agents switches after each move, moving from one level to the next level, and each agent must reply to the move of the previous agent.

This restriction is quite strict but it still allows us to include a large class of argumentation systems in our synthesiser; for instance, all argumentation systems that can be described as dialogue games. In general, we can synthesise arguments that can be described as a sequence of recursive steps (each of which involves turn taking between the pair of agents) terminating in a base case.

4.2.1 DID Elements

The basic element of every DID is a locution which is represented as an icon. A locution icon (as shown in Figure 4.2) is simply a rectangle divided into three sections. The topmost section contains the name of the locution (Locutions agent protocol concept). The left hand section contains sender attributes (Role name, Role arguments, and Agent ID), and the right hand section contains receiver attributes (Role name, Role arguments, and Agent ID).

The left hand section and the right hand section contain Sender and Receiver agents roles agent protocol concept. A rhombus shape represents conditions (Commitment rules, Post-condition and Pre-condition rules agent protocol concepts) that apply to each move; when connected to the left hand section it represents sender pre-conditions, and when connected to the right hand section it represents receiver post-conditions.

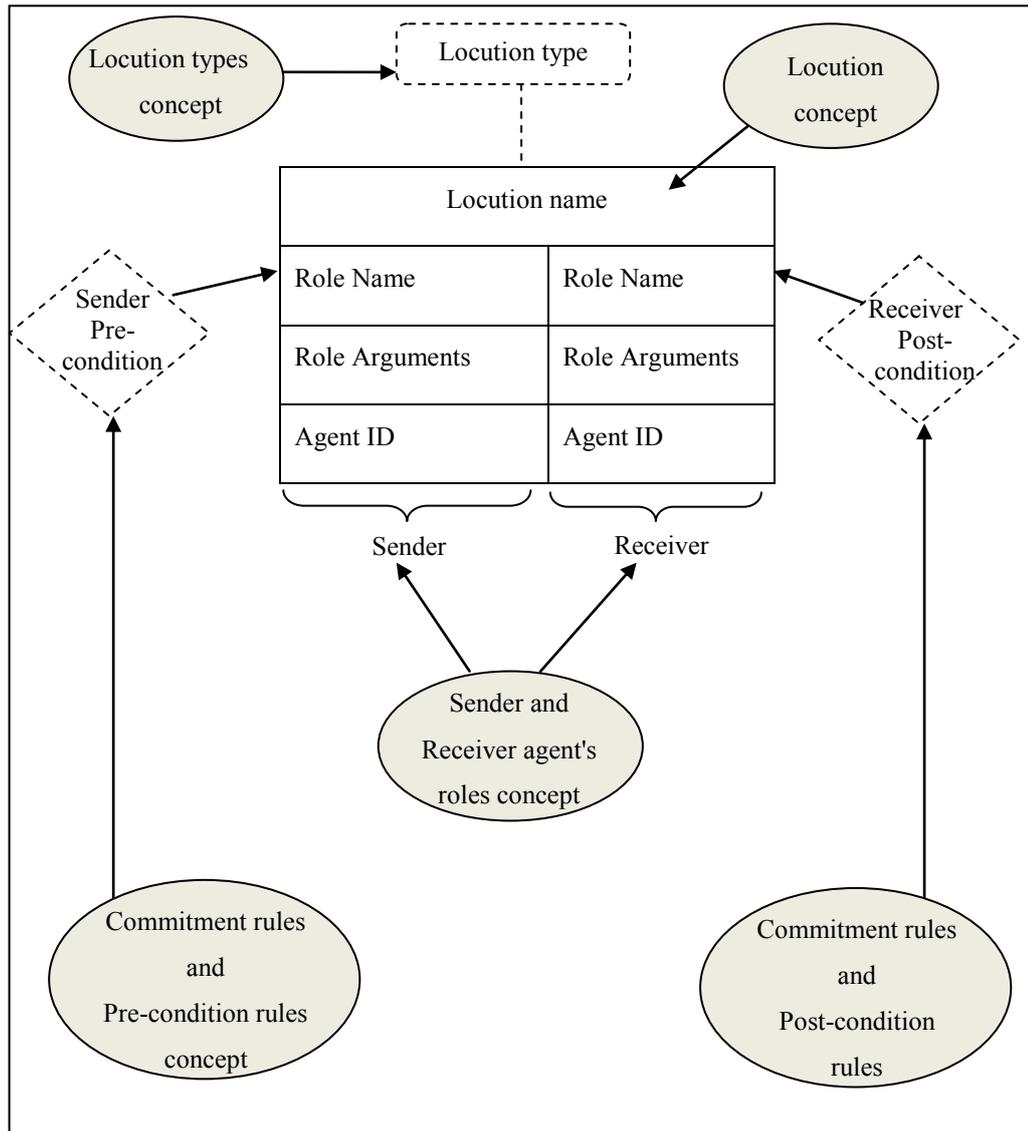


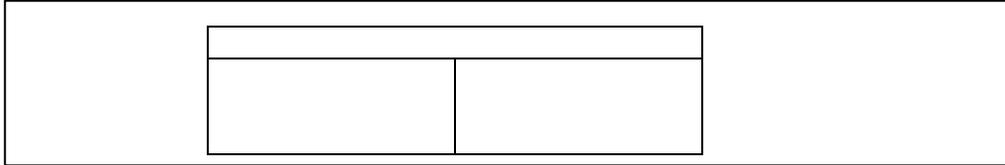
Figure 4.2: Locution Icon

Dotted rectangles represent the locution type (Locution types agent protocol concept): Starting (can be used to open a dialogue), Termination (can be used to terminate the dialogue), and Intermediate locution (can be used to remain in the dialogue).

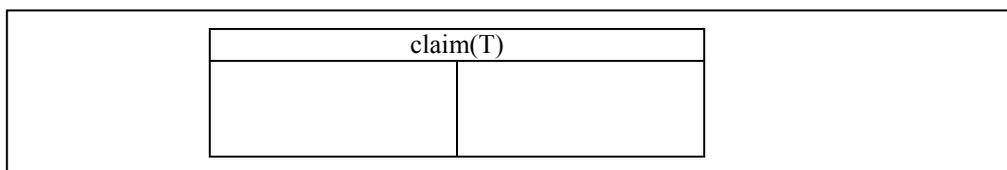
A DID is created by linking the locution icons together. The links between locution icons represent reply relations between arguments (Structural rules agent protocol concept). Finally, the turn-taking between agents switches after each move, moving from one level to the next level.

4.2.2 How to Draw a DID Diagram

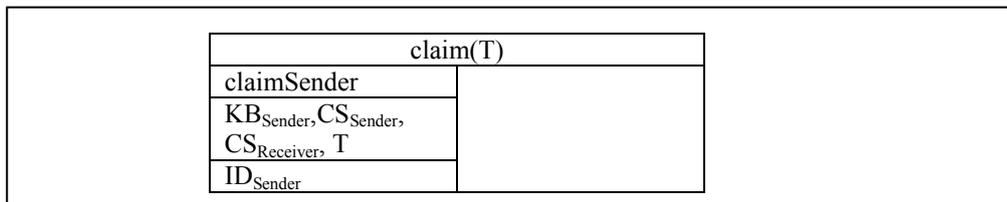
- (1) The first step is to identify dialogue game locutions.
- (2) The next step is to draw a rectangle for each locution, and divide it into three sections: 1) a rectangle on the top of the rectangle; 2) a rectangle on the left; 3) and a rectangle on the right. The below symbol represents a locution icon:



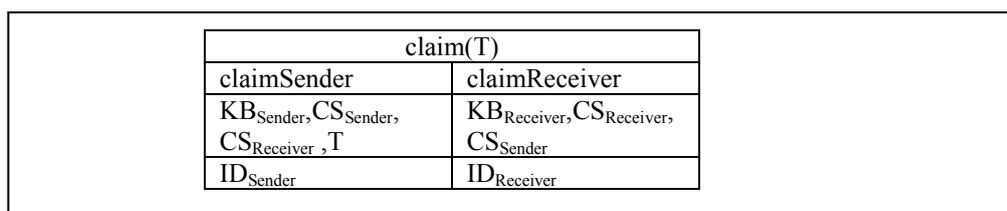
- a) Write the locution name (e.g. claim(T)) in the topmost section of the icon.



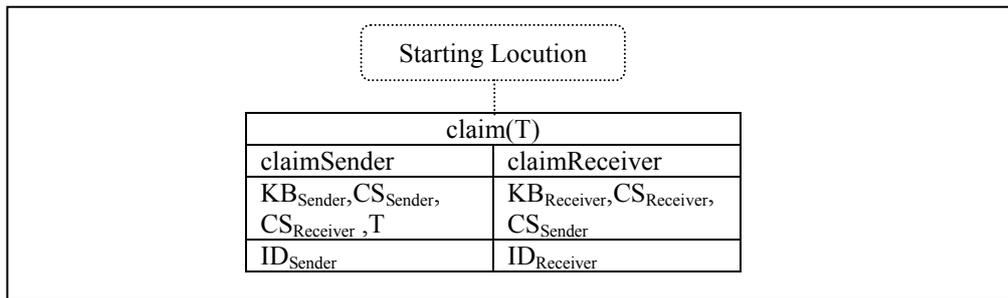
- b) Next, go to the left hand section and divide it into three rows and write the sender role name (e.g. claimSender), role arguments (e.g. $(KB_{\text{Sender}}, CS_{\text{Sender}}, CS_{\text{Receiver}}, T)$), and agent ID (e.g. ID_{Sender}). Note that the sender role name, arguments and agent ID must be the same for all locutions at the same level, since each level has one role (this restriction allows us to do the automatic agent protocol syntheseses).



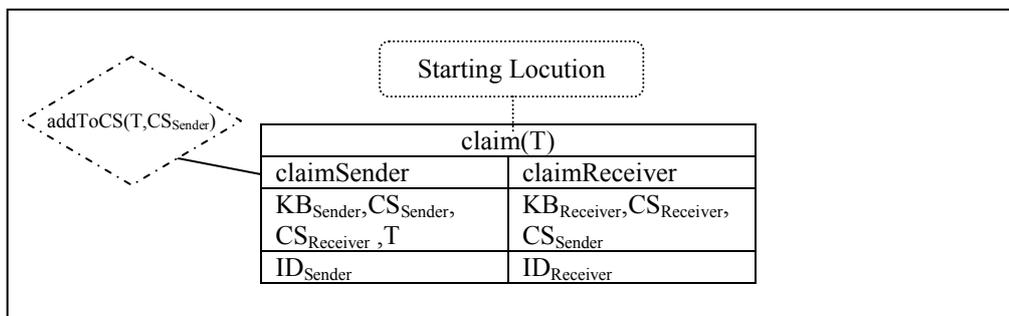
- c) Then, go to the right hand section, divide it into three rows and write the receiver role name (e.g. claimReceiver), role arguments(e.g. $(KB_{\text{Receiver}}, CS_{\text{Receiver}}, CS_{\text{Sender}})$), and agent ID (e.g. ID_{Receiver}). Note that the receiver role must be the same for all locutions at the same level (this restriction allows us to do the automatic agent protocol syntheseses).



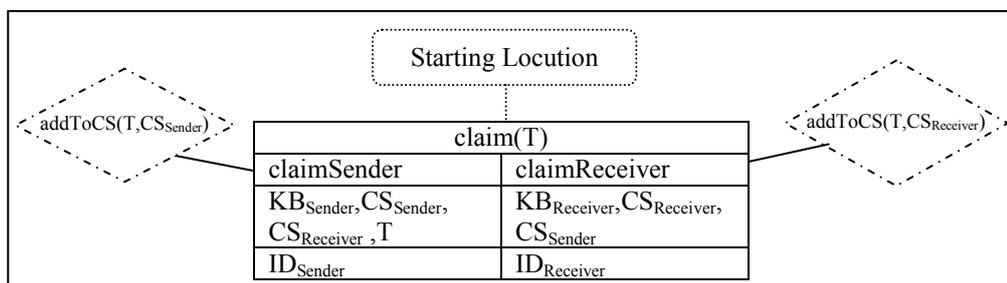
- d) Next, draw a rectangle with rounded corners and a dotted line instead of a solid line to signify locution type. Write the locution type inside the shape. Following this, draw a downward dotted line from this shape to the locution icon. Note that there are only three types of locutions: Starting Locution (SL), Intermediate Locution (IL) and Termination Locution (TL). Choose starting if an agent(s) is going to use this locution(s) in order to open a dialogue. Finally, choose intermediate if the next agent can make a move (utter locution(s)) after this locution, or choose termination if the agent needs to use this locution to end a dialogue.



- e) Draw a rhombus for the sender pre-condition with a dotted line. Write the pre-condition in the shape (e.g. addToCs(T,CS_{Sender})). Draw a solid line from this shape to the left hand section of the locution icon. This solid line is indicating that the sender agent can send this locution only if he is able to achieve this pre-condition. Note that if there is more than one pre-condition are connected to the sender, then either one of these two scenarios is applicable: 1) if the relation between pre-conditions is 'and' draw a rhombus shape for each pre-condition; 2) if the relation between pre-conditions is 'or' draw one rhombus shape and write all the pre-conditions in the shape and connect them by using 'or'.

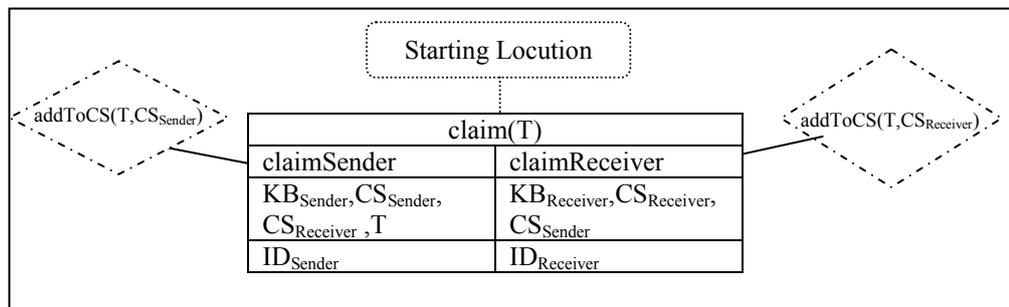


- f) Draw a rhombus for the receiver post-condition with a dotted line. Write the post-condition in the shape (e.g. $\text{addToCs}(T, \text{CS}_{\text{Receiver}})$). Draw a solid line from this shape to the right hand section of the locution icon. This solid line is indicating that the receiver agent satisfies this post-condition after it receives the locution. Note that if there is more than one post-condition are connected to the sender, then either one of these two scenarios is applicable: 1) if the relation between post-conditions is 'and' draw a rhombus shape for each post-condition; 2) if the relation between post-conditions is 'or' draw one rhombus shape and write all post-conditions in the shape and connect them by using 'or'.

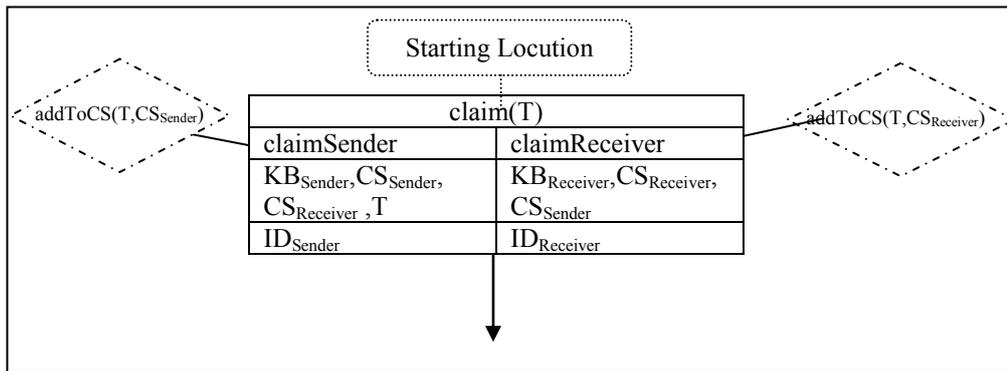


- (3) Step three is to connect the locutions together by following the reply rules:

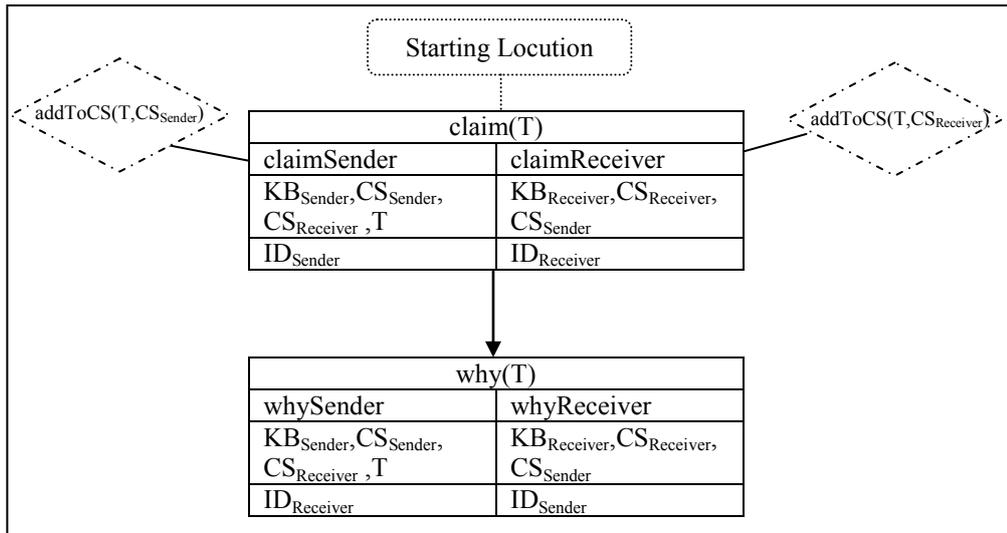
- a) Put the starting locution icon(s) at the top of the diagram.



- b) Draw a downward arrow from this icon indicating that when this process is completed (message sent and received), a new activity will begin on the following lower level (new message will be sent and received). Note that the turn taking between agents switches as we move from one level to the next level.



c) Put one reply locution below the downward arrow.



d) Continue drawing downward arrows and put the reply locution below the downward arrow (from the starting locution(s)) until all reply locutions to the starting locution appear in the diagram on level two.

e) Complete the DID diagram by continuing to draw arrow(s) between locutions until all reply rules of the dialogue game appear in the DID. Note that since the DID is used to represent multi-agent interaction protocol rules between two agents, you cannot draw any more arrows between two locution icons when the reply relation between two locution icons has already appeared in the DID.

4.2.3 Example (Persuasion Dialogue)

Figure 4.3 illustrates a DID structure of a persuasion dialogue [Parkken, 2000] (see chapter 3, section 3.4). In Figure 4.3, there are five locutions: three attack locutions

which have reply moves (*claim*, *argue* and *why*), and two surrender locutions (*concede* and *retract*) which do not have any reply moves. There are three types of locution: starting (*claim*), termination (*concede* and *retract*) and intermediate (*why* and *argue*).

In this example, a dialogue always starts with a *claim* and ends with a *concede* or *retract* locution. A rhombus shape represents conditions (pre- and post-conditions) that apply to each move. The variable *KB* (knowledge base list) represents the agent's private knowledge, defined as arguments expressed in the AIF. The variable *CS* (commitment store list) contains a set of arguments expressed in the AIF to which the player has committed during the discussion. Initially, the *CS* is empty.

In this dialogue, agent *P* can open the discussion by sending a *claim*(*T*) locution if he is able to satisfy the *addTopicToCS*(*T*,*CS*) pre-condition (note that adding an argument to the agent commitment store is a conditions that it is always satisfied). Then, turn-taking switches to agent *O*. *O* has to choose between two different possible reply locutions: *why*(*T*) or *concede*(*T*). *O* will make his choice using the pre-conditions which appear in the rhombus shape. In order to choose *concede*(*T*), *O* must be able to satisfy the four pre-conditions which connect with *concede*: 1) *findTopicInKB*(*T*, *KB_O*) which returns true if agent *O* is able to find *T* in its knowledge base *KB_O*; 2) *notFindTopicInCS*(*T*,*CS_O*) which returns true if agent *O* is not able to find *T* in its commitment store *CS_O*; 3) *notFindOppTopicInCS*(*not*(*T*),*CS_O*) which returns true if agent *O* is not able to find the opposite of *T* (*not*(*T*)) in its commitment store *CS_O*; 4) *addTopicToCS*(*T*,*CS_O*) which always returns true and results in agent *O* adding *T* to its commitment store *CS_O*. If *O* is not able to utter *concede*(*T*) because the explained pre-conditions are not satisfied, then *O* will send *why*(*T*). After that, the turn switches to *P*, and so on. The argument terminates once *P* or *O* sends *concede* or *retract* locutions.

The basic Scenario of the Interaction Protocol of Persuasion Dialogue

Figure 4.4 represents the persuasion dialogue graph of the complex car safety example (see chapter 3, section 3.2):

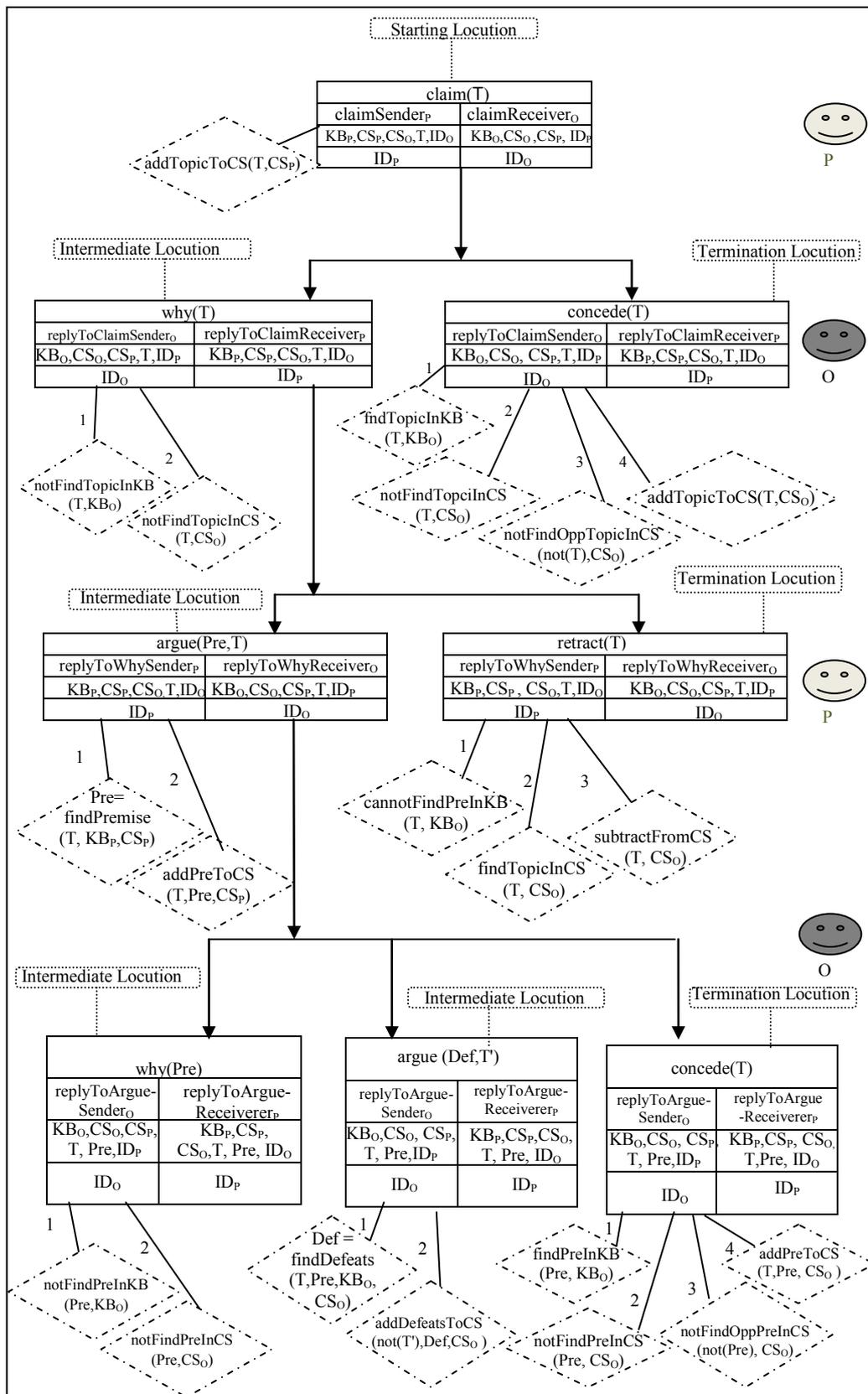


Figure 4.3 DID Structure of a Persuasion Dialogue

- (1) Dialogue takes place between two agents, P and O .
- (2) P has KB_P and CS_P , and O has KB_O and CS_O .
- (3) Initially the CS_P and CS_O are empty.
- (4) P and O can access both CS_P and CS_O .
- (5) P opens the discussion by sending *claim*("My car is safe").
- (6) O checks with its argumentation system AS_O ($AS_O = \{KB_O, CS_O\}$) whether "My car is safe" is acceptable or not. It finds that "My car is safe" is not acceptable,
- (7) O challenges "My car is safe". In others words, it asks what is the reason behind P 's proposal of "My car is safe". In this example, O will challenge "My car is safe" by sending the *why*("Why is your car safe") locution.
- (8) P responds to the challenge by declaring the supporting premises *Pre* for "My car is safe". In this example, P is offering grounds for a claim by sending *argue*("Since it has an airbag") locution.
- (9) O checks with its argumentation system AS_O whether "if car has an airbag, then the car is safe" is acceptable or not. In this example, O finds a counterargument for P 's argument and sends an *argue*("Your car is not safe since the newspapers recently reported on airbags expanding without cause") locution.
- (10) P finds a counterargument for O 's argument and sends an *argue*("Newspaper reports are very unreliable sources of technological information") locution.
- (11) O finds a counterargument for P 's argument and sends an *argue*("Still your car is not safe, since its maximum speed is very high") locution.
- (12) P checks with its argumentation system AS_P whether "if the car maximum speed is very high, then the car is not safe" is acceptable or not. In this example P finds that it is and retracts his main claim by sending a *retract*("My car is safe") locution.

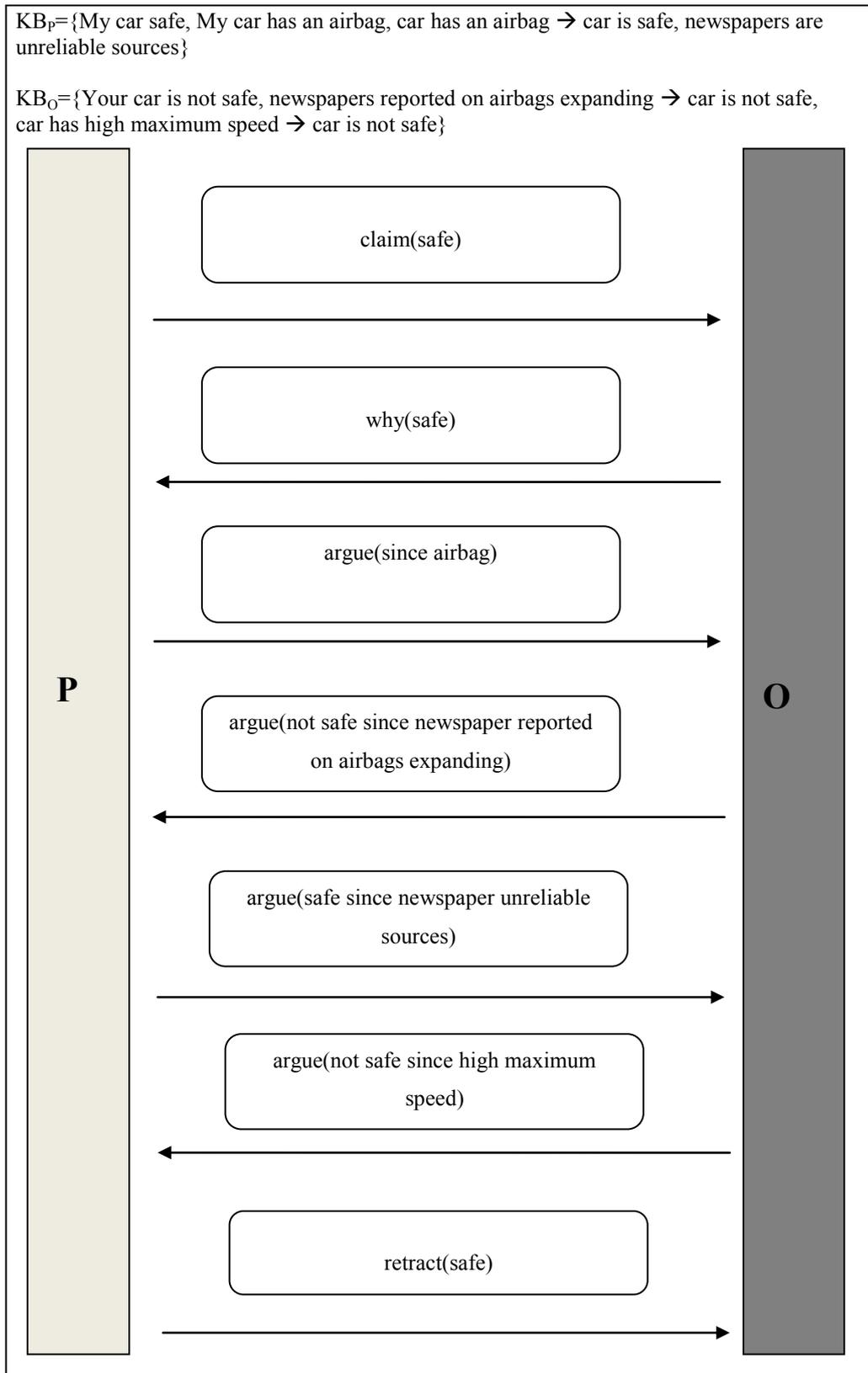


Figure 4.4: The Complex Car Safety Example

- (13) The commitment stores of P and O at the end of the dialogue are:
- $CS_P = \{\text{My car has an airbag, Newspapers are unreliable sources}\}$
 - $CS_O = \{\text{Your car is not safe, Newspapers reported on airbags expanding} \rightarrow \text{car is not safe, car has high maximum speed} \rightarrow \text{car is not safe}\}$

4.2.4 DID for Two Agents Formal Definition

Up to this point we have explained the DID syntax and how to use it and draw the DID diagrams. However, some readers may be interested to understand formally the meaning of the DID syntax. One way to do this is to use an existing formal definitions language from agents community such as Prakken's dialogue formal specification language [Parkken, 2000].

In this section, we formally specify the DID for two agents, as an extension of AIF. This formal definition called Dialogue Formal Specification Language (DFSL) is based on Prakken's framework [Parkken, 2000]. It is used to describe dialogue (argument) interaction protocol rules in a high-level way. Readers not interested in such details are encouraged to skip ahead to Section 4.3.

Definition 1: Dialogue

A dialogue protocol ' D ' is defined as a tuple:

(L , Players, CS, KB, Roles, Acts, ActType, Replies, Moves, LegalMoves) where:

Definition 2: Topic

L is a set of strings which specifies the dialogue topic;

$Args(L)$ is a set of all well-formed AIF arguments expressed as *I-nodes*, therefore $Args(L) \subseteq I\text{-nodes}$ (see chapter 3 for more information about I-nodes).

Definition 3: Players

Players = {player₁, player₂}

Where,

- Each player $player_i$ has its own commitment store set $CS_i \subseteq \wp(Args(L))$, which contains a set of propositions to which the player is committed in the discussion¹⁷.
- Each player $player_i$ has its own knowledge base or beliefs set $KB_i \subseteq \wp(Args(L))$, which represents the propositions on which the agent believes.

Definition 4: Commitment Store

'CS' is a function which gives the commitment store set of the player at a particular move.

$$CS: \text{Players} \times \text{Moves} \rightarrow \wp(Args(L))$$

Initially $CS(player_i, M_1) = \emptyset$, where $i = 1$ or 2

Definition 5: Knowledge Base

'KB' is a function which gives the knowledge base set of the player

$$KB: \text{Players} \rightarrow \wp(Args(L))$$

Definition 6: Roles

Roles = $\{r_1, r_2, \dots, r_{m-1}, r_m\}$ is a set of role identifiers.

Where $m \geq 2$ (there are at least two roles: one for the first agent and one for the second agent)

¹⁷ For any set S :

$\wp(S)$ = the powerset of any set S

\subseteq = a partial order on the set $\wp(S)$ of all subsets of S .

Definition 7: Acts

'Acts' is the set of speech acts (permitted messages or moves).

$Acts = \{loc(T_1, T_2, \dots, T_n) \text{ such that for every } n \geq i \geq 1, T_i \in \text{Args}(L)\}$

Definition 8: ActType

'ActType' is a function which determines the type of 'Act'.

ActType: $Acts \rightarrow \wp(\text{Types})$

Where,

- $\text{Types} = \{\text{Starting, Intermediate, Termination}\}$
- Starting: to open a dialogue,
- Intermediate: to remain in the dialogue,
- Termination: to terminate the dialogue.

Definition 9: Replies

'Replies' is a function which takes 'Acts' and return its possible replies according to the dialogue protocol.

Replies : $Acts \rightarrow \wp(Acts)$

For instance $\text{Replies}(\text{claim}(T)) = \{\text{why}(T), \text{concede}(T)\}$

Definition 10: Pre-conditions

'PreC' is a function which specifies the move pre-conditions according to the dialogue protocol. It takes as input parameters an act, the sender's commitment store, and the sender's knowledge base and returns a Boolean.

PreC : $Acts \times \wp(\text{args}(L)) \times \wp(\text{args}(L)) \rightarrow \text{Boolean}$

For example:

$$\text{PreC}(\text{claim}(\alpha), \text{CS}(\text{player}_1, M_t), \text{KB}(\text{player}_1)) = \exists \alpha \in \text{CS}(\text{player}_1, M_t) \cup \text{KB}(\text{player}_1)$$

Definition 11: Post-conditions

'PostC' is a function which specifies the move post-conditions according to the dialogue protocol. It takes as input parameters an act, the receiver's commitment store, and the receiver's knowledge base and returns a Boolean.

$$\text{PostC}: \text{Acts} \times \wp(\text{args}(\text{L})) \times \wp(\text{args}(\text{L})) \rightarrow \text{Boolean}$$

Definition 12: Move

A move $M_t \in \text{Moves}$, $t \geq 1$, is defined as:

$$M_t = (\text{player}_t, \text{act}_t, M_{t-1}, \text{nextPlayer}_t, \text{sender}_t, \text{receiver}_t)$$

Where,

- $\text{Player}_t \in \text{Players}$ represents the player of the move,
- $\text{Act}_t \in \text{Acts}$ represents the speech act performed in the move,
- $M_{t-1} \in \text{Moves} \cup \{\text{null}\}$ represents the previous move (M_t is a reply to M_{t-1}),
- $\text{nextPlayer}_t \in \text{Players} \cup \{\text{null}\}$ represents the next player in the dialogue,
- $\text{sender}_t \in \text{Roles}$ represents the role identifier of player (sender agent),
- $\text{receiver}_t \in \text{Roles}$ represents the role identifier of the nextPlayer (receiver agent),

Definition 13: Legal Move for Two

'legalMove' is a function which specifies the legal moves at a particular moment in the dialogue. It takes the dialogue history (list or sequence of moves) at a particular moment and the commitment store of the two players:

LegalMovesTwo: $\text{MoveSeq} \times \wp(\text{args}(L)) \times \wp(\text{args}(L)) \rightarrow \wp(\text{Moves})$

Rule 1: (Start a Dialogue)

This rule says that a dialogue always starts with a Starting act:

$\text{LegalMovesTwo}([], \text{CS}_1, \text{CS}_2) = \{ M_1 \}$

Where,

- $M_1 = (\text{player}_1, \text{act}_1, \text{null}, \text{player}_2, \text{sRole}_1, \text{rRole}_1)$,
- $\text{ActType}(\text{act}_1) = \text{Starting}$,
- $\text{PreC}(\text{act}_1, \text{KB}_1, \text{CS}_1) = \text{true}$, where $\text{KB}(\text{player}_1) = \text{KB}_1$
- $\text{PostC}(\text{act}_1, \text{KB}_2, \text{CS}_2) = \text{true}$, where $\text{KB}(\text{player}_2) = \text{KB}_2$

Rule 2: (Dialogue Termination)

This rule says that a dialogue always terminates with a Termination act:

$\text{LegalMovesTwo}([M_1, M_2, \dots, M_n], \text{CS}_1, \text{CS}_2) = \emptyset$

if

- $M_n = (\text{player}_n, \text{act}_n, M_{n-1}, \text{null}, \text{sRole}_n, \text{rRole}_n)$,
- $\text{ActType}(\text{act}_n) = \text{Termination}$,
- $\text{PreC}(\text{act}_n, \text{KB}_n, \text{CS}_n) = \text{true}$, where:
 - $\text{KB}(\text{player}_n) = \text{KB}_n$
 - $\text{CS}(\text{player}_n) = \text{CS}_n$
- $\text{PostC}(\text{act}_n, \text{KB}_m, \text{CS}_m) = \text{true}$, where:
 - $n \neq m$

- $KB(\text{player}_m) = KB_m$
- player_m represents the receiver of act_n
- $CS(\text{player}_m) = CS_m$

Rule 3: (Reply to an Agent's Move)

This rule says that only one move could be a reply to a move:

$\text{LegalMovesTwo}([M_1, M_2, \dots, M_t], CS_1, CS_2) = \{M_{t+1}\}$

if

- $\text{player}_i \neq \text{player}_j$
- $M_t = (\text{player}_i, \text{act}_t, M_{t-1}, \text{player}_j, \text{sRole}_t, \text{rRole}_t)$,
- $\text{ActTypes}(\text{act}_t) \in \{\text{Starting}, \text{Intermediate}\}$,
- $\text{PreC}(\text{act}_{t+1}, KB_i, CS_i) = \text{true}$, where:
 - $KB(\text{player}_i) = KB_i$
 - $CS(\text{player}_i) = CS_i$
- $M_{t+1} = (\text{player}_j, \text{act}_{t+1}, M_t, \text{player}_i, \text{sRole}_{t+1}, \text{rRole}_{t+1})$,
- $\text{act}_{t+1} \in \text{Replies}(\text{act}_t)$ (M_{t+1} replies to M_t),
- $\text{PostC}(\text{act}_{t+1}, KB_j, CS_j) = \text{true}$, where:
 - $KB(\text{player}_j) = KB_j$
 - $CS(\text{player}_j) = CS_j$

With this rule we are specifying also the turn-taking restriction. The sender of move M_t is the receiver of move M_{t+1} and the receiver of move M_t is the sender of move M_{t+1} .

Note that in order to send M_{t+1} , $player_i$ must satisfy PreC and after M_{t+1} , $player_j$ must satisfy PostC.

Example of DFSL of Persuasion Dialogue

This example describes the persuasion dialogues in chapter 3, section 3.4 [Parkken, 2000; Parkken, 2005] by using DFSL:

(1) Players:

In this dialogue, there are two participants: one participant (proponent 'P') attempts to persuade another participant (opponent 'O') to change his point of view about a particular topic 'T'.

Players={P,O}

(2) There are five locutions (Acts):

Acts = {claim(T), why(T), concede(T), argue(Pre,T), retract(T)}

(3) ActType(Act):

Act	ActType (Act)
claim	{Starting}
why	{ Intermediate }
concede	{Termination}
argue	{ Intermediate }
retract	{Termination}

(4) Replies(Act):

In the persuasion dialogue, the Replies rules are as follows:

Act	Replies(Act)
claim(T)	{why(T) , concede(T)}
why(T)	{argue(Pre), retract(T)}
concede(T)	∅
argue(Pre,T)	{ why(Pre), argue(Def,T'), concede(T)}
retract(T)	∅

(5) **PreC(Act,KB,CS):**

Lets Player = P. In the persuasion dialogue, the Pre-conditions are as follows:

Act	PreC(Act,KB,CS)	Note
claim(T)	addTopicToCS(T,CS _p)= true	<ul style="list-style-type: none"> • <i>addTopicToCS</i> function always returns true and results in agent <i>P</i> adding <i>T</i> to its commitment store <i>CS_p</i>
why(T)	notFindTopicInKB(T,KB _p) = true and notFindTopicInCS(T,CS _p) = true	<ul style="list-style-type: none"> • <i>notFindTopicInKB</i> function returns true if agent <i>P</i> is not able to find <i>T</i> in its Knowledge Base <i>KB_p</i>. • <i>notFindTopicInCS</i> function returns true if agent <i>P</i> is not able to find <i>T</i> in its Commitment Store <i>CS_p</i>.
concede(T)	findTopicInKB(T, KB _p) = true and notFindTopicInCS (T,CS _p) = true and notFindOppTopicInCS (not(T),CS _p) = true and addTopicToCS(T,CS _p)= true	<ul style="list-style-type: none"> • <i>findTopicInKB</i> function returns true if agent <i>P</i> is able to find <i>T</i> in its Knowledge Base <i>KB_p</i>. • <i>notFindTopicInCS</i> function returns true if agent <i>P</i> is not able to find <i>T</i> in its Commitment Store <i>CS_p</i>. • <i>notFindOppTopicInCS</i> which returns true if agent <i>P</i> is not able to find the opposite of <i>T</i> (<i>not(T)</i>) in its commitment store <i>CS_p</i>. • <i>addTopicToCS</i> function always returns true and results in agent <i>P</i> adding <i>T</i> to its commitment store <i>CS_p</i>.
argue(Pre, T)	Pre = findPremise(T, KB _p , CS _p) = true and addPreToCS(T,Pre,CS _p) = true	<ul style="list-style-type: none"> • Where T= topic and Pre= Promises which is used to support a claim (Topic) • <i>findPremise</i> function returns true if agent <i>P</i> is able to find Pre either in its knowledge base <i>KB_p</i> or its commitment store <i>CS_p</i>. • <i>addPreToCS</i> function always returns true and results in agent <i>P</i> adding <i>T</i> and <i>Pre</i> to its commitment store <i>CS_p</i>.

Act	PreC(Act,KB,CS)	Note
argue(Def, T')	Def = findDefeats(T, Pre, KB _P , CS _P) = true and addDefeatToCS (T',Def,CS _P) = true	<ul style="list-style-type: none"> • Where Def = Defeat an argument which is used to attacking an argument (T or Pre) with a counterarguments (Def) • <i>findDefeats</i> function returns true if agent <i>P</i> is able to find Def either in its knowledge base <i>KB_P</i> or its commitment store <i>CS_P</i>. • <i>addDefeatToCS</i> function always returns true and results in agent <i>P</i> adding <i>Def</i> and <i>T'(T or Pre)</i> to its commitment store <i>CS_P</i>.
retract(T)	cannotFindPreInKB(T, KB _P) = true and findTopicInCS (T, CS _P) = true and subtractFromCS(T,CS _P)= true	<ul style="list-style-type: none"> • <i>cannotFindPreInKB</i> function returns true if agent <i>P</i> is not able to find any promises (pre) in its knowledge base <i>KB_P</i> to support a claim (T). • <i>findTopicInCS</i> function returns true if agent <i>P</i> is able to find <i>T</i> in its Commitment Store <i>CS_P</i>. • <i>subtractFromCS</i> function always returns true and results in agent <i>P</i> subtract <i>T</i> from its commitment store <i>CS_P</i>.

(6) LegalMovesTwo(M_t , CS_P , CS_O)

From Figure 4.5 of the persuasion dialogue, we can see that:

- Dialogue begins by making a *claim* move

M_1 = initial move, ActType(Act(M_1)) = Starting and Act(M_1)= {claim}

- In the persuasion dialogue, the argument terminates once agents send a *concede* or *retract* locution. In other words, both *concede* and *retract* \in Termination. There is no reply move to these moves (there are no arrows coming out from these moves)

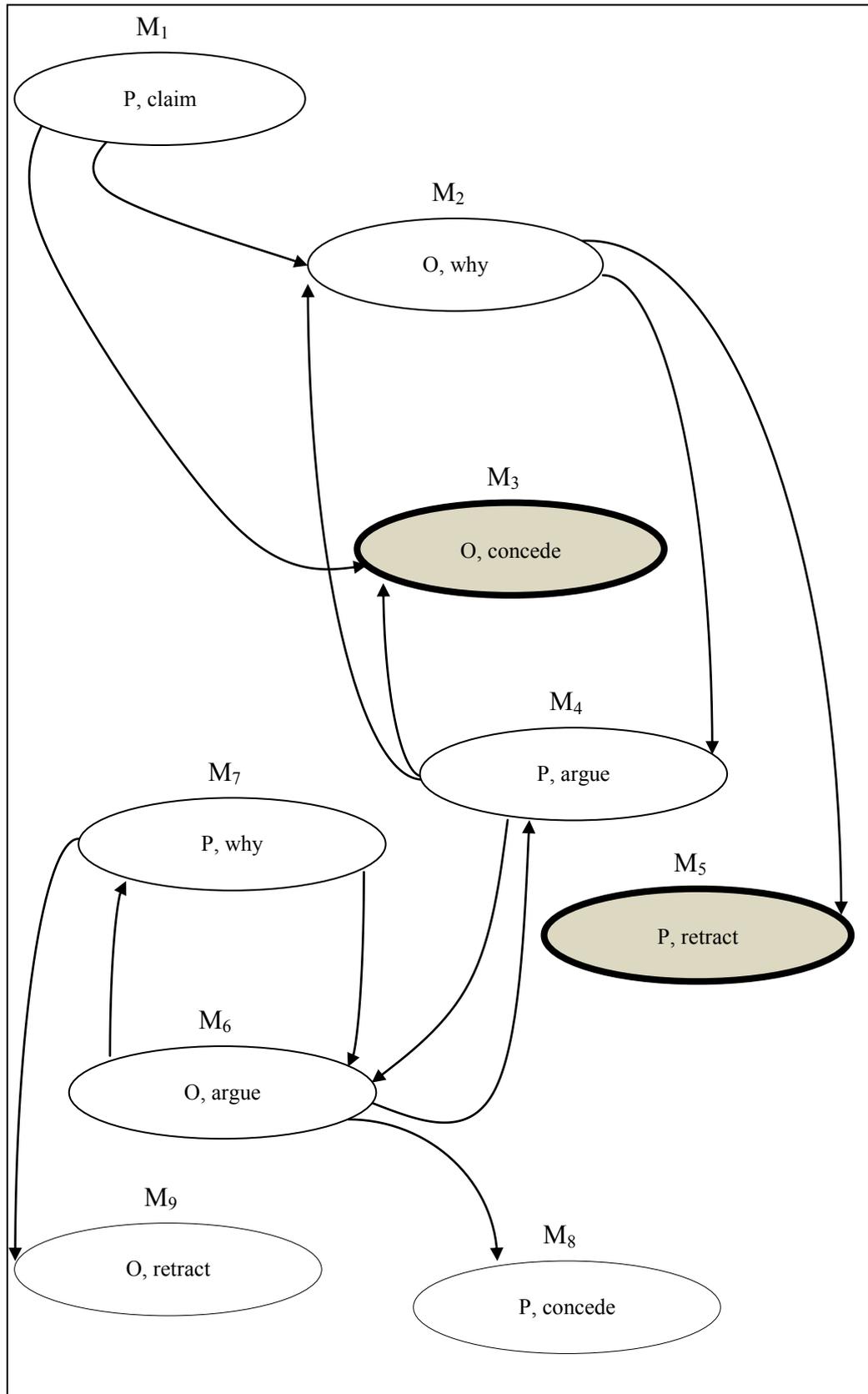


Figure 4.5: The Persuasion Dialogue Legal Moves

- Both *why* and *argue* \in {Intermediate}. There are several corresponding moves to these moves (there are arrows coming out from these moves).
- The turn-taking between participants switches after each move:
 - if M_1 then Player = P,
 - else NextPlayer = O iff Player = P

and NextPlayer = P iff Player = O

Appendix A presents a DID, DFSL and example of a negotiation dialogue.

4.3 Dialogue Interaction Diagram for Embedding Dialogue

4.3.1 DID for Embedding Dialogue

The DID can be used to model embedded dialogues. The DID allows agents to shift among different types of dialogues by connecting the starting locution of the sub-dialogue with the main dialogue locutions (changing the type of starting locution of the sub-dialogue to the intermediate locution in the main dialogue, and then connect this locution with all other locutions in the main dialogue).

4.3.2 DFSL for Embedding Dialogue

In this section we define embedded dialogue in a formal way. If you're not interested in such details, you can skip forward to section 4.4.

Definition 13: Embedded Dialogue

Let $D1$ and $D2$ are both dialogues. $Loc1$ is a start locution in $D1$ and $Loc2$ is a start locution in $D2$.

If $D2$ is a subdialogue of $D1$ then:

- $Loc2$ is an intermitted locution in $D1$
- $Loc2$ appears in all level of $D1$ instead of level one

- $D1$ will terminate if :
 - $D1$ termination conditions is satisfied, and
 - $D2$ has already terminated

4.3.3 Example

Black and Anthony's [Black and Anthony, 2007] work focuses on inquiry dialogues (see chapter 3, section 3.5 for more details about inquiry dialogues), which allow two agents to share knowledge in order to construct arguments in a dialogue within the medical domain. It provides a protocol as well as a specific strategy for modelling inquiry dialogues (a dialogue strategy that enables agents to select just one of the legal moves). Essentially, it embeds inquiry dialogues inside another inquiry dialogue and allows agents to shift between these inquiry dialogues.

Each inquiry dialogue has its own Question Store (QS), which is used to keep track of dialogue beliefs. During the dialogue, both agents will try to provide arguments for the belief(s) in the QS, which may lead them to open more sub-dialogues. These sub-dialogues have a topic whose consequent is the belief(s) in the current QS. In fact, an agent can open an inquiry dialogue by making an *open* move with the belief ' γ ' and create its QS and add ' γ ' to it ($QS = \{\gamma\}$). Then, if an agent wants to open a sub-dialogue, he can make a move with δ where $\delta = \beta_1, \beta_2, \beta_3, \dots, \beta_n \rightarrow \gamma$.

To terminate an inquiry dialogue, two *close* moves must appear next to each other and all sub-dialogues, which are embedded within this dialogue, must already be terminated.

DFSL

We will start by describing the inquiry dialogue in [Black and Anthony, 2007] by using DFSL:

(1) **Players:** $Players = \{P'', P\}$

Each player has its own KB and CS:

- P'' argumentation system $AS_{P''}$ ($AS_{P''} = \{KB_{P''}, CS_{P''}\}$)
- P argumentation system AS_P ($AS_P = \{KB_P, CS_P\}$)

(2) **There are four locutions (Acts):**

Acts = {open(γ), assert (Φ, γ), close(γ), subclose(δ)}

(3) **ActType(Act):**

Act	ActType (Act)	Note
open	{Starting, Intermediate}	In the main inquiry dialogue <i>open</i> locution type is starting but in the subdialogue we change the type of the <i>open</i> locution to intermediate in order to connect the two dialogues together.
assert	{ Intermediate }	
close	{Intermediate, Termination}	To terminate an inquiry dialogue, two <i>close</i> moves must appear next to each other. The first <i>close</i> type is intermediate (ActType(close) = {Intermediat}) and the second <i>close</i> type is termination (ActType(close) = {Termination}).
subclose	{ Intermediate }	

(4) **Replies(Act):**

In the inquiry dialogue the Replies rules are as follows:

Act	Replies(Act)	Note
open(γ)	{assert (Φ, γ), open(δ), close(γ)}	<ul style="list-style-type: none"> • (Φ, γ) is an argument , Φ is the argument support and γ is the argument claim
close(γ)	{assert (Φ, X), open(δ), close(γ)}	<ul style="list-style-type: none"> • When ActType (close) = {Intermediate} • X variable in <i>assert</i>(Φ, X) represents either γ or $\beta_n \in \delta$.
close(γ)	\emptyset	<ul style="list-style-type: none"> • when ActType (close) = {Termination}

Act	Replies(Act)	Note
assert(Φ, γ)	{assert ($\Phi 2, X2$),open($\delta 2$), close(γ)}	<ul style="list-style-type: none"> $X2$ variable in <i>assert</i> ($\Phi 2, X2$) represents either γ or δ
open(δ)	{assert ($\Phi 2, X2$),open($\delta 2$), close(γ), subclose(δ)}	
assert($\Phi 2, X2$)	{assert ($\Phi 3, X3$),open($\delta 3$), close(γ), subclose(δ),subclose(Z) }	<ul style="list-style-type: none"> Z variable in <i>subclose</i>(Z) represents either δ or $\delta 2$.
open($\delta 2$)	{assert ($\Phi 2, X2$),open($\delta 2$), close(γ), subclose(Z)}	<ul style="list-style-type: none"> Z variable in <i>subclose</i>(Z) represents either δ or $\delta 2$.
subclose(δ)	{assert ($\Phi 2, X2$),open($\delta 2$), close(γ), subclose(Z),subclsoe(δ)}	<ul style="list-style-type: none"> subclose(δ) after subclose(δ) ends sub-dialogue δ

(5) PreC(Act,KB,CS):

Lets Player = P'' . In an inquiry dialogue, the Pre-conditions are as follows:

Act	PreC(Act,KB,CS)	Note
Open(γ)	findInKB($\gamma, KB_{P''}$) = true and emptyCS($CS_{P''}$) = true and addToQueryStore (QS, γ) = true and addToOpenDialogue ($\gamma, OpenD$) = true	when ActType (open) = {Starting}, four functions must return true: <ul style="list-style-type: none"> <i>findInKB</i> function returns true if agent P'' is able to find γ in its Knowledge Base $KB_{P''}$. <i>emptyCS</i> function returns true if agent P'' Commitment Store $CS_{P''}$ is empty. <i>addToQueryStore</i> function always returns true and results in agent P'' adding γ to dialogue Question Store QS. <i>addToOpenDialogue</i> function always returns true and results in agent P'' adding γ to Open Dialogue list <i>OpenD</i>.

Act	PreC(Act,KB,CS)	Note
open(δ)	isRelationship(δ, γ) = true and findInQS(QS, γ) = true and findInKB(δ, KB) = true and notFindInQS(QS, δ) = true and addToQueryStore(QS2, δ) = true and addToOpenDialogue($\delta, OpenD$) = true and addToSubD($\delta, \gamma, SubD$) = true	when ActType (open) = {Intermediate}, seven functions must return true: <ul style="list-style-type: none"> • <i>isRelationship</i> function returns true if agent P'' is able to find a relation between δ and γ. • <i>findInQS(QS, γ)</i> function returns true if agent P'' is able to find γ in the dialogue Question Store QS. • <i>findInKB(δ, KB)</i> function returns true if agent P'' is able to find δ in its Knowledge Base $KB_{P''}$. • <i>notFindInQS</i> function returns true if agent P'' is not able to find $\delta = \beta_1, \beta_2, \beta_3, \dots, \beta_n$ in the dialogue Question Store QS. • <i>addToQueryStore</i> function always returns true and results in agent P'' adding $\delta = \beta_1, \beta_2, \beta_3, \dots, \beta_n$ to dialogue Question Store $QS2$. • <i>addToOpenDialogue($\delta, OpenD$)</i> function always returns true and results in agent P'' adding δ to Open Dialogue list $OpenD$. • <i>addToSubD</i> function always returns true and results in agent P'' adding δ to SubDialogue list $SubD$.
assert(Φ, γ)	findInQS(QS, γ) = true and notFindInCS($\Phi, CS_{P''}$) = true and findInKBorCS($((\Phi, \gamma), KB_{P''}, CS_p)$) = true and addToCS($\Phi, CS_{P''}$) = true	<ul style="list-style-type: none"> • <i>findInQS</i> function returns true if agent P'' is able to find γ in the dialogue Question Store QS. • <i>notFindInCS</i> function returns true if agent P'' is not able to find Φ in its Commitment Store $CS_{P''}$. • <i>findInKBorCS($((\Phi, \gamma), KB_{P''}, CS_p)$)</i> function returns true if agent P'' is able to find (Φ, γ) in either in its knowledge base $KB_{P''}$ or its commitment store $CS_{P''}$. • <i>addToCS($\Phi, CS_{P''}$)</i> function always returns true and results in agent P adding Φ to its commitment store $CS_{P''}$.

Act	PreC(Act,KB,CS)	Note
Assert (Φ, X)	$setInitialValueForX(X)$ $= true$ and $findInQS(QS, X) = true$ and $notFindInCS(\Phi, CS_{P''}) = true$ and $findInKBorCS((\Phi, X), KB_{P''}, CS_P) = true$ and $addToCS(\Phi, CS_{P''}) = true$	<ul style="list-style-type: none"> • $setInitialValueForX$ function always returns true and results in set initial value for X. Note that X can be either γ or $\beta_n \in \delta$. • (See $assert(\Phi, \gamma)$ for more information about functions definition)
close(γ)	$findInOpenDialogue(\gamma, OpenD) = true$ and $allSubDialogueClosed(\gamma, SubD, ClosedD) = true$ and ($notFindInKBandCS((\Phi, \gamma), KB_{P''}, CS_P) = true$ or $findInCS(\Phi, CS_{P''}) = true$) and ($noRelationship(\delta, \gamma) = true$ or $notfindInKB(\delta, KB_{P''}) = true$ or $findInQS(QS, \delta) = true$)	when ActType (close) = {Intermediate}, at last four functions of seven functions must return true: <ul style="list-style-type: none"> • $findInOpenDialogue$ function returns true if agent P'' is able to find γ in the Open Dialogue list $OpenD$. • $allSubDialogueClosed$ function returns true if all subdialogue of γ is already closed. • $notFindInKBandCS$ function returns true if agent P'' is not able to find (Φ, γ) in either in its knowledge base $KB_{P''}$ or other agent P commitment store CS_P. • $findInCS$ function returns true if agent P'' is able to find Φ in its commitment store $CS_{P''}$. • $noRelationship$ function returns true if agent P'' is not able to find a relation between δ and γ. • $notfindInKB$ function returns true if agent P'' is not able to find δ in its knowledge base $KB_{P''}$. • $findInQS$ function returns true if agent P'' is able to find $\delta = \beta_1, \beta_2, \beta_3, \dots, \beta_n$ in the dialogue Question Store QS.

Act	PreC(Act,KB,CS)	Note
close(γ)	(notFindInKbandCS($(\Phi, \gamma), KB_{P''}, CS_P$) = true or findInCS($\Phi, CS_{P''}$) = true) and (noRelationship(δ, γ) = true or notfindInKB($\delta, KB_{P''}$) = true or findInQS(QS, δ) = true) and addToClosedDialogue ($\gamma, ClosedD$) = true	when ActType (close) = {Termination}, at last three functions of seven functions must return true: <ul style="list-style-type: none"> • <i>cannotFindInKBandCS</i> function returns true if agent P'' is not able to find (Φ, γ) in either in its knowledge base $KB_{P''}$ or other agent P commitment store CS_P. • <i>findInCS</i> function returns true if agent P'' is able to find Φ in its commitment store $CS_{P''}$. • <i>noRelationship</i> function returns true if agent P'' is not able to find a relation between δ and γ. • <i>notfindInKB</i> function returns true if agent P'' is not able to find δ in its knowledge base $KB_{P''}$. • <i>findInQS</i> function returns true if agent P'' is able to find $\delta = \beta_1, \beta_2, \beta_3, \dots, \beta_n$ in the dialogue Question Store QS. • <i>addToClosedDialogue</i> function always returns true and results in agent P'' adding γ to closed Dialogue list <i>ClosedD</i>.
subclose(δ)	findInOpenDialogue ($\delta, OpenD$) = true and allSubDialogueClosed($\delta, SubD, ClosedD$) = true and (notFindInKBandCS ($(\Phi, \delta), KB_{P''}, CS_P$) = true or findInCS($\Phi, CS_{P''}$) = true) and (noRelationship(δ_2, δ) = true or notfindInKB($\delta_2, KB_{P''}$) = true or findInQS(QS, δ_2) = true)	when agent sends subclose(δ) after open, assert or subclose(Z), at last five functions of eight functions must return true: <ul style="list-style-type: none"> • <i>FindInOpenDialogue</i> • <i>allSubDialogueClosed</i> • <i>notFindInKBandCS</i> • <i>findInCS</i> • <i>noRelationship</i> • <i>notfindInKB</i> • <i>findInQS</i> (See <i>close(γ)</i> , ActType (close) = {Intermediate}, for more information about functions definition)

Act	PreC(Act,KB,CS)	Note
subclose(δ)	$($ notFindInKbandCS($(\Phi, \delta), K$ $B_p, CS_p) = \text{true}$ or findInCS(Φ, CS_p) = true $)$ and $($ noRelationship(δ_2, δ) = true or notfindInKB(δ_2, KB_p) = true or findInQS(QS, δ_2) = true $)$ and addToClosedDialogue $(\delta, \text{ClosedD}) = \text{true}$	when agent sends subclose(δ) after subclose(δ), at last three functions of seven functions must return true: <ul style="list-style-type: none"> • <i>cannotFindInKBandCS</i> • <i>findInCS</i> • <i>noRelationship</i> • <i>notfindInKB</i> • <i>findInQS</i> • <i>addToClosedDialogue</i> (See <i>close(γ)</i> , where ActType (close) = {Termination}, for more information about functions definition)

(6) LegalMovesTwo(M_t, CS_{A1}, CS_{A2})

From the inquiry dialogue depicted in Figure 4.8, we can see that:

- Dialogues begin by making an *open* move.

$M_1 = \text{initial move}$, ActType(Act(M_1)) = {Starting} and Act(M_1) = {open}

- In the inquiry dialogue, the argument terminates once one agent sends *close* which is followed by a *close move* by the second agent. In other words, to terminate an inquiry dialogue, two close moves must appear next to each other in the sequence
- *Assert, close, subclose* and *open* \in {Intermediate}. There are several corresponding moves to these moves (there are arrows coming out from these moves):

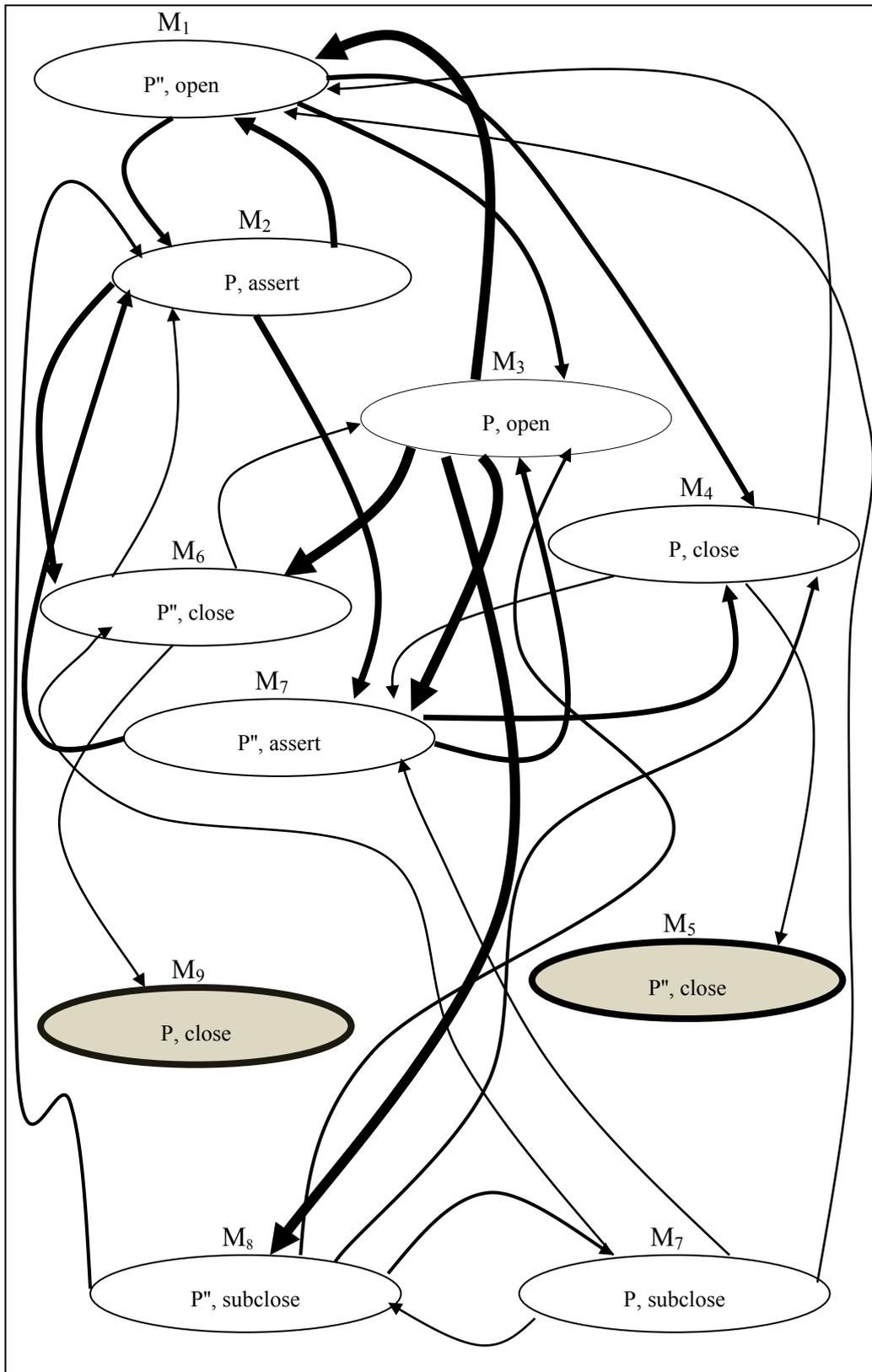


Figure 4.8: The Inquiry Dialogue Legal Moves

Bridging the Specification Protocol Gap in Argumentation

- *assert* move, by either P or P'' , could be followed by *Assert*, *close* and *open*.
 - *subclose* move, by either P or P'' , could be followed by *Assert*, *close*, *subclose* and *open*.
 - if the dialogue has not terminate yet, *close* move, by either P or P'' , could be followed by *Assert*, *close* and *open*
 - *open* move, by either P or P'' , could be followed by *Assert*, *close* and *open*.
- The turn-taking between participants switches after each move (the agents take it in turns to make moves):
 - if M_1 then $\text{Player} = P''$,
 - else $\text{NextPlayer} = P$ iff $\text{Player} = P''$ and $\text{NextPlayer} = P''$
iff $\text{Player} = P$

DID

Figure 4.9 illustrates the DID structure of an inquiry dialogue (note that pre-conditions and post-conditions for locutions are not shown in this figure but are shown in Figure 4.10(a), Figure 4.10(b) and Figure 4.10(c)). In Figure 4.9, there are four locutions: *open*, *assert*, *subclose* and *close*. There are three types of locutions: starting (*open*), termination (*close*), and intermediate (*assert*, *close*, *subclose* and *open*).

The dialogue always starts with an *open* and ends with a *close* locution. P'' can open the discussion by sending an *open*(γ) locution if he is able to satisfy the four pre-conditions which are connected to the sender role of this locution. Then, turn-taking switches to P . P has to choose between three different possible reply locutions: *assert*(Φ, γ), *open*(δ) or *close*(γ). P will make his choice using the pre-conditions that appear in the rhombus shape. For example, in order to choose *assert*(Φ, γ), P must be

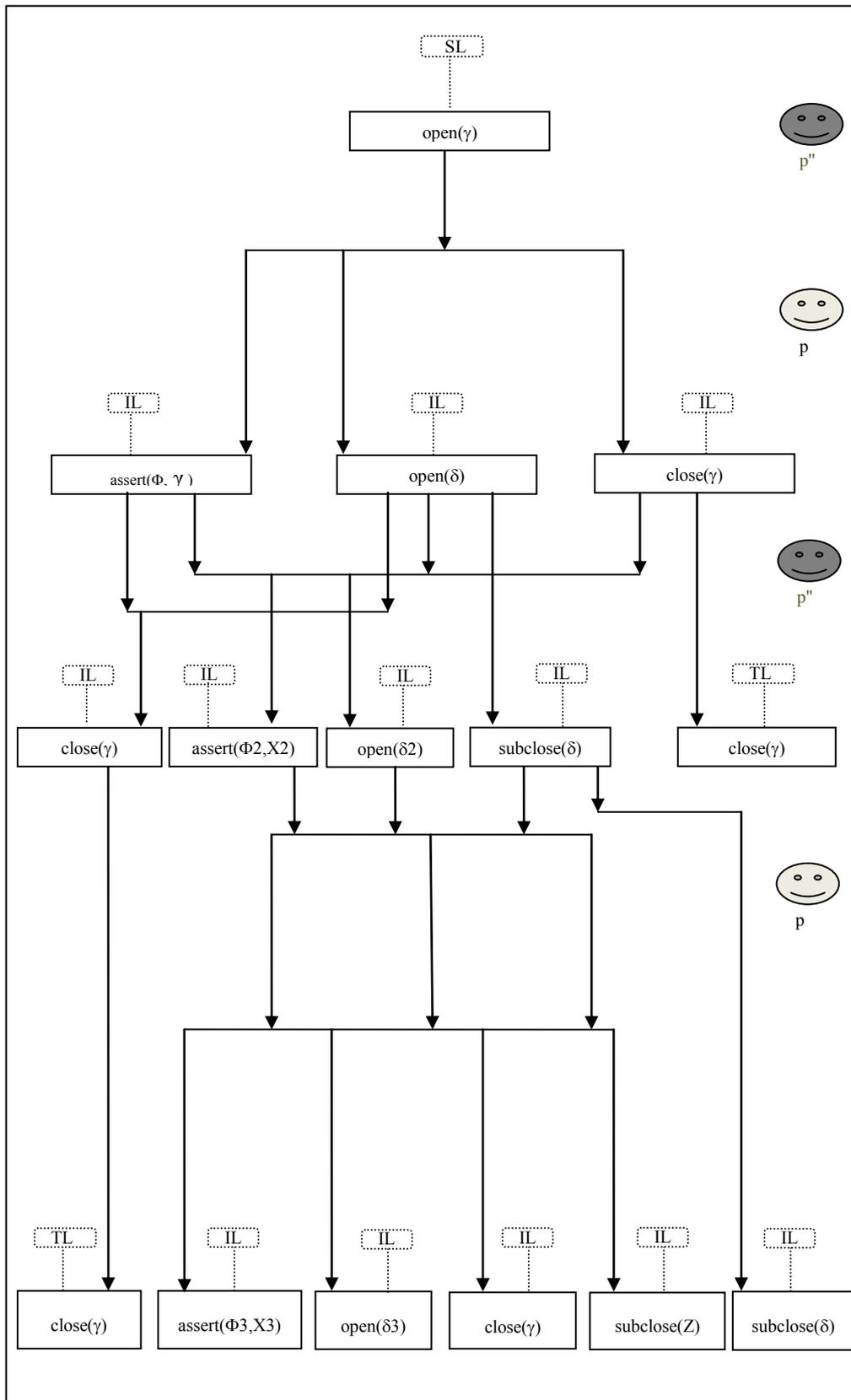


Figure 4.9: DID Structure of an Inquiry Dialogue

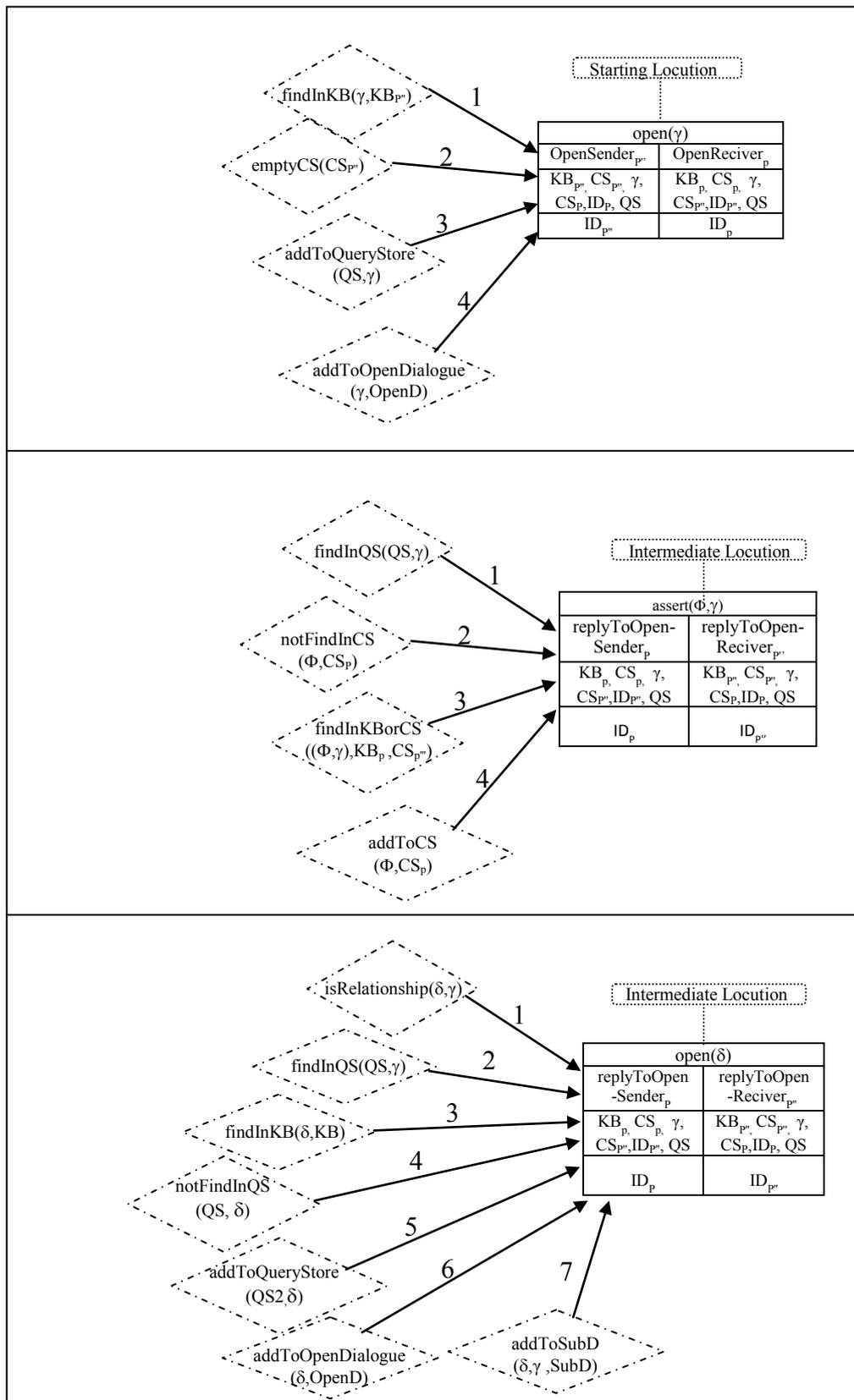


Figure 4.10(a): Inquiry Dialogue Locutions Pre-conditions and Post-conditions

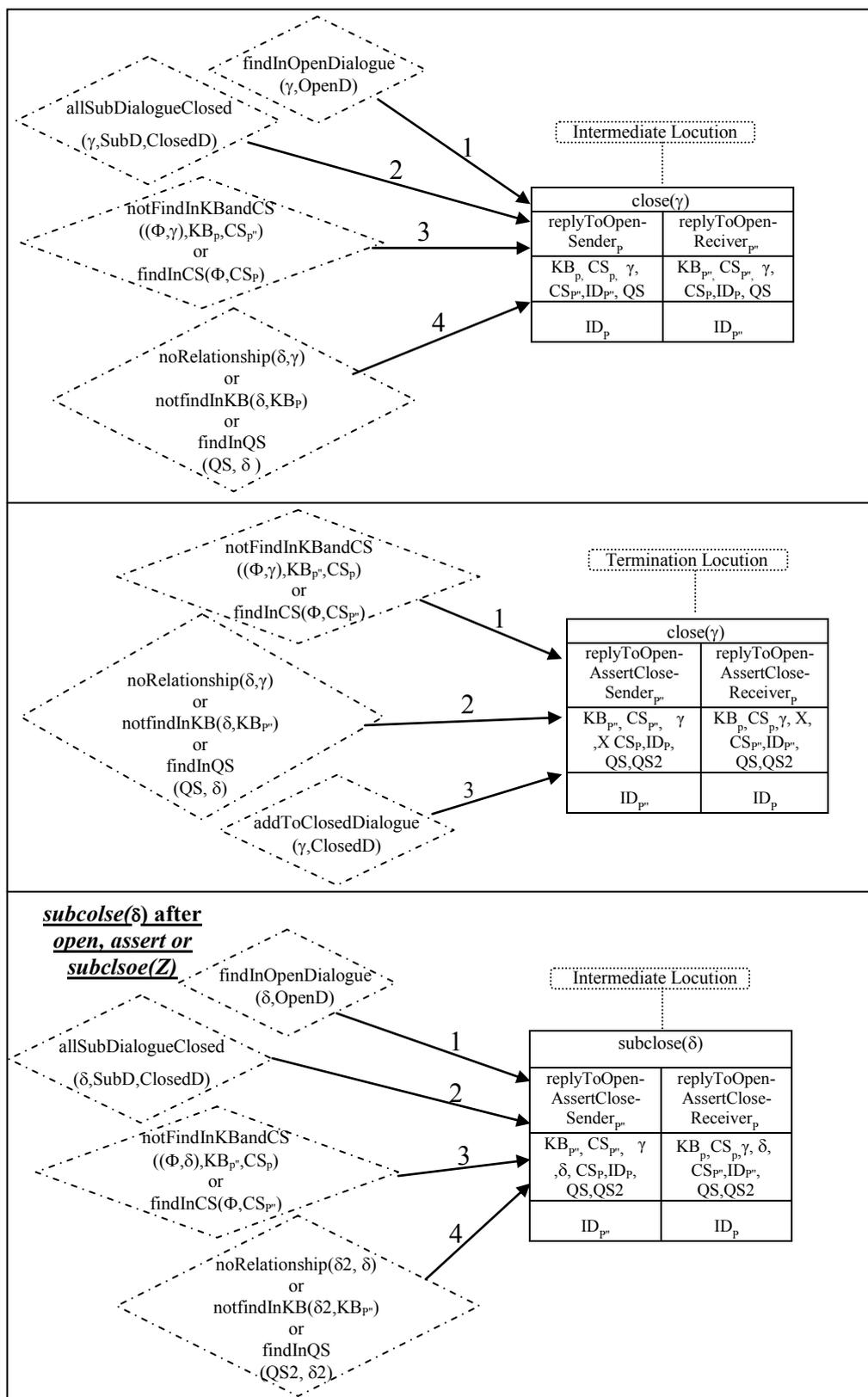


Figure 4.10 (b): Inquiry Dialogue Locutions Pre-conditions and Post-conditions

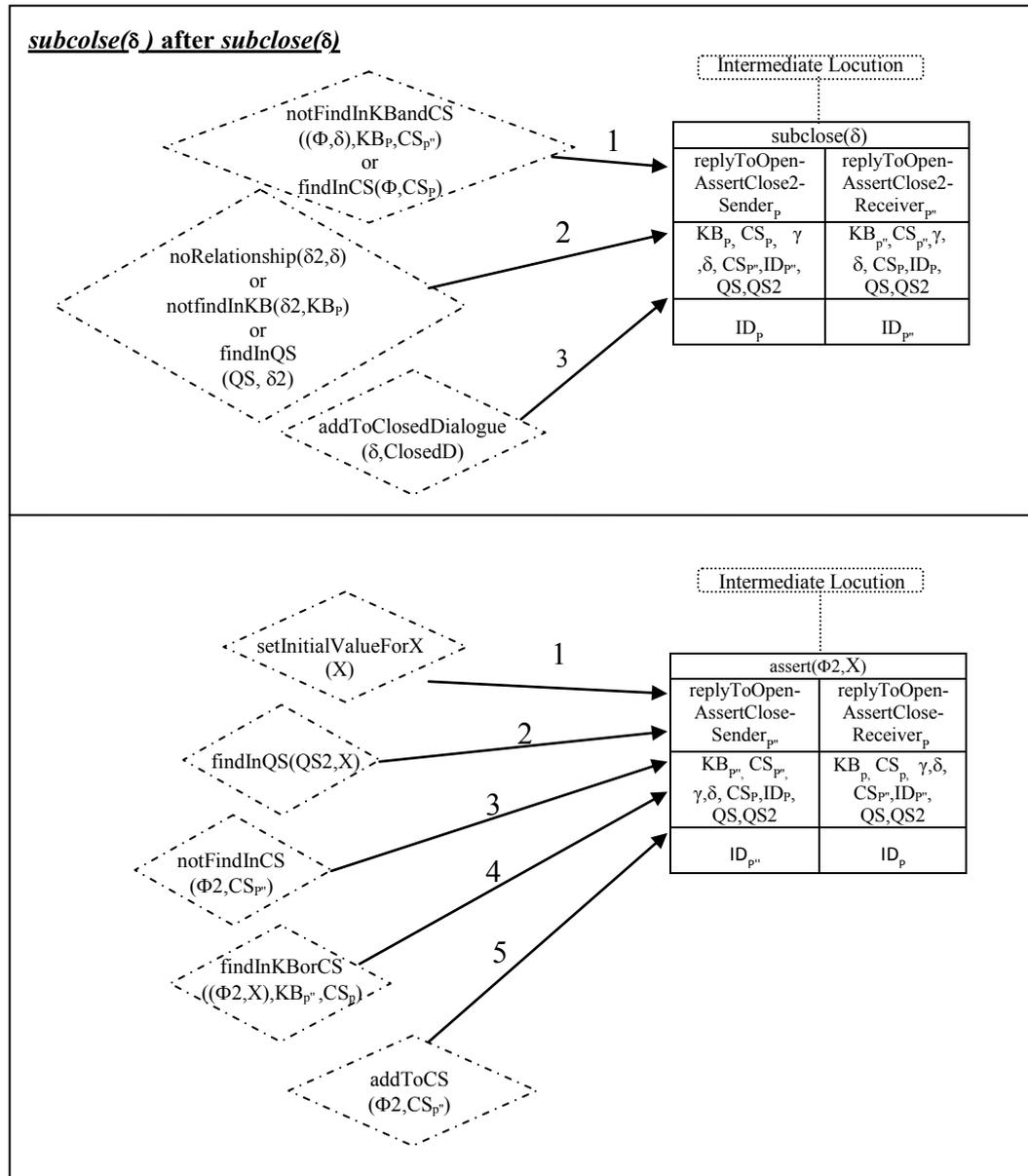


Figure 4.10 (C): Inquiry Dialogue Locutions Pre-conditions and Post-conditions

able to satisfy the four pre-conditions which connect with *assert*: (1) *findInQS*(γ) which returns true if agent P is able to find γ in the dialogue question store QS ; (2) *notFindInCS*(Φ, CS_P) which returns true if agent P is not able to find γ in its commitment store CS_P ; (3) *find*($(\Phi, \gamma), KB_P, CS_{P''}$) which returns true if agent P is able to find (Φ, γ) either in its knowledge base KB_P or in the commitment store of agent $P''(CS_{P''})$; (4) *addToCS* ($\Phi, CS_{P''}$) which always returns true and results on the agent P adding Φ to its commitment store CS_P . After that, the turn switches to P'' ,

and so forth. The argument terminates when two close moves appear next to each other. Note that X variable in $assert(\Phi_2, X)$ represents either γ or $\beta_n \in \delta$.

An example of inquiry dialogue [Black and Anthony, 2007] is shown in Figure 4.11. The goal of the dialogue is to find an argument for believing 'c'. The agent's knowledge bases are shown at the top of the figure.

In this example, there is one main dialogue (D1 with $QS1=\{c\}$ start at move 1) and three sub-dialogues (D2 with $QS2=\{b\}$ start at move 3, D3 with $QS1=\{a\}$ start at move 8, and D4 with $QS4=\{d, e\}$ start at move 16) are created during the augmentation process. The commitment store of agent P'' is changed at move 8 ($CS_{P''} = \{d\}$) and move 16 ($CS_{P''} = \{d, e, d \wedge e \rightarrow b, b \rightarrow c\}$). The commitment store of agent P is changed at move 9 ($CS_P = \{e\}$) and 13 ($CS_{P''} = \{e, d, d \wedge e \rightarrow b\}$). At move 18 the main dialogue ends after it succeeds in achieving its goal (finding an argument for the 'c' belief).

4.4 Dialogue Interaction Diagram for Argumentation between N-agent

4.4.1 Need for Dialogue Games among N-agent

At times, in order to solve a particular problem, more than two agents have to work together. Each agent has a responsibility to contribute to a finding final solution [Dignum and Vreeswijk, 2003].

For example, five members of a family, each with their own favourite holiday, try to decide where to go. This family can reach an acceptable solution and share their experience by allowing all family members to take part in the dialogue.

4.4.2 Issues of Dialogue Games among N-agent

Dignum and Vreeswijk's work [Dignum and Vreeswijk, 2003] highlights some of the key issues of N-agents' (multi-party) dialogues:

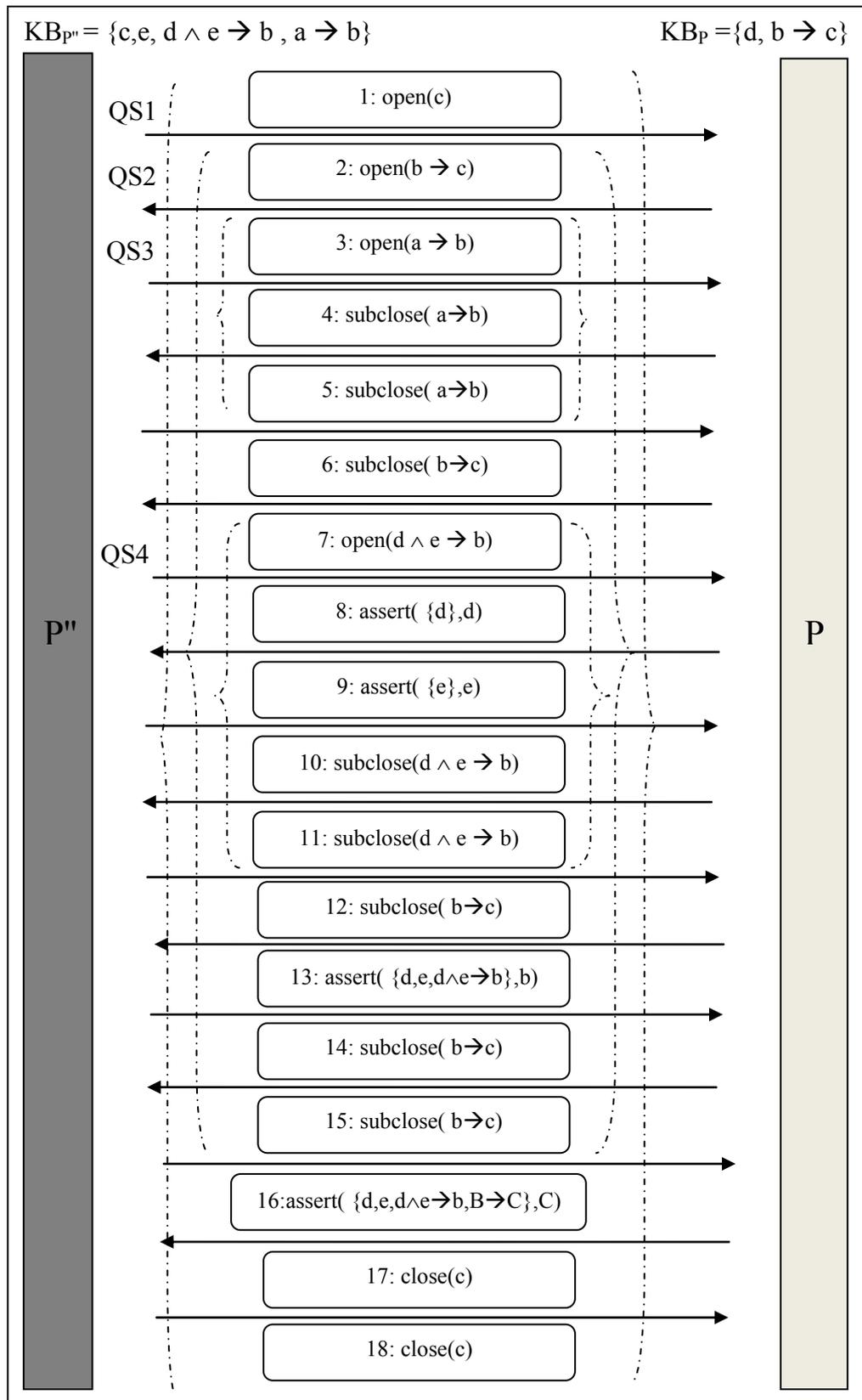


Figure 4.11: Embedded Inquiry Dialogue Example

(1) Open/closed system:

In two agent dialogue systems, neither agent can leave the dialogue. However, in N-agents' dialogue systems, there are two types:

- Open system: during a dialogue, an agent can join and leave the group.
- Closed system: during a dialogue, existing agents cannot leave the group and new agents cannot join the group. In other words, if the dialogue starts with N-agent, it must end with N-agent.

(2) Player's roles:

In two agent dialogue systems, one agent can be the speaker (e.g. proponent in persuasion dialogue) and the other agent must be the audience (e.g. opponent in the persuasion dialogue). However, in N-agents' dialogue systems, there can be more than one speaker agent and more than one audience agent.

(3) Addressing:

In two agent dialogue systems, one agent sends a message and the other agent receives the message. However, in N-agents' dialogue systems the following can happen:

- One-to-one system: one agent sends a message and one agent receives the message
- One-to-many system: one agent sends a message and more than one agent receives the message
- One-to-all system: one agent sends a message and all other agents receive the message

(4) Turn taking (coordination):

In two agent dialogue systems, there is a turn taking method (the speaker will become the audience in the next turn and so on). However, in N-agents' dialogue systems:

- One agent will take the next turn; or
- More than one agent will take the next turn; or
- The turn could pass from one agent to another (under some conditions).

(5) Termination:

In two agent dialogue systems, the dialogue will terminate when one (or both agents) has achieved its main goal. However, in N-agents' dialogue systems:

- All agents have to achieve the dialogue main goal (e.g. in a persuasion dialogue: all agents have to be persuaded); or
- The majority of agents have to achieve the dialogue main goal (e.g. in a persuasion dialogue: the majority of agents have to be persuaded)

In the following sections, we will present a new system for dialogue among N-agent. This system will be:

- A closed system; and
- A flexible addressing system (messages could be one-to-one, one-to-many, or one-to-all); and
- A system where more than one agent can take the next turn; and
- A flexible termination system (the software engineer can decide the termination condition).

4.4.3 Method for Dialogue Games among N-agent

In this section, we describe a method for dialogue among N agents. We adapted this method from [Ito and Shintani, 1996]. The idea is to consider the dialogue among N-agent as a dialogue between two agents by dividing agents into groups composed of two agents under certain conditions. For example, Figure 4.12 shows an example of a persuasion dialogue among seven agents (*A, B, C, D, E, F* and *G*):

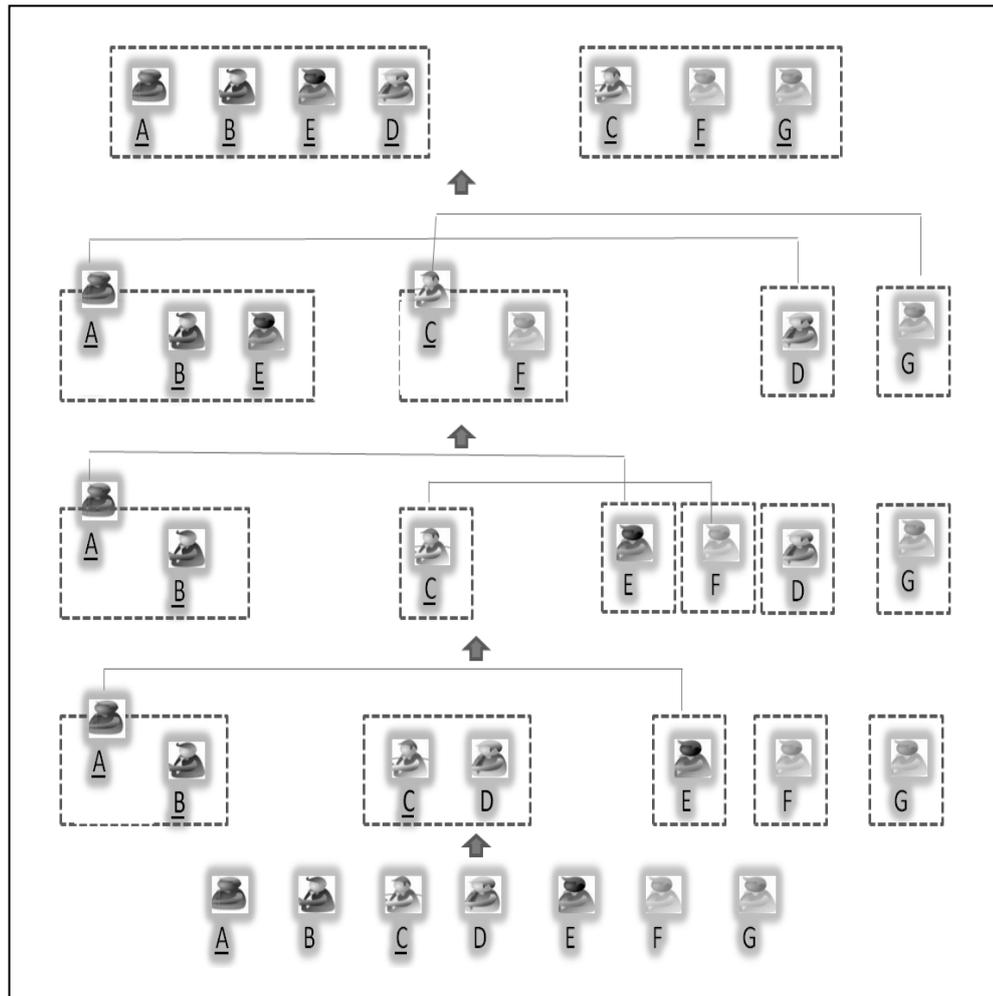


Figure 4.12: Dialogue Among N-agent

- (1) Agents *A* and *C* accept the main topic, whereas *B*, *D*, *E*, *F* and *G* reject the main topic (note that in this figure the accepted agents are underlined and the rejected agents are not underlined).
- (2) Agents are divided into groups composed of two agents under one condition, which is that we cannot put two accepted or two rejected agents in one group. In this example, group one consists of *A* and *B* and group two consists of *C* and *D* (note that if the number of agents is even, every agent has a partner. If the number of agents is odd, the last agent lacks a partner).

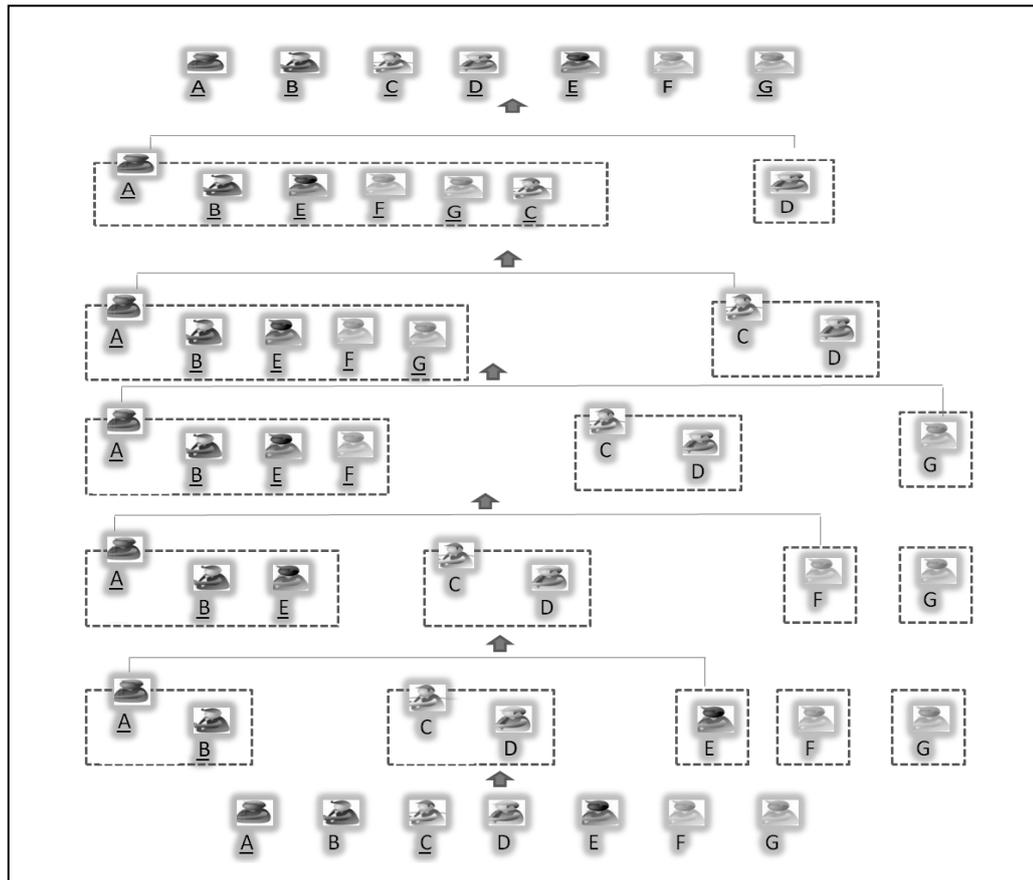


Figure 4.13: Example two of Dialogue Among N-agent.

- (3) Within each group, dialogues take place between two agents in order to reach the argument. In this example, agent *A* argues with agent *B* and agent *C* argues with agent *D*.
- (4) Within each group (whose members have the same opinion) the system will randomly select one agent to represent the beliefs of the group. In this example, since agents *A* and *B* accept the main topic, the system will select agent *A* to represent his group.
- (5) Agents are divided into groups composed of two agents under two conditions: 1) we cannot put two accepted or rejected agents in one group; 2) we cannot put the agents, who previously argued about the same topic and did not reach a decision, in one group. Group one now consists of *A*, *B* and *E* and group two consists of *C* and *F*.

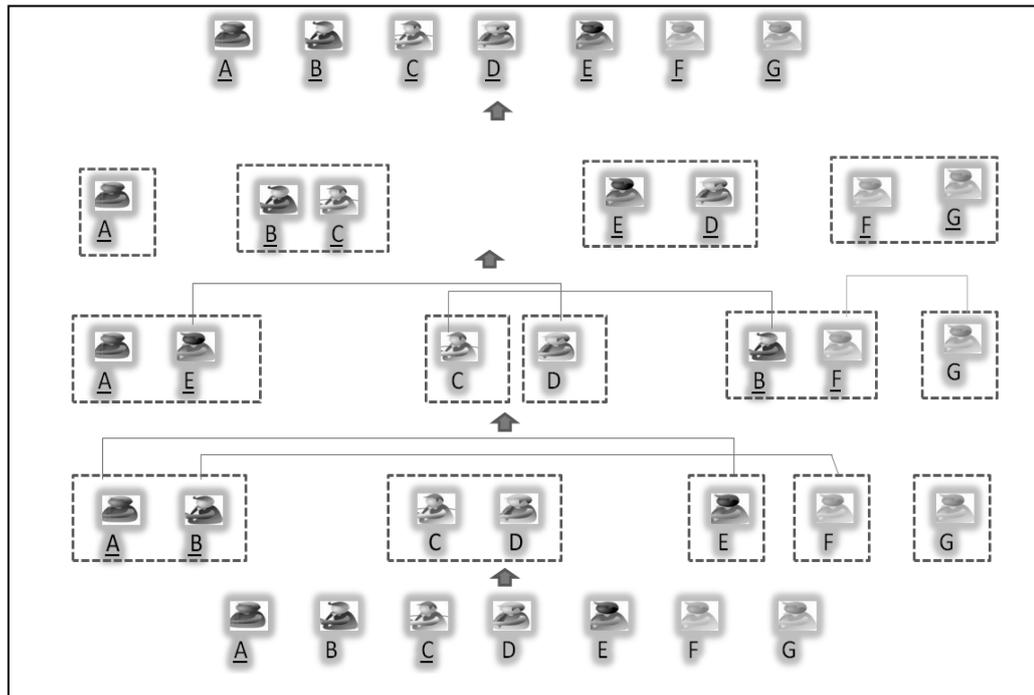


Figure 4.14: Example three of Dialogue Among N-agent.

- (6) The system reverts back to step 3 and repeats the same steps over and over again until agents reach an agreement. In this example, agent *A* argues with agent *E* and agent *C* argues with agent *F*.
- (7) *A* represents his group and *C* represents his group. Then, the groups become (*A*, *B*, *E*, *D*) and (*C*, *F*, *G*). Lastly, *A* argues with agent *D* and *C* argues with *G* and the agents reach a conclusion.

Figure 4.13 and Figure 4.14 illustrate different examples of dialogue among N-agent.

In Figure 4.13, the system divides agents into two groups: group one consists of *A* and *B* and group two consists of *C* and *D*. Then, in the second round, the system divides agents into two groups: group one consists of *A*, *B* and *E* and group two consists of *C* and *D*. After that, *F* becomes a member of group one in the third round and *G* becomes a member of group one in the fourth round. Finally, *A* persuades *C* and then *D*.

In Figure 4.14, the system divides agents into groups composed of two agents: group one consists of *A* and *B* and group two consists of *C* and *D*. Instead of selecting a representative for each group's belief, each agent reports accepting or rejecting the

main topic. Following this, the system divides agents into groups composed of two agents under the same condition (we cannot put two accepted or two rejected agents in one group): group one consists of *A* and *E*, group two consists of *C* and *D*, and group three consists of *B* and *F*. After that, the system divides agents into groups composed of two agents: group one consists of *B* and *C*, group two consists of *E* and *D*, and group three consists of *F* and *G*. Finally, the dialogue succeeds if all agents are persuaded.

As mentioned in the beginning of this section, the idea of dividing agents into groups composed of two agents under certain conditions is mentioned first in the Ito and Shintani's work [Ito and Shintani, 1996]. In their work they prove (using decision support system based on multi-agent negotiation) that this is a correct procedure that will always terminate and produces the correct results.

4.4.4 DID for N-agent

As mentioned in section 4.1, to represent an argument protocol in full, nine concepts are required (Locutions; Participants Commitment Store and Commitment rules; Structural rules; Turn Taking rules; Post-condition rules; Pre-condition rules; Locution types; and Sender and receiver agents roles). However, in N-agents' dialogue, we need to add more concepts:

- (1) Recursion rules (recursive-conditions and recursive-arguments): a set of rules which, when repeating them over the recursive arguments, can repeat the same task more than once until the recursive-condition cannot be achieved¹⁸. In N-agents' systems, an agent's role may need to recurse by sending the same locution to more than one agent (one-to-many system and one-to-all system) under some recursive-condition. These conditions are usually done over some recursive argument.

¹⁸ In agent protocol (e.g. LCC) recursion is accomplished by repeating the same process (or agent role) a specified number of times (the process or role calls itself) either to process a list or to loop it until the recursive condition fails.

- (2) Repeated locution: in the case of N-agent, more than one agent could use the same locution icon.

Therefore, we need to add an extra diagrammatic notation to the DID for N-agent, which represents recursion rules and repeated locution.

Figure 4.15 illustrates the locution icon for N-agent. A solid red rhombus represents a recursive-condition (which denotes applying the same part of the role definition more than once until it reaches a recursive-condition that fails). The red oval shape represents a recursive argument. The dotted, rounded-corner, rectangle box around the locution icon represents the recursive use of the locution by more than one agent.

Note that, the dividing agents condition, of the described method for dialogue among N agents in section 4.4.3, could be a pre-condition, post-condition or recursive-condition. Therefore, we must use either the solid red rhombus (where dividing rules = recursive-contentions) or the dotted rhombus (where dividing rules = pre- or post-conditions) to represent dividing agents condition.

Appendix B presents the DID for N-agent Formal Definition and a detail example of a persuasion dialogue among N-agent.

4.4.5 Problems and Solutions of DID for N-agent

As we can see from the Figure 4.15, in the case of DID for N-agent, the diagram may become too complex for the user to create, understand and edit. In other words, describing DID for N-agent in the diagrammatical way could be unpractical for the user for two primary reasons:

- (1) DID for N-agent overloads the diagrammatic notation with new arrows and symbols. These notations can confuse the user and make the overall task (drawing DID for N-agent) more difficult than writing the agent protocol by hand.

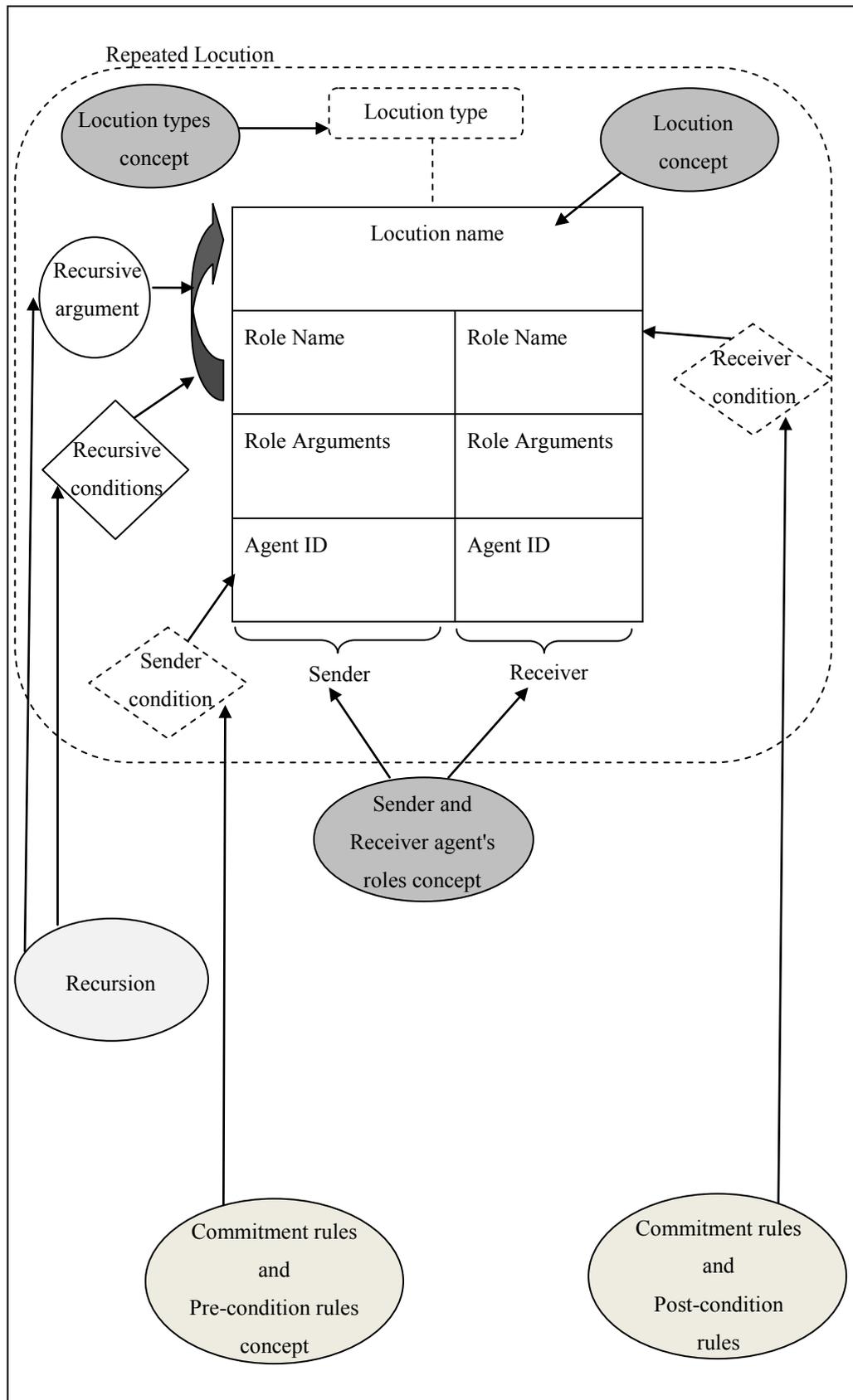


Figure 4.15: Locution Icon For N-agent.

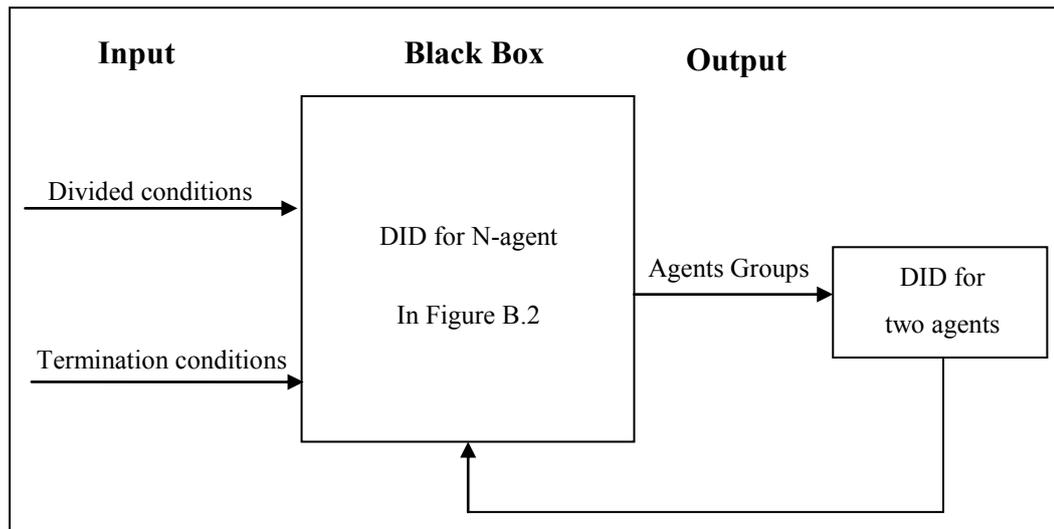


Figure 4.16: Black Box of DID for N-agent

- (2) Drawing DID for N-agent is complex since DID for N-agent is too close to agent protocol. The user needs to understand the notation of recursive, how to set up the constraint, and must learn how to write an agent protocol.

To solve this problem, we will hide the details of DID diagrams for N-agent in a black box (reusable diagram) and use parameters, which are transformational, to get the information needed, in order to do the protocol automated synthesis, from the user.

Essentially, we will get the divided conditions and termination conditions from the user. Then, the black box divides agents into groups composed of two agents under divided conditions and terminates the dialogue between N-agent when the termination conditions are satisfied (see Figure 4.16).

4.5 Summary

This chapter has presented a new recursive visual high-level language called DID between AIF (or other argumentation-based formalism) and multi-agent protocol languages (e.g. LCC). DID provides mechanisms to represent, in an abstract way, the dialogue game protocol rules by giving an overview of the permitted moves and their relationship to each. It can model any interaction between two agents (unique-moves

and immediate-reply protocol) that can be described as a sequence of recursive steps terminating in a base case.

DID explains the order and type of messages that two or more agents can interchange and the rules of the message interchange. However, a DID cannot explain how two or more agents can cooperate and interact with each other in situations where more complex protocols involving more than turn-taking are required.

In practice, the DID language provides the first step to get from the user the missing agent protocol concepts. In chapter 5, we will present the next step which allows us to get the missing development language concepts and perform the automated synthesis of multi-agent protocol.

Chapter 5

Synthesis of Concrete Protocols

As mentioned in the previous chapter, to fully generate via automatic synthesis the agent protocols from any AIF description we need to obtain missing concepts (information) from both the user and the development language. The previous chapter provides a detailed description on how to obtain these missing agent protocol concepts from the user, by using the DID language. DID explains the order and type of messages that two or more agents can interchange and the rules of the message interchange. However, it does not explain how two or more agents can cooperate and interact with each other because it omits essential concepts related to the dynamics of interaction between agents.

This chapter proposes a mechanism on how to obtain the missing concepts from the development language as well as to provide a fully automated synthesis method to generate argumentation agent protocols from DID. In practice, when dealing with the agent interaction protocol synthesis and the development of an agent protocol, common codes and relations can be found. These codes can be specified as design patterns, which are independent from any particular protocol specification problem and can recur repeatedly across protocols. In this chapter, we put forward some protocol design patterns that can be embedded in the automated synthesis tools and used with DID to support agent protocol development activity. The reason for introducing protocol design patterns in argumentation is that by re-using them it is possible to reduce the effort of building argumentation agent protocols.

We open this chapter with a description of LCC-Argument protocol design patterns in Section 5.1. Section 5.2 presents the automated synthesis steps for generating agent protocols between two-agents and N-agent automatically. Finally, section 5.3 presents a summary of the LCC-Argument protocol design patterns and the automated synthesis method.

5.1 LCC-Argument Patterns

By taking a closer look at the LCC protocol in chapter 2, we can see that this protocol is quite complex, and therefore requires us to consider issues that the software engineer may not be aware of until later in the implementation process, such as synchronisation of the role. To overcome this problem, we supply LCC-Argument patterns, which are re-usable, parameterisable LCC specifications that can be embedded in automated synthesis tools and used with DID to support agent protocol development. This allows us to reduce the effort of building more complex argumentation protocols by re-using design patterns repeatedly to generate argumentation protocols (see chapter 2 for more information about design pattern). The set of these more complex design patterns is, in theory, unbounded (for the same reason that design patterns in traditional software engineering are unbounded) but in practice families of interaction patterns occur.

In fact, LCC-Argument patterns capture the different relationships and interactions between LCC agents' roles. These patterns provide common LCC argument code for developing protocols and their components along with explaining how two or more agents can interact with each other. They are generic solutions to the common LCC argumentation protocol development problem that recur across protocols repeatedly and can be adapted to generate specific protocols.

To explain LCC-Argument patterns, we will use the following seven generic characterisations (adapted from Appleton, Taylor and Wray works [Appleton,1998; Taylor and Wray, 2004] to suit the needs of our argumentation domain):

- (1) *Name*: a meaningful unique name which could be used to refer to the pattern's knowledge and structure;
- (2) *Problem*: a statement or a question that relates to the problem which describes the problem that the pattern solves;

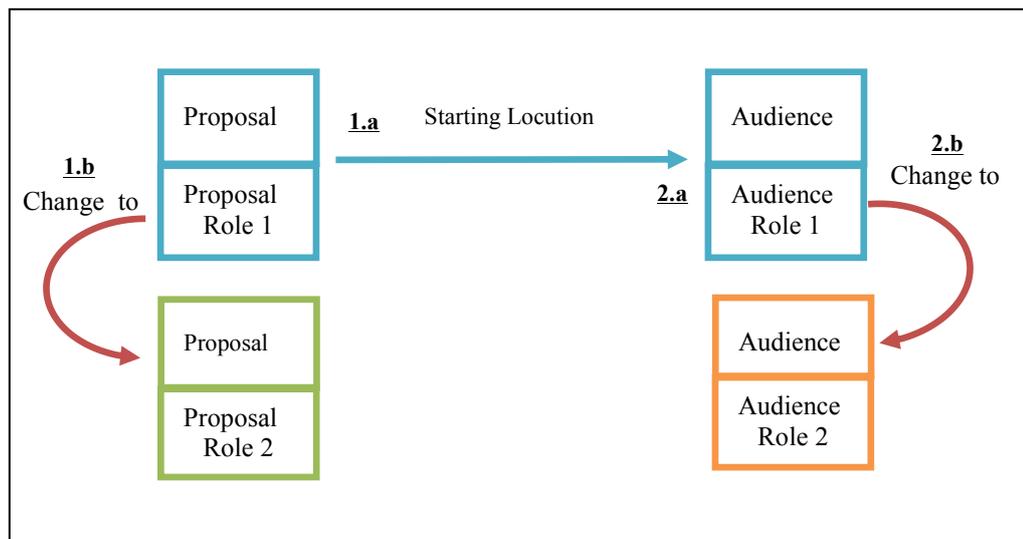
- (3) *Solution*: relationship between the pattern's roles, which describes how the problem is solved, often including a diagram that describes how the problem is solved;
- (4) *Context (Pre-conditions)*: the initial configuration of the protocol before the pattern is applied;
- (5) *Consequence (Post-conditions)*: the configuration of the protocol after the pattern has been applied;
- (6) *Structure*: identifies the pattern's structure, its roles and their relationship to each other;
- (7) *Rewriting methods*: a set of rewriting rules based on the semantics of LCC, which allow generic relationship between roles to be rewritten in a specific way (Note that, there might be a direct, complex or indirect relation between roles).

Pattern1:

Name: Starting pattern (SP).

Problem: How to start an argument (dialogue).

Solution:



Both agents send/receive a message (locution) and then change their roles to remain in the dialogue:

- (1) Proposal (speaker) agent proposes an action (start a dialogue) by sending a starting locution (step 1.a) and then changes its role (step 1.b).
- (2) Audience agent receives a starting locution (step 2.a) and then changes its role (step 2.b)

Context (Pre-conditions): Use a Starting Pattern when a proposal agent has not started a dialogue.

Consequence (Post-conditions):

- (1) Both the proposal and audience agents engage in a dialogue.
- (2) Both the proposal and audience agents change their roles to remain in the dialogue.

Structure:

```

a(RP1(KBP,CSP, Topic, IDA),IDP) ::=
    SL(Topic) => a(RA1(KBA,CSA,IDP),IDA) ← C1
    then
    a(RP2(KBP,CSP, Topic, IDA),IDP).
a(RA1(KBA,CSA,IDP),IDA) ::=
    C2 ← SL(Topic) <= a(RP1(KBP,CSP, Topic, IDA),IDP)
    then
    a(RA2(KBA,CSA, Topic ,IDP),IDA)
    
```

Where SL represents the *Starting Locution* and CI represents a condition that must be satisfied in order for a proposal agent ID_P to send the *Starting Locution* SL . Usually, CI is a condition over Topic. $C2$ represents a condition that must be satisfied after audience agent ID_A receives the starting locution.

In this LCC code, there are two roles: R_{P1} and R_{A1} . The R_{P1} role of the proposal agent ID_P has four input parameters: (1) KB_P which represents the agent knowledge base list (the propositions that the agent believes); (2) CS_P which represents the agent commitment store list (a set of propositions to which the player is committed in the discussion). Note that CS_P is initially empty, since R_{P1} represents the first role of the proposal agent in the LCC protocol; (3) *Topic* to open dialogue; (4) ID_A which represents the audience agent identifier. The R_{P1} role begins by checking the CI condition. If the CI condition is true, then the R_{P1} role sends a *Starting Locution* SL to the R_{A1} role and then it changes its role to the R_{P2} .

The R_{A1} role of audience agent ID_A has three input parameters: (1) KB_A which represents the agent knowledge base list; (2) CS_A which represents the agent commitment store list. Note that CS_A is initially empty, since R_{A1} represents the first role of the audience agent in the LCC protocol; (3) ID_P which represents the proposal agent identifier. The R_{A1} role begins by receiving a *Starting Locution* SL from R_{P1} . Then, the R_{A1} role satisfies $C2$ and then it changes its role to the R_{A2} .

Rewriting methods: none

Pattern 2:

Name: Termination-Intermediate Pattern (TIP).

Problem: How to recur or terminate an argument (dialogue) between two agents.

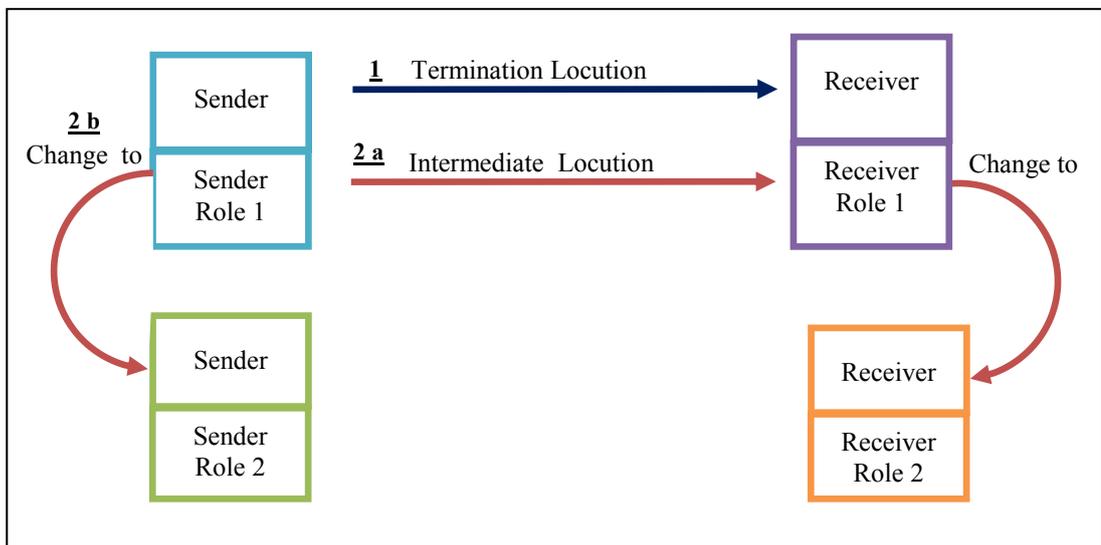
Solution: Both agents send/receive a message(s) (locution) to terminate the dialogue or to change role.

(1) Dialogue Termination (Termination locution):

- First agent (sender) sends a locution to terminate the argument.
- Second agent (receiver) receives a locution, which states the sender's intention to terminate the argument.

(2) Changing role (Intermediate locution):

- First agent (sender) sends a permitted locution (step 2.a) and then changes its role (step 2.b).
- Second agent (receiver) receives a permitted locution and then changes its role.



Context (Pre-conditions): Use a Termination-Intermediate Pattern when the dialogue between the proposal agent and audience agent has already started.

Consequence (Post-conditions):

(1) Dialogue Termination (Termination locution):

- The dialogue between the proposal and audience agents is terminated.

(2) Changing role (Intermediate locution):

- Both the sender and receiver agents change their roles to remain in dialogue.

Structure:

$$\begin{array}{l}
 \mathbf{a}(R_{\text{Sender}1}(\mathbf{KB}_{\text{Sender}}, \mathbf{CS}_{\text{Sender}}, \mathbf{CS}_{\text{Receiver}}, \mathbf{Topic}, \mathbf{ID}_{\text{Receiver}}), \mathbf{ID}_{\text{Sender}}) ::= \\
 R_{\text{Sender}1} \xrightarrow{TL} R_{\text{Receiver}1} \\
 \text{or} \\
 R_{\text{Sender}1} \xrightarrow{IL} R_{\text{Receiver}1} \\
 \\
 \mathbf{a}(R_{\text{Receiver}1}(\mathbf{KB}_{\text{Receiver}}, \mathbf{CS}_{\text{Receiver}}, \mathbf{CS}_{\text{Sender}}, \mathbf{Topic}, \mathbf{ID}_{\text{Sender}}), \mathbf{ID}_{\text{Receiver}}) ::= \\
 R_{\text{Receiver}1} \xleftarrow{TL} R_{\text{Sender}1} \\
 \text{or} \\
 R_{\text{Receiver}1} \xleftarrow{IL} R_{\text{Sender}1}
 \end{array}$$

This pattern represents a generic recursive clause. The variable R in the definition above represents the role name. KB and CS are the role arguments and ID is the agent identifier. TL represents Termination Locution and IL represents an Intermediate Locution. ' $\xrightarrow{\quad}$ ' represents outgoing messages from a role, and ' $\xleftarrow{\quad}$ ' represents incoming messages.

In this LCC pattern, there are two roles: $R_{\text{Sender}1}$ and $R_{\text{Receiver}1}$. The $R_{\text{Sender}1}$ role of sender agent ID_{Sender} has five input parameters: (1) KB_{Sender} which represents the agent knowledge base list; (2) CS_{Sender} which represents the agent commitment store list; (3) CS_{Receiver} which represents the receiver agent commitment store list; (4) $Topic$ to open the dialogue; (5) ID_{Receiver} which represents the receiver agent identifier. The $R_{\text{Sender}1}$ role begins by sending either a *Termination Locution* TL to the $R_{\text{Receiver}1}$ role or an *Intermediate Locution* IL . The ' $\xrightarrow{\quad}$ ' symbol indicates that the $R_{\text{Sender}1}$ role may send one or more different TL s (or IL s) to the $R_{\text{Receiver}1}$ role.

The $R_{\text{Receiver}1}$ role of the receiver agent ID_{Receiver} has five input parameters: (1) KB_{Receiver} which represents the agent knowledge base list; (2) CS_{Receiver} which represents the agent commitment store list; (3) CS_{Sender} which represents the sender agent commitment store list; (4) $Topic$ to open the dialogue; (5) ID_{Sender} which represents the sender agent identifier. The $R_{\text{Receiver}1}$ role begins by receiving either a *Termination Locution* TL from the $R_{\text{Sender}1}$ role or an *Intermediate Locution* IL . The ' $\xleftarrow{\quad}$ ' symbol indicates that the $R_{\text{Receiver}1}$ role may receive one or more different TL s (or IL s) from the $R_{\text{Sender}1}$ role.

Rewriting methods:

First (Sending Termination Method): Rewriting of the " $R_{Sender1} \xrightarrow{TL} R_{Receiver1}$ "

If there is a general relation of " $R_{Sender1} \xrightarrow{TL} R_{Receiver1}$ " then it is possible to specialise it within two different statements:

Rewrite 1: (one termination locution)

We might specialise " $R_{Sender1} \xrightarrow{TL} R_{Receiver1}$ " to an interaction statement that sends a $TL(Topic)$ termination message to agent $ID_{Receiver}$, which is achieved by the constraint $C1$. In practice, $C1$ may represent more than one condition that is connected by *or* and *and* operators. Usually, $C1$ is a condition over the role arguments (e.g. KB and CS).

$$\begin{aligned} & TL(Topic) \Rightarrow a(R_{Receiver1}(KB_{Receiver}, CS_{Receiver}, CS_{Sender}, Topic, ID_{Sender}), ID_{Receiver}) \\ & \leftarrow C1 \end{aligned}$$

Rewrite 2:(multiple termination locution)

We might specialise " $R_{Sender1} \xrightarrow{TL} R_{Receiver1}$ " to an interaction statement that sends a $TL(Topic)$ termination message to agent ID_P which is achieved by the constraint $C1$. Then, there is another termination relation between $R_{Sender1}$ and $R_{Receiver1}$.

$$\begin{aligned} & TL(Topic) \Rightarrow a(R_{Receiver1}(KB_{Receiver}, CS_{Receiver}, CS_{Sender}, Topic, ID_{Sender}), ID_{Receiver}) \\ & \leftarrow C1 \\ & \text{or} \\ & R_{Sender1} \xrightarrow{TL} R_{Receiver1} \end{aligned}$$

Second (Receiving Termination Method): Rewriting of the " $R_{Receiver1} \xleftarrow{TL} R_{Sender1}$ "

If there is a general relation of " $R_{Receiver1} \xleftarrow{TL} R_{Sender1}$ " then it is possible to specialise it within two different statements:

Rewrite 1: (one termination locution)

We might specialise " $R_{Receiver1} \xleftarrow{TL} R_{Sender1}$ " to an interaction statement that receives a $TL(Topic)$ termination message from agent ID_{Sender} . $C2$ represents a condition that

must be satisfied after receiver agent receives the *Termination Locution TL*. In practice, *C2* may represent more than one condition that is connected by *or* and *and* operators. Usually, *C2* is a condition over the role arguments (e.g. *KB* and *CS*).

$$C2 \leftarrow TL(\text{Topic})$$

$$\leq a(R_{\text{Sender1}}(\text{KB}_{\text{Sender}}, \text{CS}_{\text{Sender}}, \text{CS}_{\text{Receiver}}, \text{Topic}, \text{ID}_{\text{Receiver}}), \text{ID}_{\text{Sender}})$$

Rewrite 2:(multiple termination locution)

We might specialise " $R_{\text{Receiver1}} \lesssim^{TL} R_{\text{Sender1}}$ " to an interaction statement that receives a $TL(\text{Topic})$ Termination message from agent $\text{ID}_{\text{Sender}}$. *C2* represents a condition that must be satisfied after receiver agent receives the *Termination Locution TL*. Then, there is another termination relation between R_{Sender1} and $R_{\text{Receiver1}}$.

$$C2 \leftarrow TL(\text{Topic})$$

$$\leq a(R_{\text{Sender1}}(\text{KB}_{\text{Sender}}, \text{CS}_{\text{Sender}}, \text{CS}_{\text{Receiver}}, \text{Topic}, \text{ID}_{\text{Receiver}}), \text{ID}_{\text{Sender}})$$

or

$$R_{\text{Receiver1}} \lesssim^{TL} R_{\text{Sender1}}$$

Third(Sending Intermediate method): Rewriting of " $R_{\text{Sender1}} \overset{IL}{\approx} R_{\text{Receiver1}}$ "

Rewrite 1: (One intermediate locution)

We might specialise " $R_{\text{Sender1}} \overset{IL}{\approx} R_{\text{Receiver1}}$ " to an interaction statement that sends message $IL(\text{Topic})$ to agent $\text{ID}_{\text{Receiver}}$ which is achieved by the constraint *C3*. Following this, it changes its role. In practice, *C3* may represent more than one condition, which is connected by *or* and *and* operators. Usually, *C3* is a condition over the role arguments (e.g. *KB* and *CS*).

$$IL(\text{Topic}) \Rightarrow a(R_{\text{Receiver1}}(\text{KB}_{\text{Receiver}}, \text{CS}_{\text{Receiver}}, \text{CS}_{\text{Sender}}, \text{Topic}, \text{ID}_{\text{Sender}}), \text{ID}_{\text{Receiver}})$$

$$\leftarrow C3$$

then

$$a(R_{\text{Sender2}}(\text{KB}_{\text{Sender}}, \text{CS}_{\text{Sender}}, \text{CS}_{\text{Receiver}}, \text{Topic}, \text{ID}_{\text{Receiver}}), \text{ID}_{\text{Sender}})$$

Rewrite 2: (multiple Intermediate locutions):

We might specialize " $R_{Sender1} \stackrel{IL}{\approx} R_{Receiver1}$ " to an interaction statement that sends message $IL(Topic)$ to agent $ID_{Receiver}$, which is achieved by the constraint $C3$, after that it recurses. Then, there is then another recursive relation between $R_{Sender1}$ and $R_{Receiver1}$.

$$\begin{array}{l}
 \text{IL(Topic)} \Rightarrow a(R_{Receiver1}(KB_{Receiver}, CS_{Receiver}, CS_{Sender}, Topic, ID_{Sender}), ID_{Receiver}) \\
 \leftarrow C3 \\
 \text{then} \\
 \quad a(R_{Sender2}(KB_{Sender}, CS_{Sender}, CS_{Receiver}, Topic, ID_{Receiver}), ID_{Sender}) \\
 \\
 \text{or} \\
 R_{Sender1} \stackrel{IL2}{\approx} R_{Receiver1}
 \end{array}$$

Fourth(Receiving Intermediate method): Rewriting of " $R_{Receiver1} \stackrel{IL}{\approx} R_{Sender1}$ "
Rewrite 1: (One intermediate locution)

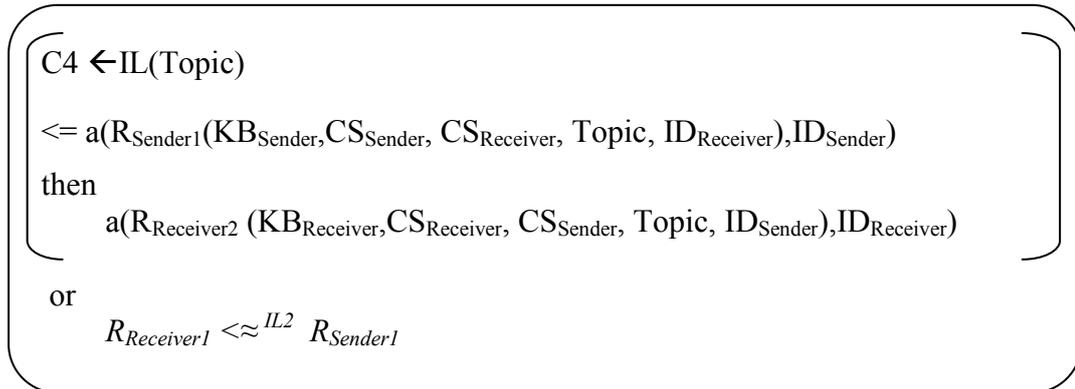
We might specialise " $R_{Receiver1} \stackrel{IL}{\approx} R_{Sender1}$ " to an interaction statement that receives message $IL(Topic)$ from agent ID_{Sender} . Following this, it changes its role. $C3$ represents a condition that must be satisfied after receiver agent receives the *Intermediate Locution IL*. In practice, $C4$ may represent more than one condition, which is connected by *or* and *and* operators. Usually, $C4$ is a condition over the role arguments (e.g. KB and CS).

$$\begin{array}{l}
 C4 \leftarrow IL(Topic) \\
 \leq a(R_{Sender1}(KB_{Sender}, CS_{Sender}, CS_{Receiver}, Topic, ID_{Receiver}), ID_{Sender}) \\
 \text{then} \\
 \quad a(R_{Receiver2}(KB_{Receiver}, CS_{Receiver}, CS_{Sender}, Topic, ID_{Sender}), ID_{Receiver})
 \end{array}$$

Rewrite 2: (multiple Intermediate locutions):

We might specialize " $R_{Receiver1} \stackrel{IL}{\approx} R_{Sender1}$ " to an interaction statement that receives message $IL(Topic)$ from agent ID_{Sender} , after that it recurses. Then, there is then another Recursive relation between $R_{Sender1}$ and $R_{Receiver1}$. $C4$ represents a

condition that must be satisfied after receiver agent receives the *Intermediate Locution IL*.



Pattern3:

Name: Broadcasting Pattern (BP)

Problem: use this pattern to solve four problems at the same time:

- (1) How to start an argument (dialogue) for $N \geq 3$ agents, or how to broadcast new Topic to $N \geq 3$ agents;
- (2) How to respond to the broadcasting;
- (3) How to divide agents into groups of two;
- (4) How to terminate an argument (dialogue) for $N \geq 3$ agents.

Solution:

- (1) Step one (Start a Dialogue or Broadcast a Topic): (see Figure 5.1)
 - a) Proposal agent proposes an action (start dialogue) by sending a *proposal(Topic)* locution to all agents (step a.1) and then changes its role to *replyToProposalReceiver_{proposal}* (step a.2).
 - b) Other agents (all agents except the proposal agent) receive a *proposal(Topic)* locution (step b.1) and then change their role to *replyToProposalSender* (step b.2).

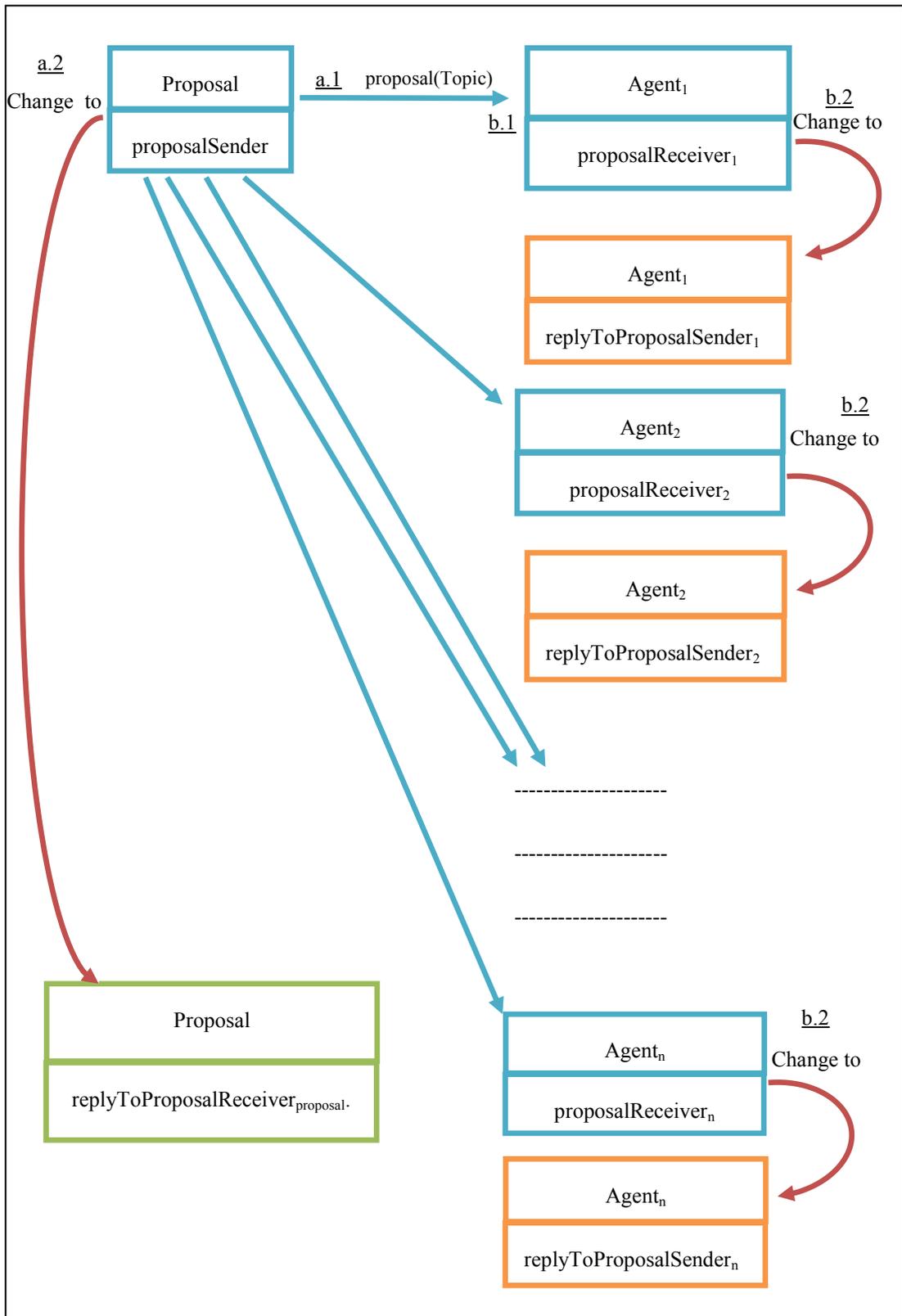


Figure 5.1: Broadcasting Pattern Solution (Step one)

(2) Step two (Respond to the Broadcasting): (see Figure 5.2)

- a) Other agents send either an *accept*(Topic) or *reject*(Topic) locution to the proposal agent (step a.1) and then changes their role to *replyToProposalSender* (step a.2).
- b) The proposal agent receives either an *accept*(Topic) or *reject*(Topic) locution (step b.1) and then changes its role to *replyToProposalSender* (step b.2).

(3) Step three (Divide or Terminate):

a) Divide: (see Figure 5.3)

- i. The proposal agent sends *argueWith*(Topic,Agent_P,Agent_O) location for a pair of agents (step i.1): Agent_P and Agent_O (telling them to interact together) and then recurses (step i.2) or changes its role (step i.3).
- ii. Both Agent_P and Agent_O receive *argueWith*(Topic,Agent_P,Agent_O) location (step ii.1) and then change their roles to *startDID* role (step ii.2).

b) Terminate: (see Figure 5.4)

- i. The proposal agent sends *reachAgreement*(Topic) location to all other agents (step i.1) and then terminates its role (step i.2).
- ii. All other agents receive *reachAgreement*(Topic) (step ii.1) and then terminate their roles (step ii.2).

Context (Pre-conditions):

Use the Broadcasting Pattern when a proposal agent has not already started a dialogue for $N \geq 3$ agents.

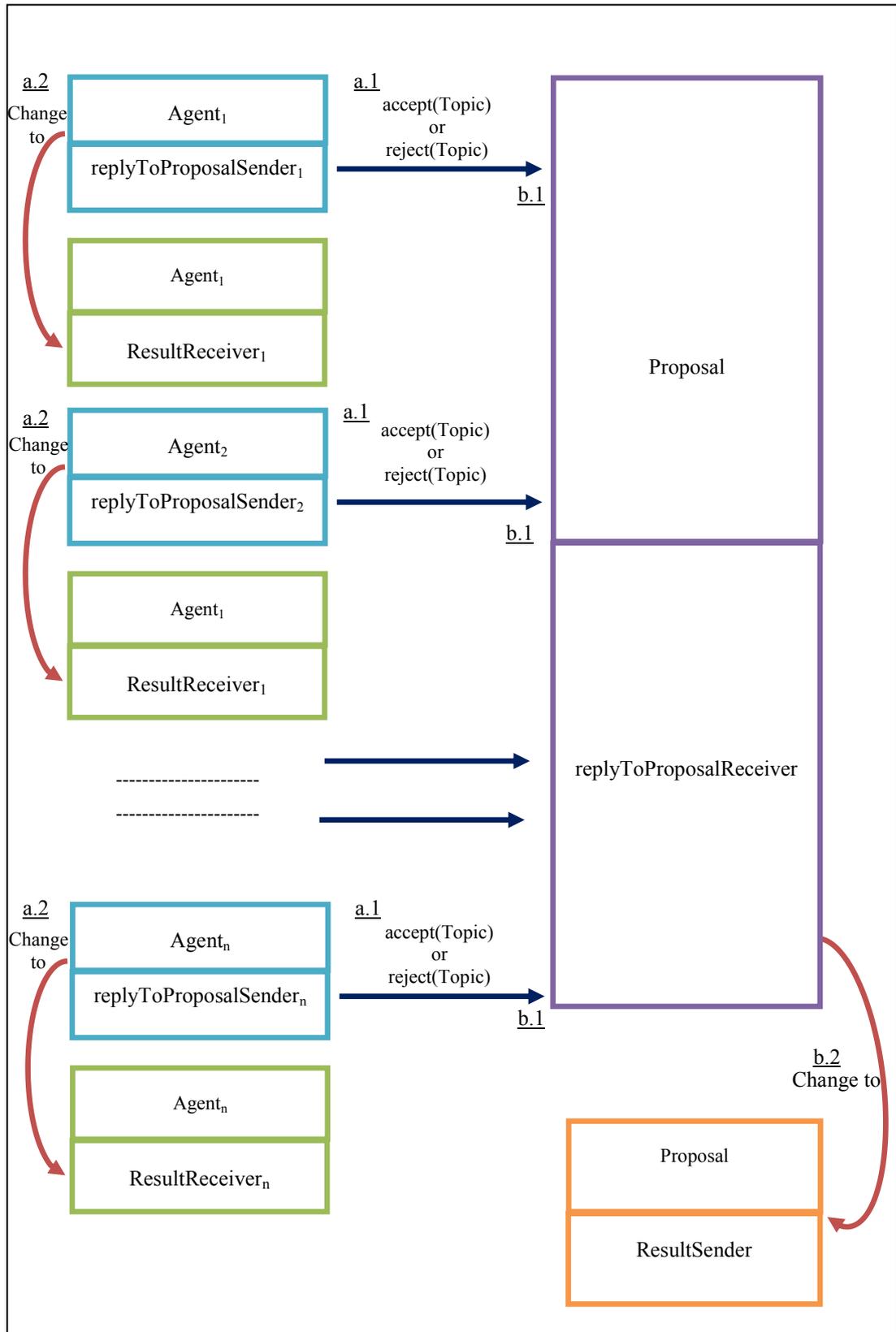


Figure 5.2: Broadcasting Pattern Solution (Step two)

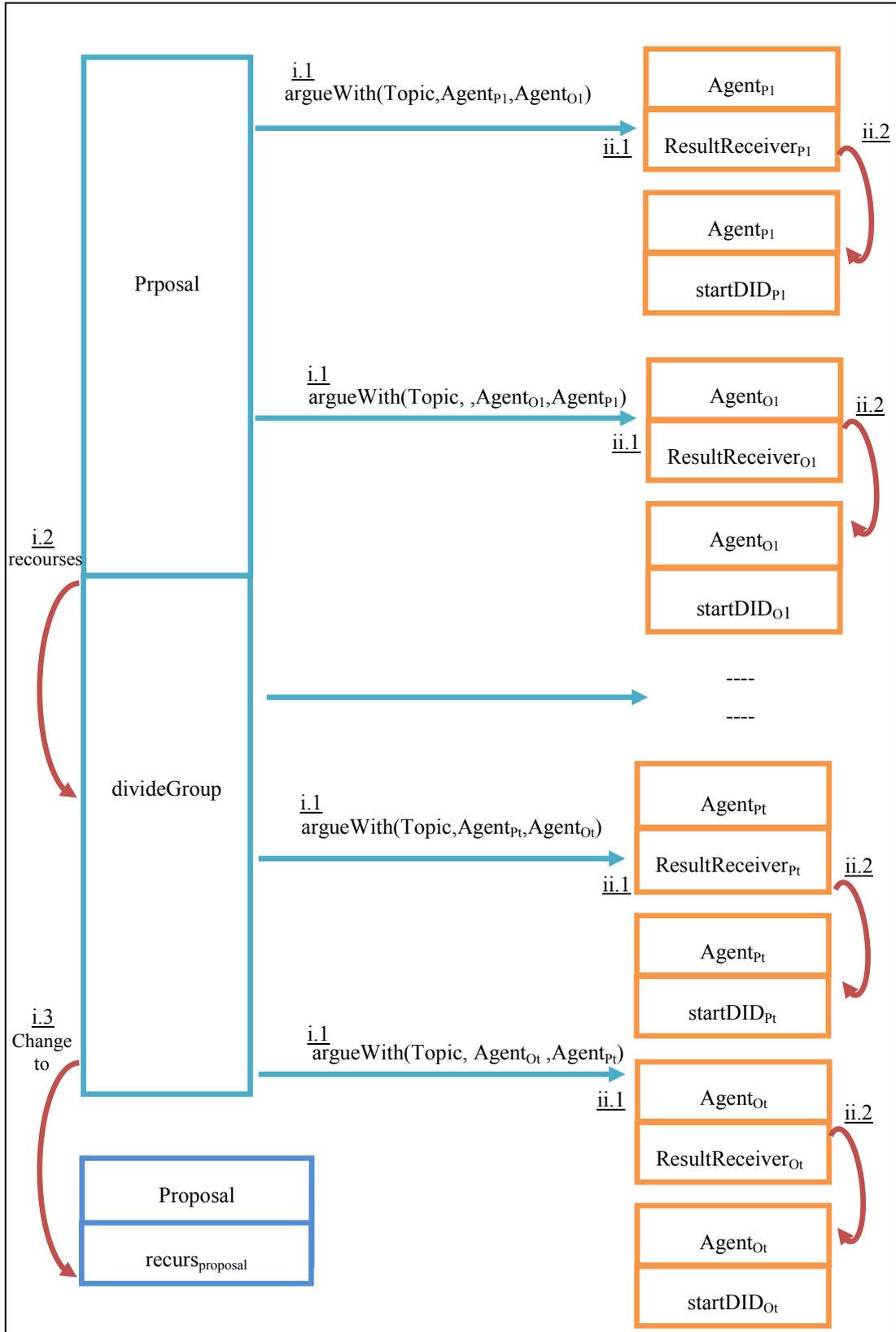


Figure 5.3: Broadcasting Pattern Solution (Step three:Divide)

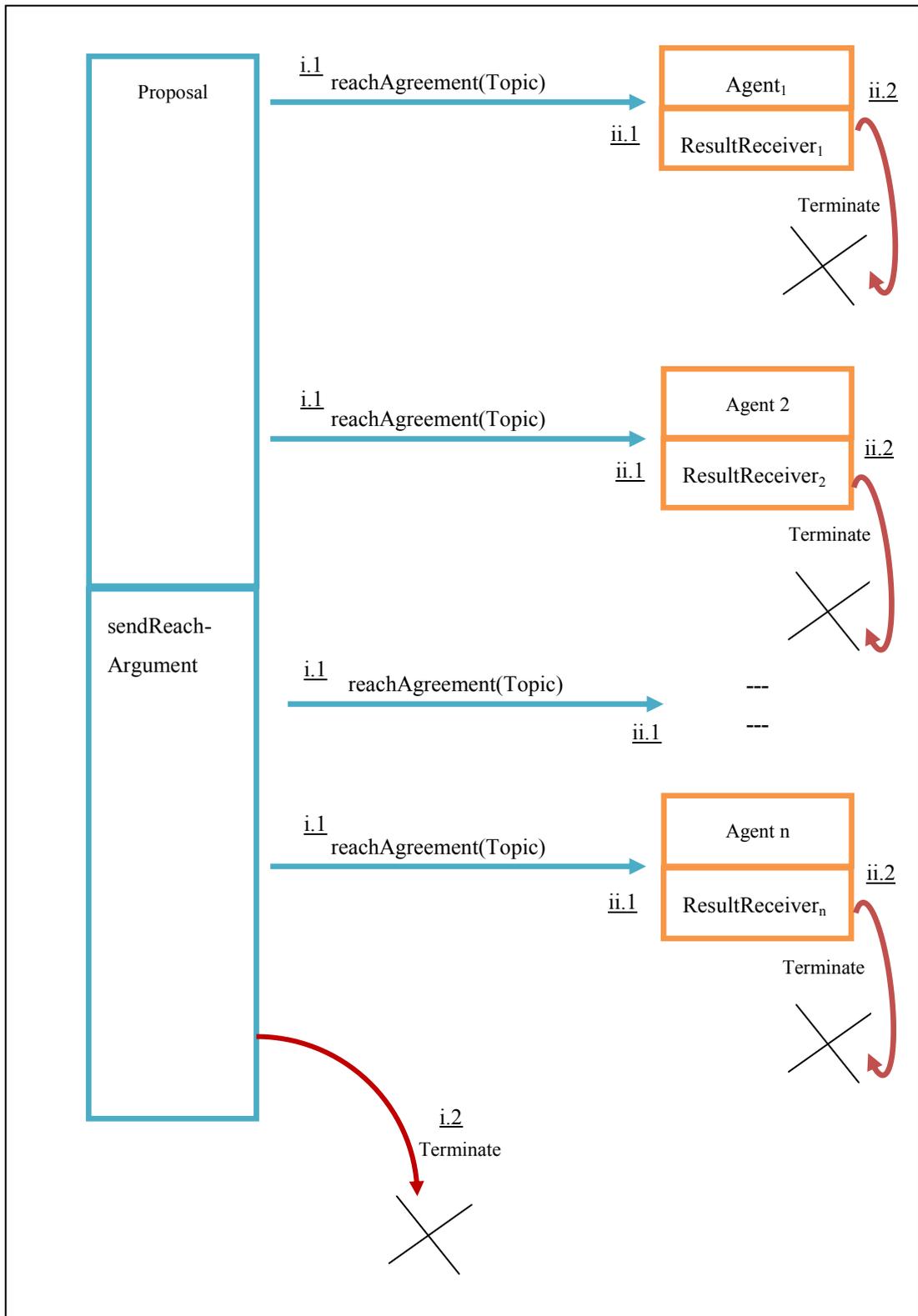


Figure 5.4: Broadcasting Pattern Solution (Step three:Termination)

Consequence (Post-conditions):

(1) Step one (Start a Dialogue or Broadcast a Topic):

- Proposal and other agents engaged in a dialogue.
- Proposal agent committed to $\text{Topic} \in \text{CS}_{\text{Proposal}}$ (updates its commitment store by adding the Topic to it).
- Proposal and all other agents (receivers) change their roles so as to remain in dialogue.

(2) Step two (Respond to the Broadcasting):

Both sender and receiver agents change their roles so as to remain in dialogue.

(3) Step three (Divide or Terminate):

- Divide: Divide agents into groups of two and start dialogues between two agents.
- Terminate: The dialogue between N-agent is terminated.

Structure:

Broadcasting Pattern contains 8 roles:

- Two roles to solve the first problem (How to start an arguments (dialogue) for $N \geq 3$ agents, or how to broadcast new Topic to $N \geq 3$ agents) (see Figure 5.5):

(1) proposalSender_{proposal} (2) proposalReceiver_{ID}

- Two roles to solve the second problem (How to respond to the broadcasting?) (see Figure 5.6):

(1) ReplyToProposalSender_{ID} (2) replyToProposalReceiver_{proposal}

```

a(proposalSenderproposal(AgentList,NAgent,NSupporters,Topic),IDproposal))::=
proposal(Topic) => a(proposalReceiverID(KBID,CSID,IDproposal), ID)
← getAgentIDFromList (AgentList,otherAgents,ID) and addTopicToCS(Topic,CSproposal)

then

(
a(replyToProposalReceiverproposal (AgentList, NAgent,NSupporters,Topic,0,[ ],[ ],0,0), IDproposal)
← agentListEmpty(AgentList)

or

a(proposalSenderproposal (OtherAgents,NAgent,NSupporters,Topic), IDproposal)
).

a(proposalReceiverID(KBID,CSID,IDproposal), ID)::=
proposal(Topic)<=a(proposalSenderproposal(AgentList,NAgent,NSupporters,Topic), IDproposal)

then

a(replyToProposalSender(KBID,CSID, Topic,IDproposal), ID).

```

Figure 5.5 : Structure (*proposalSender_{proposal}* and *proposalReceiver_{receiver}* roles)

- Four roles to solve the third and fourth problems (How to divide agents into groups of two, and how to terminate an argument (dialogue) for $N \geq 3$ agents) (see Figure 5.7):

(1) *resultSender_{proposal}* (2) *sendReachAgreement_{Proposal}*
(3) *divideGroup_{Proposal}* (4) *resultReceiver_{ID}*

Where *DivideC2* represents a condition that must be satisfied in order for a proposal agent to divide agents into groups composed of two agents. By default, *DivideC2* is "*lessThan(NAccepting,NSupporters) and isEmpty(RejectionList) and isEmpty(AcceptingList)*". *TerminationC1* represents a condition that must be satisfied in order for a proposal agent to terminate the dialogue between N-agent. By default, *TerminationC1* is "*greaterThanOrEqual(NAccepting,NSupporters)*" a function which returns true if *NAccepting* is greater than or equal to *NSupporters*. *AgentGroupC3* represents a function which divides agents into groups composed of two agents.

```

a(replyToProposalSenderID(KBID,CSID,Topic,IDproposal), ID) ::=
(
    accept(Topic) => a(replyToProposalReceiverproposal(_,_,_,_,_,_,_,_,_),IDproposal)

    ← findTopicInKB(Topic, KBID) and notFindTopicInCS (Topic,CSID) and
      notFindOppTopicInCS (not(Topic),CSID) and addTopicToCS(Topic,CSID)

    or

    reject(Topic) => a(replyToProposalReceiverproposal ( _,_,_,_,_,_,_,_,_),IDproposal)

    ← notFindTopicInKB(Topic,KBProposal) and notFindTopicInCS(Topic,CSProposal)
)

then

a(resultReceiverID(KBID,CSID,Topic,IDproposal), ID) .

a(replyToProposalReceiverproposal(AgentList,NAgent,NSupporters,Topic,NReply,
AcceptingList,RejectingList,NAccepting,NRejecting), IDproposal) ::=
(

    addIDToList(SendingList,OtherSedingList,ID) and
    addToAcceptingList(AcceptingList,AccList,ID) and
    increaseAccepting(NAccepting,NAcc) and
    RejList= RejectionList and
    NRej is NRejection
    ← accept(Topic) <= a(replyToProposalSenderID(KBID,CSID,Topic,IDproposal), ID)

    or

    addToRejectingList(RejectingList,RejList,ID) and
    increaseRejecting(NRejecting,NRej) and
    increaseReply (NReply,NRep) and
    AccList=AcceptingList and
    NAcc is NAccepting
    ← reject(Topic) <= a(replyToProposalSenderID( KBID,CSID,Topic,IDproposal), ID)
)

then

a(resultSenderproposal ( AgentList,NAgent, NSupporters,Topic,NReply,AcceptingList,
    RejectionList,NAccepting,NRejection ), IDproposal) ← isEqual(NRep,NAgent).
    
```

Figure 5.6: Structure (*replyToProposalSender* and *replyToPrposalReceiver_{proposal}* roles)

```

a(resultSenderproposal(AgentList,NAgent,
NSupporters,Topic,AcceptingList,RejectionList, AgentGroup), IDproposal) ::=

a(sendReachAgreementproposal (AgentList,NAgent,Topic),IDproposal)
← TerminationC1

or

a(divideGroupproposal (AgentList , NAgent,NSupporters ,Topic,AcceptingList,RejectionList,
[ ]),IDproposal) ← DivideC2.

a(sendReachAgreementProposal (AgentList, Topic ),IDProposal) ::=

reachAgreement(Topic) => a(resultReceiverID(KBID,CSID,IDproposal), ID)
← getAgentIDFromList (AgentList,otherAgents,ID)

then
(
null ← isAgentListEmpty(AgentList)
or
a(sendReachAgreementproposal (OtherAgents, Topic), IDproposal)
).

a(divideGroupProposal (AgentList, NAgent,NSupporters ,Topic,
AcceptingList,RejectionList,AgentGroup), IDproposal )::=
(
argueWith (Topic,P,O) => a(resultReceiverP(KBP,CSP,Topic,IDproposal), P)
← AgentGroupC3
then
argueWith (Topic,O,P) => a(resultReceiverO(KBO,CSO,Topic,IDproposal), O)
)

then
(
a(recursproposal (AgentList, NAgent,NSupporters ,0 ,Topic),IDproposal)
← RecursC4
or
a(divideGroupproposal(AgentList ,NAgent,NSupporters,Topic,Ac,Re,AGroup ),IDProposal)
)

a(resultReceiverP(KBP,CSP,CSO,Topic,IDproposal),P) ::=

reachAgreement(Topic) <= a(sendReachAgreementproposal (AgentList, Topic ),IDProposal)
or
(
argueWith(Topic,P,O) <=
a(divideGroupProposal(AgentList,NAgent,NSupporters,Topic,AcceptingList,RejectionList,
AgentGroup),IDproposal)
then
a(startDID(KBP,CSP, CSO,Topic, IDproposal, O),P)
).
    
```

Figure 5.7: Structure (*resultSender_{proposal}* , *sendReachAgreement_{Proposal}*, *divideGroup_{Proposal}* and *resultReceiver* roles)

RekursC4 represents a condition that must be satisfied in order for a proposal agent to recur. By default, *RekursC4* is "*isListEmpty(Re) or isListEmpty(Ac)*" which returns true if Re (or Ac) list is empty list.

The meaning of each role argument is shown in Table 5.1. The meaning of each function is shown in Table 5.2(a) and Table 5.2(b).

In this LCC pattern (Figure 5.5, Figure 5.6 and Figure 5.7), the *proposalSender_{Proposal}* role of proposal agent *ID_{Proposal}* has four input parameters: *AgentList*, *NAgent*, *NSupporters* and *Topic*. The *proposalSender_{Proposal}* role begins by sending the *proposal(Topic)* message to one agent (at the head of the *AgentList* list) and then if the *AgentList* list is empty, the proposal agent changes its role to *replyToProposalReceiver_{Proposal}* role, otherwise, it recurses over the remaining agents (recurses over the *OtherAgents* list. Note that *OtherAgents = AgentList - {the head of the AgentList}*). The *proposalReceiver_{ID}* role begins by receiving the *proposal(Topic)* message from the *proposalSender_{Proposal}* role and then the receiver agent changes its role to the *replyToProposalSender* role.

By default, *AgentGroupC3* is a call to the "*creatOneAgentGroup*" function which creates one agent group by getting one *agent ID* from the *RejectingList* and one *agent ID* from the *AcceptingList*.

The control then changes to the *replyToProposalSender* role. The *replyToProposalSender* role of agent *ID* has four input parameters: *KB_{ID}*, *CS_{ID}*, *Topic* and *ID_{Proposal}*. It begins by checking if it can accept *Topic* by checking four conditions: *findTopicInKB*, *notFindTopicInCS*, *notFindOppTopicInCS* and *addTopicToCS*. If all of these conditions is true, the *replyToProposalSender* sends the *accept(Topic)* message to *replyToProposalReceiver_{Proposal}* role. Otherwise, the *replyToProposalSender* role checks two conditions: *notFindTopicInKB* and *notFindTopicInCS*. If these two conditions are true, it sends the *reject(Topic)* message to the *replyToProposalReceiver_{Proposal}* role. Then it changes its role to the *resultReceiver_{ID}* role.

Argument	Meaning
AgentList	Agents ID list
NAgent	The number of agents (note that the number of agents ≥ 3)
NSupporters	The number of supporters agents which is used to end a dialogue when agents reach an agreement (when the supporter number is equal to the number of the acceptance agents)
Topic	Main dialogue topic
ID _{Proposal}	Proposal agent ID
OtherAgents	Agents ID list Where, OtherAgents = AgentList - {The head of the AgentList}
KB _{ID}	Agent Knowledge Base
CS _{ID}	Agent Commitment Store
AcceptanceList	The list of the accepting agents ID (note that when $\text{replyToPrposalReceiver}_{\text{Proposal}}$ role is called <i>AcceptanceList</i> is empty)
RejectioList	The list of the rejected agents ID (note that when $\text{replyToPrposalReceiver}_{\text{Proposal}}$ role is called <i>RejectioList</i> is empty)
N _{AccAgents}	The number of accepted agents (note that when $\text{replyToPrposalReceiver}_{\text{Proposal}}$ role is called <i>N_{AccAgents} equal 0</i>)
N _{RejAgents}	The number of rejected agents (note that when $\text{replyToPrposalReceiver}_{\text{Proposal}}$ role is called <i>N_{RejAgents} equal 0</i>)
N _{Reply}	The number of reply agents (note that when $\text{replyToPrposalReceiver}_{\text{Proposal}}$ role is called <i>N_{Reply} equal 0</i>)
AgentGroup	Agent group list. Each element of the agent group list is composed of two agents ID (P,O)
P	Agent ID
O	Agent ID

Table 5.1 : Broadcasting Pattern Roles Arguments

The *replyToProposalReceiver_{Proposal}* role of the proposal agent has four input parameters: *AgentList*, *NAgent*, *NSupporters* and *Topic*. It also has five output parameters: *NReply*, *AcceptingList*, *RejectionList*, *NAccepting* and *NRejection*. The values of the output parameters when the role begins are as follows: *NReply=0*, *AcceptingList=[]*, *RejectionList=[]*, *NAccepting=0* and *NRejection=0*.

The *replyToProposalReceiver_{Proposal}* begins by receiving either the *accept(Topic)* or *reject(Topic)* from the *replyToProposalSender* role. If it receives *accept(Topic)* message, it: (1) adds the accepting agent *ID* to the *AcceptingList* by achieving

Bridging the Specification Protocol Gap in Argumentation

Function	Meaning
getAgentIDFromList (AgentList,otherAgents,ID)	The <i>getAgentIDFromList</i> function gets one agent ID from the <i>AgentsList</i> and puts the remainder agents in the <i>otherAgents</i> list.
addTopicToCS (Topic,CS)	The <i>addTopicToCS</i> function always returns true and results in the agent adding <i>Topic</i> to its commitment store <i>CS</i> .
agentListEmpty (AgentList)	The <i>agentListEmpty</i> function returns true if <i>AgentList</i> is empty (which means that proposal agent broadcasts the <i>Topic</i> to all agents)
findTopicInKB (Topic, KB _{ID})	The <i>findTopicInKB</i> function returns true if the agent is able to find <i>Topic</i> in its Knowledge Base <i>KB</i>
notFindTopicInCS (Topic,CS _{ID})	The <i>notFindTopicInCS</i> function returns true if the agent is not able to find <i>Topic</i> in its Commitment Store <i>CS</i>
notFindOppTopicInCS (not(Topic),CS _{ID})	The <i>notFindOppTopicInCS</i> which returns true if the agent is not able to find the opposite of <i>Topic</i> (<i>not(Topic)</i>) in its commitment store <i>CS</i>
notFindTopicInKB (Topic,KB)	The <i>notFindTopicInKB</i> function returns true if the agent is not able to find <i>Topic</i> in its Knowledge Base <i>KB</i>
addToAcceptingList (AcceptingList,AccList,ID)	The <i>addToAcceptingList</i> function always returns true and results in proposal agent adding the accepting agent <i>ID</i> to the <i>AcceptingList</i> ($AccList = AcceptingList \cup \{ID\}$).
addToRejectingList (RejectingList,RejList,ID)	The <i>addToRejectingList</i> function always returns true and results in proposal agent adding the rejecting agent <i>ID</i> to the <i>RejectingList</i> ($RejList = RejectingList \cup \{ID\}$).
increaseRejecting (NRejecting,NRej)	The <i>increaseRejecting</i> function increases the number of rejecting agents by adding one to <i>NRejecting</i> ($NRej = NRejecting + 1$)
increaseAccepting (NAccepting,NAcc)	The <i>increaseAccepting</i> function increases the number of accepting agents ($NAcc = NAccepting + 1$)
increaseReply (NReply,NRep)	The <i>increaseReply</i> function increase the number of replying agents by adding one to <i>NReply</i> ($NRep = NReply + 1$)
RejList= RejectingList	Assigns the value of <i>RejectingList</i> argument to the <i>RejList</i> variable
NRej is NRejecting	Assigns the value of <i>NRejecting</i> argument to the <i>NRej</i> variable

Table 5.2 (a): Broadcasting Pattern Functions

Bridging the Specification Protocol Gap in Argumentation

Function	Meaning
AccList=AcceptingList	Assigns the value of <i>AcceptingList</i> argument to the <i>AccList</i> variable
NAcc is NAccepting	Assigns the value of <i>NAccepting</i> argument to the <i>NAcc</i> variable
isEqual(NRep,NAgent)	The <i>isEqual</i> function returns true if the number or replied agents <i>NRep</i> is equal to the number of agents <i>NAgent</i> .
greaterThanOrEequal(NAccepting, NSupporters)	The <i>greaterThanOrEequal</i> function returns true if the number of accepting agents <i>NAccepting</i> is greater than or equal to the number of supporter agents <i>NSupporters</i> . (<i>NAccepting</i> >= <i>NSupporters</i>)
lessThan(NAccepting ,NSupporters)	The <i>lessThan</i> function returns true if the number of accepting agents <i>NAccepting</i> is less than the number of supporter agents <i>NSupporters</i> .
creatOneAgentGroup(RejectingList,Re,AcceptinList,Ac, AgentGroup, AGroup,P,O)	The <i>creatOneAgentGroup</i> function: (1) Creates one agent group by getting one agent <i>O</i> from the <i>Rejectinglist</i> and one agent <i>P</i> from the <i>Acceptinglist</i> ; and (2) Adds the new agents groups to <i>AGroup</i> list (<i>AGroup</i> = <i>AgentGroup</i> + {(<i>P,O</i>)}; and (3) Saves the remained rejection agent in <i>Re</i> list and saves the remained accepting agents in <i>Ac</i> .
isListEmpty(Re) or isListEmpty(Ac)	The <i>isListEmpty</i> function returns true if <i>Re</i> (or <i>Ac</i>) list is empty list

Table 5.2 (b): Broadcasting Pattern Roles Functions

addToAcceptingList function; (2) increases the number of accepting agents by achieving *increaseAccepting* function; (3) increases the number of replying agents by achieving *increaseReply* function; (4) gives default value for the *RejList* argument (*RejList=RejectingList*); and (5) gives default value for the *NRej* argument (*NRej is NRejecting*). If the *replyToProposalReceiver_{Proposal}* role receives the *reject(Topic)* message, it: (1) adds the rejecting agent *ID* to the *RejectingList* by achieving *addToRejectingList* function; (2) increases the number of rejecting agents by

achieving *increaseRejecting* function; (3) increases the number of replying agents by achieving *increaseReply* function; (4) gives default value for the *AccList* argument (*AccList=AcceptingList*); and (5) gives default value for the *NAcc* argument (*NAcc* is *NAccepting*).

The proposal agent then changes the *replyToProposalReceiver_{proposal}* role to the *resultSender_{proposal}*. The *resultSender_{proposal}* role has nine input parameters: *AgentList*, *NAgent*, *NSupporters*, *Topic*, *NReply*, *AcceptingList*, *RejectionList*, *NAccepting* and *NRejection*. The *replyToProposalReceiver_{proposal}* role begins by checking *TerminationC1* condition. If this condition is true, then the proposal agent changes its role to the *sendReachAgreement_{proposal}* role. Otherwise, the *replyToProposalReceiver_{proposal}* role checks *DivideC2* condition. If this condition is true, then the proposal agent changes its role to the *divideGroup_{proposal}* role.

The *sendReachAgreement_{proposal}* role has two parameters: *AgentList* and *Topic*. It begins by sending the *reachAgreement(Topic)* message to one agent (at the head of the *AgentList* list) and then it recurses over the remaining agents (recurses over the *OtherAgents* list, where *OtherAgents = AgentList - {the head of the AgentList}*). The *sendReachAgreement_{proposal}* role ends once the *reachAgreement(Topic)* message is sent to all the agents.

The *divideGroup_{proposal}* role has six input parameters: *AgentList*, *NAgent*, *NSupporters*, *Topic*, *AcceptingList* and *RejectionList*. It also has one output parameter: *AgentGroup*. This role is responsible for dividing the agents in the *AgentList* list into a group composed of two agents. It begins by checking *AgentGroupC3*. If this condition is true, then this role creates the first agent group by taking one agent from the head of the *AcceptingList* and one agent from the head of the *RejectionList*. It then sends the *argueWith* message to the first group (agent *P* and agent *O*) and asks them to start arguing together about the dialogue *Topic*. Then, if the *RecursC4* condition is true, the proposal agent changes its role to the *recurs_{proposal}* role, otherwise, it recurses.

Finally, the control changes to the *resultReceiver_{ID}* role. The *resultReceiver_{ID}* role of agent *ID* has four input parameters: *KB_{ID}*, *CS_{ID}*, *Topic* and *ID_{Proposal}*. It begins by receiving either the *reachAgreement(Topic)* message or the *argueWith(Topic,P,O)* message from the proposal agent. The *resultReceiver_{ID}* role ends once it has received the *reachAgreement(Topic)* message. Otherwise, agent *ID* changes its role to *startDID* role.

Rewriting methods: none

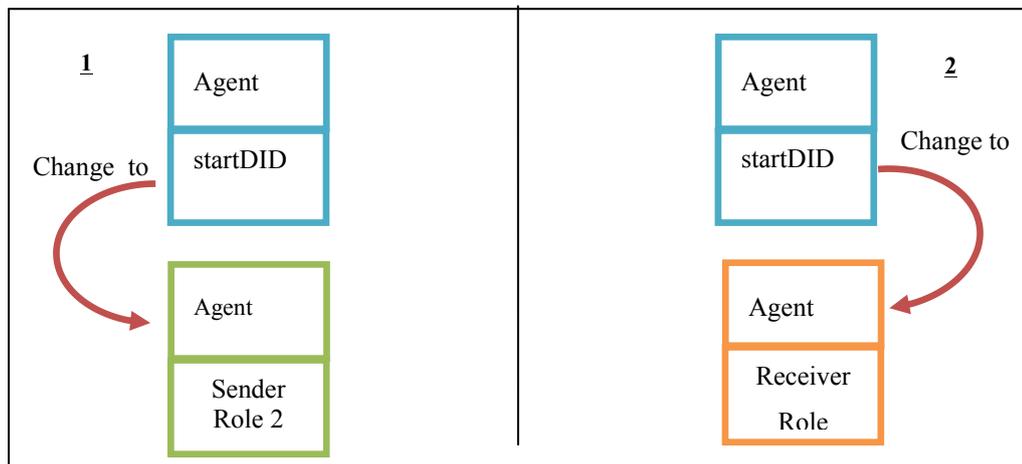
Pattern 4:

Name: Move-To-Dialogue Pattern (MTDP).

Problem: How to move from a dialogue for N-agent to a dialogue for two agents.

Solution:

- (1) The agent changes its role to the sender starting role of the two agent dialogue, if it is able to satisfy the conditions of the sender role;
- (2) Or the agent changes its role to the receiver starting role of the dialogue between two agents if it is able to satisfy the conditions of the receiver role;



Context (Pre-conditions): Use a Move-To-Dialogue Pattern to connect the N-agent dialogue with a two agents' dialogue.

Consequence (Post-conditions): Start the dialogue between two agents.

Structure:

$$\begin{aligned}
 & \mathbf{a}(\mathbf{startDID}_{ID}(\mathbf{KB}_{ID}, \mathbf{CS}_{ID}, \mathbf{CS}_{PartnerID}, \mathbf{Topic}, \mathbf{ID}_{Proposal}, \mathbf{PartnerID}), \mathbf{ID}) ::= \\
 & \mathbf{a}(R_{Sender1}(\mathbf{KB}_{ID}, \mathbf{CS}_{ID}, \mathbf{CS}_{PartnerID}, \mathbf{Topic}, \mathbf{ID}_{Proposal}, \mathbf{PartnerID}), \mathbf{ID}) \leftarrow C1 \\
 & \text{or} \\
 & \mathbf{a}(R_{Receiver1}(\mathbf{KB}_{ID}, \mathbf{CS}_{ID}, \mathbf{CS}_{PartnerID}, \mathbf{Topic}, \mathbf{ID}_{Proposal}, \mathbf{PartnerID}), \mathbf{ID}) \leftarrow C2.
 \end{aligned}$$

Where $R_{Sender1}$ represents the first sender role in the dialogue between two agents and $R_{Receiver1}$ represents the first receiver role in the dialogue between two agents. $C1$ represents a condition that must be satisfied in order for an agent to change its role to the sender role (the Starting Locution sender role of the dialogue between two agents). $C2$ represents a condition that must be satisfied in order for an agent to change its role to the receiver role (the Starting Locution receiver role of the dialogue between two agents).

Rewriting methods: none

Pattern 5:

Name: Recurs-To-N-Dialogue Pattern (RTNDP).

Problem: How to inform the proposal about the ending of the dialogue between two agents.

Solution:

- (1) Each agent (in the dialogue between two agents) sends an *end* message to the proposal agent when the dialogue between two agents terminates.
- (2) The proposal agent sums up the reply and changes its role to the *proposalSender_{proposal}*, only if the number of replied agents equals the number of agents. See Figure 5.8.

Context (Pre-conditions): The dialogue between two agents has terminated.

Consequence (Post-conditions): N-agent dialogue recurs.

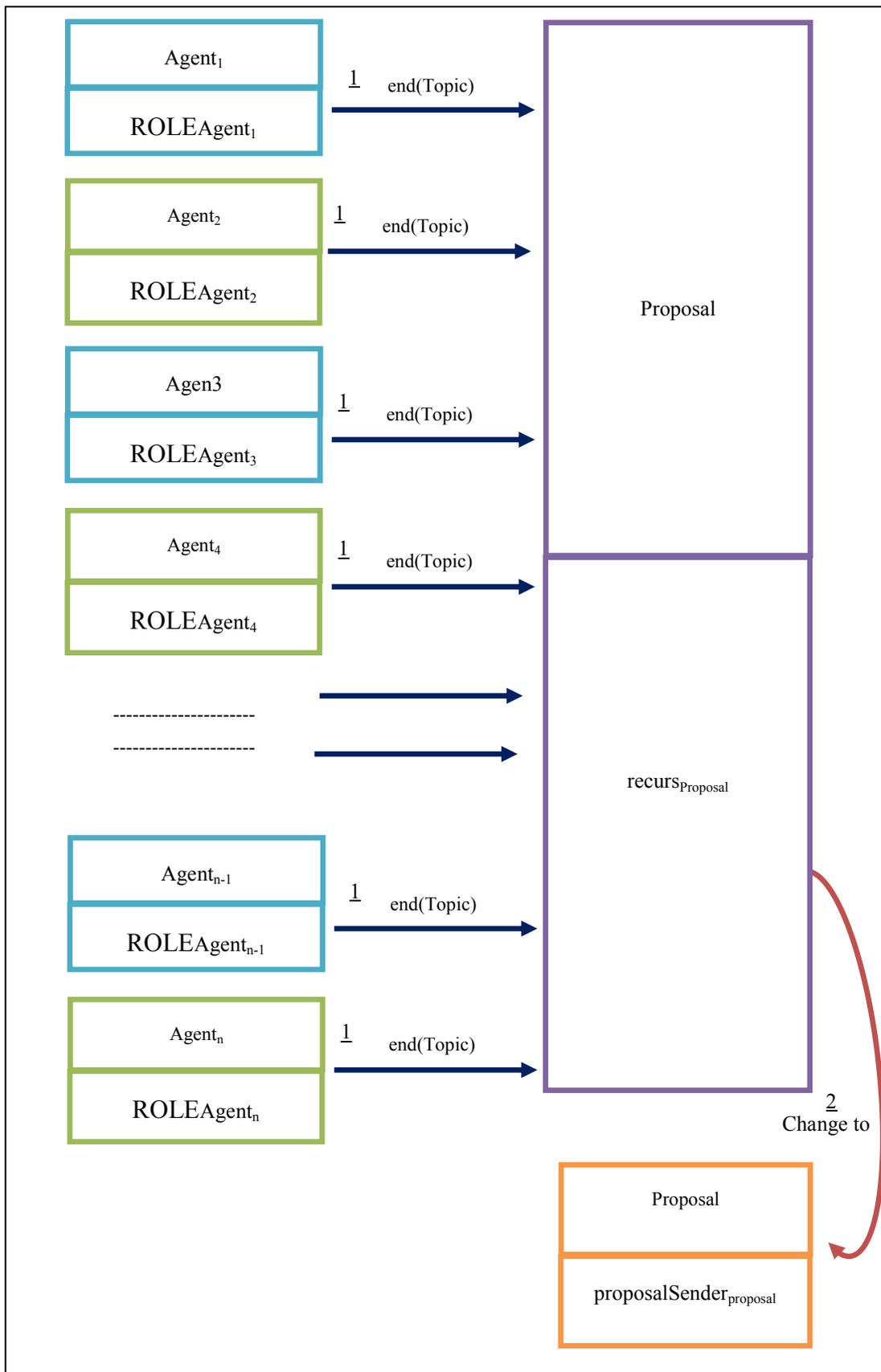


Figure 5.8: Solution of Recurs-To-N-Dialogue Pattern

Structure:

```

a(recursProposal (AgentList, NAgent, NSupporters, NReply, Topic), IDProposal)
 ::=
 N = NReply + 1 ← end(Topic)
 <= a( Rsender ( _ , _ , _ , Topic, IDProposal, _ ), IDsender )
 or
 N = NReply + 1 ← end(Topic)
 <= a( Rreceiver ( _ , _ , _ , Topic, IDProposal, _ ), IDreceiver )

 or

 recursProposal <≈end RSender2

 then
 (
 a(proposalSenderproposal (AgentList, NAgent, NSupporters, Topic), IDproposal)
 ← isEqual(N, NAgent)
 or
 a( recursProposal (AgentList, NAgent, NSupporters, N, Topic), IDProposal ) .

```

In this LCC code, there is one role *recurs_{Proposal}*. The *recurs_{Proposal}* role of the proposal agent *ID_{Proposal}* has five input parameters: *AgentList*, *NAgent*, *NSupporters*, *NReply* and *Topic*. The *recurs_{Proposal}* role begins by receiving two or more *end* locutions from sender agents *R_{sender}* and receiver agents *R_{Receiver}* (*R_{sender}* and *R_{Receiver}* role in the LCC protocol between two agents). Then, it checks *isEqual* condition (*isEqual* condition returns true if the number of replied agents *N* is equal to the number of agents *NAgents*). If *isEqual* condition is true, the proposal agent changes its role to the *proposalSender_{Proposal}* role, otherwise, it recurses.

Rewriting methods:
Rewriting of the "recurs_{Proposal} <≈^{end} R_{Sender2}"

If there is a general relation of "recurs_{Proposal} <≈^{end} R_{Sender2}" then it is possible to specialise within two different statements:

Rewrite 1: (one end locution)

We might specialise "recurs_{Proposal} \ll R_{Sender2}" to an interaction statement that sends two *end(Topic)* messages (one from sender agent and one from receiver agent in the LCC protocol for two agents) to the proposal agent.

N = NReply + 1 \leftarrow end(Topic) \ll a(R_{sender2} (_ , _ , _ , Topic, ID_{Proposal} _ ,), ID_{sender2})
 or
 N = NReply + 1 \leftarrow end(Topic) \ll a(R_{receiver2} (_ , _ , _ , Topic, ID_{Proposal}, _), ID_{receiver2})

Rewrite 2: (multiple end locutions)

We might specialise " recurs_{Proposal} \llapprox R_{Sender2} " to an interaction statement that sends two *end(Topic)* messages (one from sender agent and one from receiver agent in the LCC protocol for two agents) to the proposal agent. Then, there is another relation between proposal agent and senders (recurs_{Proposal} \llapprox R_{Sender3}).

N = NReply + 1 \leftarrow end(Topic) \ll a(R_{sender2} (_ , _ , _ , Topic, ID_{Proposal} _ ,), ID_{sender2})
 or
 N = NReply + 1 \leftarrow end(Topic) \ll a(R_{receiver2} (_ , _ , _ , Topic, ID_{Proposal}, _), ID_{receiver2})
 or
 recurs_{Proposal} \llapprox R_{Sender3}

This section describes in detail five LCC–Argument patterns. In the next section, we will use these five patterns along with DID to generate an LCC agent protocol.

5.2 Agent Protocol Automated Synthesis Tool

LCC–Argument patterns only provide a general solution to the common agent argumentation protocol development problems. Even though these patterns include some LCC codes they are not codes in themselves (final protocol) [Budinsky et.al., 1996]. Therefore, we need an automated synthesis tool that can be used to translate the patterns into final code.

Our automated agent protocol synthesis tool "**GenerateLCCProtocol**" (see chapter 7 for more details), summarised pictorially in Figure 5.9, can generate agent protocols

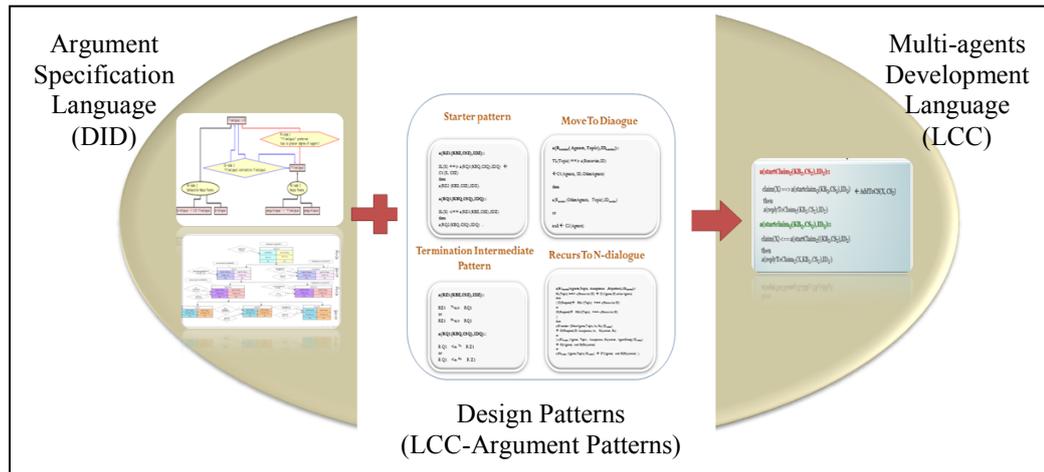


Figure 5.9: Agent Protocol Automated Synthesis Tool

Locution Type	Pattern Name
Starting Locution	Starting Pattern
Termination Locution	Termination- Intermediate Pattern
Intermediate Locution	Termination- Intermediate Pattern

Table 5.3: Relationship Between Locution Type and Patterns

from DID diagrams automatically. It receives as input a DID and returns the corresponding LCC argumentation agent protocol by using LCC–Argument patterns. In practice, by using this tool, no additional programming is required.

5.2.1 Automated Synthesis Steps for Generating Agent Protocol between Two Agents

In general, during the automated synthesis process, every time we progress from level to level in the DID diagram the tool generates a pair of LCC clauses or roles and switches roles (the sender agent will become the receiver and vice versa). The automated synthesis process occurs from the top-down and moving left to right. The synthesis process matches each level of the DID with only one LCC-Argument pattern.

The automated synthesis process of the two agents' protocol consists of five steps (The two agents protocol automated synthesis algorithm is illustrated in Figure 5.10. A worked example is described in detail in appendix C):

1.	Input (DID, LCC-Argument patterns)	
2.	Select&Save Icon= one DID locution icon	(Step1)
3.	Select&Save Pattern= one pattern from the LCC-Argument patterns library	(Step2)
4.	If (Pattern has rewriting methods) then	(Step3)
5.	If (level has one locution icon) then	
6.	Select&Save RewriteMethod= Rewrite 1	
7.	If (level has more than one locution icon) then	
8.	Select&Save RewriteMethod= Rewrite 2	
9.	Match (Icon,Pattern,RewriteMethod)	(Step4)
10.	Go To line 2	(Step5)
11.	End matching all level in the DID with the corresponding patterns	
12.	Output LCC protocol	

Figure 5.10: Two Agents Protocol Automated Synthesis Algorithm

- (1) The tool begins with the locution icon at the top of the DID. Note that if more than one locution icon appears in one level, then the tool begins with the locution to the left (since it works from left to right).
- (2) Following this, the tool selects one pattern from the LCC-Argument patterns library. This pattern depends on the locution type. Note that each locution type is connected to only one LCC-Argument pattern. See Table 5.3.
- (3) After that, if the selected pattern has rewriting methods, the tool selects one or more of the rewriting methods. The number of rewriting methods selected is dependent on the number of locution icons in this level. If this level has one locution icon, the tool selects the rewriting method **Rewrite 1** (rewriting method with one locution). If this level has more than one locution icon, the tool selects the rewriting method **Rewrite 2** (rewriting method with multiple locutions).
- (4) Finally, the tool applies the selected pattern by matching formal parameters (variables) with its corresponding values in the locution icon to generate pairs of LCC clauses or roles (sender and receiver roles). If the selected pattern has rewriting methods, the tool matches the formal parameters (variables) in the selected rewriting methods with its corresponding values in the locution icon, to generate pairs of LCC clauses or roles. The matching process matches one parameter at a time. It begins with the locution icon and occurs from the top-

down and left to right. It then moves to the left side conditions and then to the right side conditions. Finally, if the selected pattern has recursive (changing) roles, the tool moves to the next level and matches the recursive roles in the pattern with the recursive roles in the locution icon on the next level.

- (5) Moves to the next level in the DID and repeats steps 2, 3 and 4. Note that the automated synthesis process finishes when the tool matches the last level in the DID with one of the LCC-Argument patterns. If the selected pattern has recursive (changing) roles, the tool moves to the locution icon reply level, which represents the reply rules of the selected locution icon, and matches the recursive roles in the pattern with the recursive roles in the locution icon on this level.

5.2.2 Automated Synthesis Steps for Generating Agent Protocol for N-agent

In general, during the automated synthesis process of the N-agents' protocol, the tool uses Broadcasting, Move-To-Dialogue and Recurs-To-N-Dialogue patterns to divide agents into groups of two and to generate LCC protocols for N-agent. It then follows the automated synthesis process of the two agents' protocol (see section 5.4) to generate the LCC protocol from the DID for two agents, which allows pairs of groups to communicate with each other.

The automated synthesis process of the N-agents' protocol consists of four steps (The N-agents' protocol automated synthesis algorithm is illustrated in Figure 5.11. A worked example is described in detail in appendix C):

- (1) The tool begins with the Broadcasting Pattern. It gets *TerminationC1*, *DivivdeC2*, *AgentGroupC3*, and *RecursC4* conditions from the user. Note that if the user does not specify these conditions, the tool uses the default functions of these conditions (see section 5.3 pattern 4).
- (2) Following this, the tool uses the *Move-To-Dialogue Pattern* to connect N-agents' dialogue with the two agents' dialogue. The tool applies this pattern by matching

1. **Tool Input** (DID, LCC-Argument patterns)
2. **Use Broadcasting Pattern** (Step1)
3. **Pattern Input** (TerminationC1, DivivdeC2, AgentGroupC3, RecursC4)
4. **Use Move-To-Dialogue Pattern** (Step2)
5. **Match**(Starting locution icon in the DID for two agents, Move-To-Dialogue Pattern)
6. **Use Recurs-To-N-Dialogue Pattern** (Step3)
7. recursNumber = number of Termination locution icon in the DID for two agents -1
8. **If** (reurseNumber = 0) then *//one Termination Locution*
9. Select&Save RewriteMethod=**Rewrite 1**
10. **Match** (Termination Icon, Recurs-To-N-Dialogue Pattern, RewriteMethod)
11. **Else** *//more than one Termination Locution*
12. **Loop begin** (if i=1)
13. Select&Save RewriteMethod=**Rewrite 2**
14. **Match** (Termination Icon, Recurs-To-N-Dialogue Pattern, RewriteMethod)
15. i= i+1
16. **Loop end** (if i = reurseNumber)
17. **Go To** two agents algorithm (Step4)
18. **Add** lines to connect N-agents' protocol with two agents' protocol
19. **Output** LCC protocol

Figure 5.11: N-agents' Protocol Automated Synthesis Algorithm

formal parameters (variables) with its corresponding values in the Starting locution icon in the DID for two agents to generate one LCC role.

- (3) After that, the tool uses the *Recurs-To-N-Dialogue Pattern* to generate the LCC role which is used to inform the proposal agent about the ending of the dialogue between two agents:

- a) The tool selects one or more rewriting methods. The number of selected rewriting methods is the number of the *Termination Locution* icons in the DID for two agents, minus one. For example, if the number of *Termination Locution* icons is equal to five, then the number of *end messages* is equal to $5 \times 2 = 10$ and the number of rewriting methods is equal $5-1= 4$. Each rewriting methods has two end messages and by default *Recurs-To-N-Dialogue* pattern receives two *end messages* one from the first *Termination*

Locution sender role and one from the first *Termination Locution* receiver role.

b) The tool applies this pattern by matching the formal parameters (variables) with their corresponding values in the Termination locution icons in the DID for two agents to generate one of the LCC clauses or roles for the proposal agent.

(4) Finally, the tool follows the steps of the automated synthesis process of two agents' protocol to generate the LCC protocol from the DID for two agents. Note that the tool adds two lines after each Termination message (locution) in the LCC protocol for two agents to connect N-agents' protocol with two agents' protocol:

- Line one: Sending end message to proposal.
- Line two: Changing agents' role to *proposalReceiver_{ID}* (agent change from the LCC protocol for two agents to LCC protocol for N-agent).

```
(
  TL (Topic) => a(R, ID)
  then

  end(Topic)=>
  a(recursProposal(AgentList, NAgent, NSupporters, NReply, Topic), IDProposal)

  then

  a(proposalReceiverID (KBID, CSID, IDproposal), ID)
)
```

5.3 Summary

This chapter has presented a set of LCC–Argument patterns as well as a fully automated synthesis method to generate LCC argumentation agent protocols by using DID and LCC-Argument patterns. In practice, the argument LCC protocol is

quite complex, and therefore requires considering issues that the software engineer may not be aware of until later in the implementation process, such as synchronisation of the role. The usage of DID and LCC-Argument patterns can speed up the protocol development process and help to prevent subtle design issues that can cause errors in the protocol. It also improves code readability and the efficiency of role synchronisation mechanisms.

Our automated synthesis tool enables to generate any LCC argumentation agent protocol for two agents. However, in the case of the dialogue between N-agent ($N \geq 3$), the automated synthesis tool uses a broadcasting method to divide agents into groups composed of two agents under certain conditions. Then the tool uses DID and LCC-Argument patterns for two agents to allow pairs of groups to communicate with each other. Therefore, the user needs to either write a new LCC protocol or define new patterns to be able to work with different structures concerning how the set of agents is partitioned. This means that in the case of N-agent there is no finite, complete set of patterns.

Adding new patterns and writing protocols from scratch requires profound knowledge of agent protocols, and adding new patterns risks introducing errors into the synthesiser. It is impractical to ask software engineers to ensure that the protocol is error-free each time they want to write a protocol or add new patterns or to fully consider the semantics of the DID. Therefore, the next chapter proposes a verification model, which is used to ensure that key properties of the DID specification are preserved by the resulting LCC protocol.

Chapter 6

Verification Method based on Coloured Petri Nets and SML

Chapter 5 discussed the automatic generation of LCC protocols from DID by using LCC-Argument patterns and concluded that checking the validity of the generated protocols is necessary since design patterns introduce greater scope for inaccuracy and error in the synthesis process (a poorly designed interaction pattern may result in inappropriate LCC protocols, even with a perfect synthesis mechanism).

This chapter proposes a verification methodology based on CPN and SML language to verify the semantics of the DID specification against the semantics of the synthesised LCC protocol. We automatically transform an LCC protocol to a Coloured Petri Nets (CPNs) model, which is then used to check the validity of various concurrent behaviour properties of the resulting LCC protocol by using state space techniques and CPN SML language. The verification process, illustrated in Figure 6.1, is divided into four steps:

1. Automated transformation LCC protocol to CPNXML file;
2. Construction of state space;
3. Automated creation of DID properties file;
4. Applying the verification process.

This chapter discusses the details of each of these four steps. Section 6.1 describes the automated transformation approaches from an LCC protocol to CPNXML file. Section 6.2 highlights the construction of state space approaches. Section 6.3 describes the automated creation approaches of DID properties file. Section 6.4 details the verification approach for the LCC protocol and Section 6.5 Section 6.6 summarises this chapter.

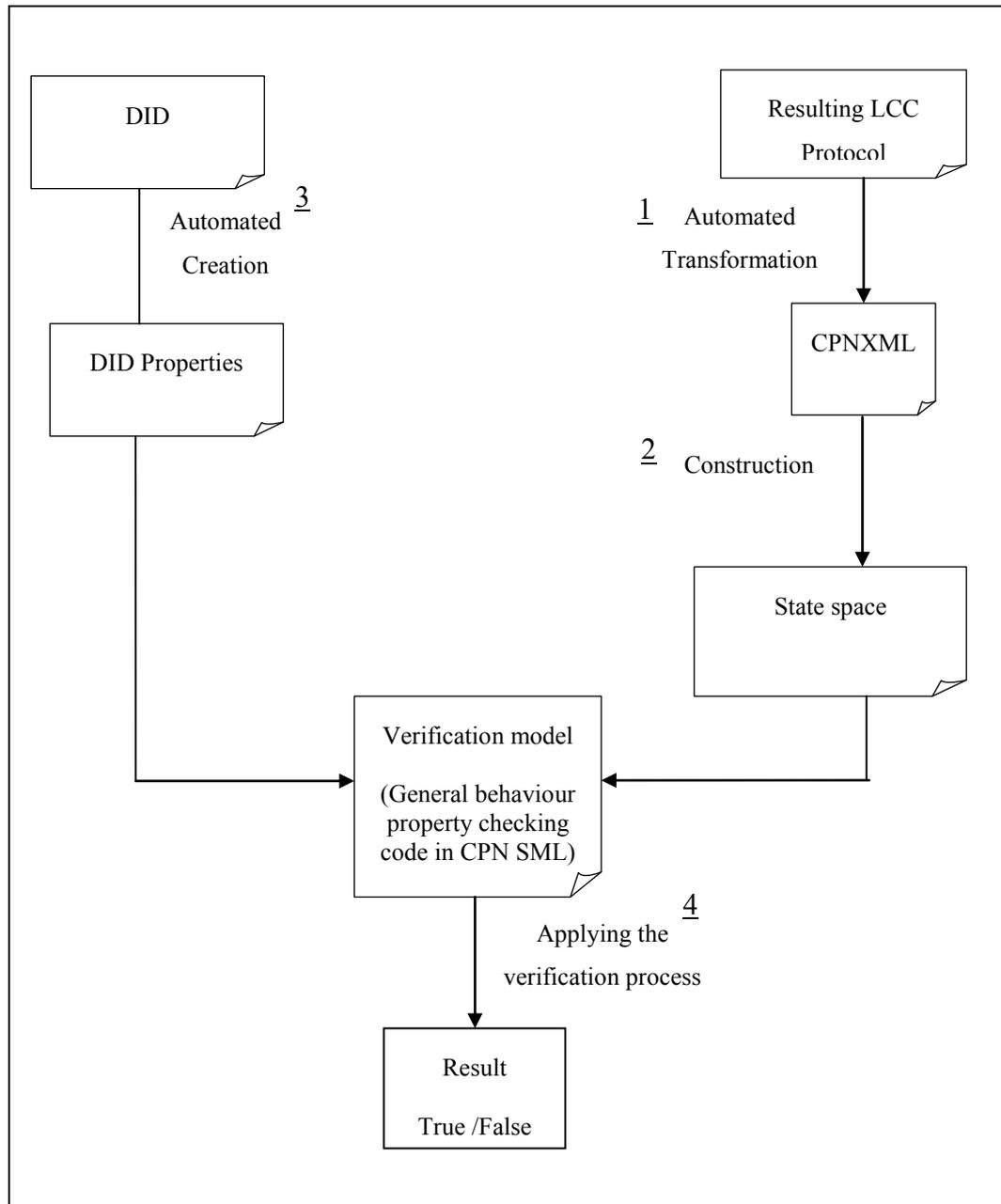


Figure 6.1: Verification Process

6.1 Step One: Automated Transformation from LCC to CPNXML

We have developed a step-by-step technique that allows to transform an LCC protocol into the CPNXML file (see chapter 2 for more details about CPNXML file) by:

- (1) Declaring colour sets and functions.
- (2) Generating a CPN subpage for each LCC role. Each subpage represents a role behaviour.
- (3) Connecting all the CPN subpages by generating one CPN superpage, which describes the interaction between roles, where the messages that are passed between two roles determine the interaction between the subpages of the two roles.

In practice, to automate the transformation process from an LCC protocol into CPNXML file we use LCC-CPNXML tables (Table 6.1, Table 6.2, Table 6.3, Table 6.4, Table 6.5, Table 6.6, Table 6.7, Table 6.8, Table 6.9, Table 6.10, Table 6.11 and Table 6.12), where transitions and places are connected according to a set of transformation rules. The use of LCC-CPNXML tables make the transformation faster and the resulting CPN model can be executed with data and analysed, not only by our tool, but also by other users (using CPN Tool) since CPN has a comprehensible graphical representation.

The following sections give more details of the transformation process from an LCC protocol into CPNXML file. Readers not interested in these details can skip ahead to section 6.2 for constructing state space step.

6.1.1 Declaration of Colour Sets and Functions

Declaration of Colour Sets

We use three different primary types of colour sets:

- (1) Type *TOPIC*. This type is used to model the main dialogue topic. It is defined as a string.

colset TOPIC = string;

- (2) Type *Message*. This type is used to model messages. It is defined as the product of the types *Locution*, *TOPIC*, *Premise*, *ID* and *ID*. The types *Locution*, *TOPIC*,

Bridging the Specification Protocol Gap in Argumentation

Premise and *ID* are defined as a string. *Locution* type represents locution (message) name (e.g. claim). *TOPIC* type represents the main dialogue topic. *Premise* type represents the topic premise. *ID* type represents agent ID. The first *ID* in the Message type represents the message sender agent's ID and the second *ID* in the Message type represents the message receiver agent's ID.

```
colset Message = product Locution * TOPIC * Premise * ID * ID ;
```

- (3) Type *Role*. This type is used to model role arguments. It is defined as the product of the types *ID*, *CSlist*, *KBlist*, *RoleName*, *TOPIC*, *Premise*, *CSlist* and *ID*. The types *RoleName*, *TOPIC*, *Premise* and *ID* are defined as a string. The *RoleName* represents the new (recursive) role name. The *TOPIC* type represents the main dialogue topic. The *Premise* type represents the topic premise. The *ID* type represents agent ID. The first *ID* in the Role type represents agent's ID and the second *ID* in the Role type represents the other agent's ID. The type *CSlist* is defined as a list of *CS* representing the possible contents of the agent commitment store at a specific time. The type *CS* is defined as a string. The first *CSlist* in the *Role* type represents agent's *CS* and the second *CSlist* in the *Role* type represents other agent's *CS*. The type *KBlist* is defined as a list of *FactXPremise* representing the possible contents of the agent knowledge base at a specific time. The type *FactXPremise* is defined as a product of the types *Fact* and *Premise*. Both *Fact* and *Premise* are defined as a string. The *Fact* type represents the agent belief and the *Premise* type represents the agent proposition or premise which is used to prove that an agent's belief is true (e.g. *Fact*= "The car is safe" and *Premise*="The car has an airbag").

```
colset FactXPremise= product Fact * Premise;  
colset KBlist =list FactXPremise;  
colset CS=string;  
colset CSlist = list CS;  
colset Role =  
    product ID* CSlist*KBlist*RoleName* TOPIC * Premise* CSlist*ID ;
```

Declaration of Functions

As mentioned in chapters 3 and 4, each agent has a knowledge base *KB* (private knowledge) and a commitment store *CS* (common knowledge). During the dialogue game the agents take turns to make moves. Each agent makes his choice between possible moves depending on its *CS* and *KB*. In practice, the *CS* is continuously updated at each move by either adding to or subtracting from it arguments.

For that reason, we defined thirteen different basic functions which are used to find, get, add or subtract an argument from either a *CS* or *KB* list. These functions are written in the CPN SML language [Jensen and Kristensen, 2009; Ullman, 1998]. See appendix D for a detailed explanation of these functions:

- (1) Add an argument 't' to a CS list:

addTopicToCS

- (2) Add a premise of an argument 't' to a CS list:

addPremiseToCS

- (3) Add a defeat of a premise or an argument to a CS list:

addDefeatToCS

- (4) Subtract an argument 't' from a CS list:

subtractFromCS

- (5) Find an argument 't' in a CS list:

findTopicInCS

- (6) Find a premise of an argument 't' in a CS list:

findPreInCS

- (7) Find an argument in a KB list:

findTopicInKB

- (8) Find a premise of an argument in a KB list:

findPreInKB

- (9) Find a defeat of a premise or an argument in a KB list:

findDefeatInKB

- (10) Find the opposite of an argument 't' in a CS list:

findOppTopicInCS

- (11) Find the opposite of the premise 'p' of an argument 't' in a CS list:

findOppPreInCS

- (12) Return (get) the premise of an argument 't' from a KB list:

getPremiseFromKB

- (13) Return (get) the defeat of an argument 't' from a KB list:

getDefeatFromKB

The CPNXML format of the three types of colour sets and thirteen functions are saved in the Global Declaration file called "*CPNmainCode*". The user does not need to know about these colour set types or functions unless he/she needs to define new types or functions. For more information about how to define new CPN SML colour set types or functions, please read [Westergaard and Verbeek, 2002; Aalst and Stahl, 2011; Jensen et al., 2007].

6.1.2 Generation of a CPN Subpage

Nine tables are used to automate the transformation process from LCC roles into CPN subpages.

<i>LCC</i> (Role)	a(RoleName(Arguments,Topic),AgentID)
LCC CPNs Model	CPNXML Structure
<i>non</i>	<page id="ID6"> <pageattr name= <i>Role Name</i> /> </page>

Table 6.1: LCC-CPNXML Transformation Table (Role)

Table one: LCC Role

Generate a new subpage for each LCC role where (as shown in Table 6.1):

- 1) The beginning of a page block is identified by the start tag <page>;
- 2) The end of a page block is identified by the end tag </page>;
- 3) The page ID= unique identifier;
- 4) The page name = role name.

Table Two: LCC Message Sending Statement

The LCC message sending code is transformed into a high-level Petri net by creating (as shown in Table 6.2):

- (1) One new transition where the transition ID = unique identifier, the transition name= "Send" + Message name, and guard condition = LCC message Boolean conditions (line 1 to 7 of Table 6.2);
- (2) One new place where the place ID = unique identifier, the place name = message name, place colour set type = Message and place (port) type= Out (line 8 to 19 of Table 6.2);
- (3) One arc (output arc), which is used to connect the new transition to the new place, where the arc ID = unique identifier, the arc type= TtoP (output arc), the transition ID reference = the new transition ID, the place ID reference = the new place ID, the arc inscription = (Message arguments) (line 20 to 28 of Table 6.2).

LCC Code (Send a Message)	Message(Topic) => a(RoleName(Arguments),AgentID) ← Conditions
CPNs Model	CPNXML Structure
<p><i>Send message symbol</i></p> <p>[Boolean conditions]</p> <div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 10px auto;"> <p>"Send"+ Message Name</p> </div> <p style="text-align: center;">(Message arguments)</p> <div style="text-align: center;"> </div>	<ol style="list-style-type: none"> 1. <trans id="ID1423689023"> 2. <text> "Send"+ message name </text> 3. <cond > 4. <text tool="CPN Tools "version="2.9.11"> 5. LCC Boolean conditions </text> 6. </cond> 7. </trans> 8. <place id="ID1423689035"> 9. <text> Message name </text> 10. <type id="ID1423689036"> 11. <text tool="CPN Tools" version="2.9.11"> 12. Message </text> 13. </type> 14. <initmark id="ID1423689037"> 15. <text tool="CPN Tools" version="2.9.11"/> 16. </initmark> 17. <port id="ID1424205036" type="Out"> 18. </port> 19. </place> 20. <arc id="ID1423689049" 21. orientation="TtoP" order="1"> 22. <transend idref="New transition ID"/> 23. <placeend idref="New place ID"/> 24. <annot id="ID1423689050"> 25. <text tool="CPN Tools version="2.9.11"> 26. Message arguments </text> 27. </annot> 28. </arc>

Table 6.2:LCC-CPNXML Transformation Table (Send a message)

Table Three: LCC Message Receiving Statement

The LCC message receiving code is transformed into a high-level Petri Net by creating (as shown in Table 6.3):

- (1) One new place where the place ID = unique identifier, the place name= message name, place colour set type = Message and place (port) type = In (line 1 to 12 of Table 6.3);
- (2) One new transition where the transition ID = unique identifier, the transition name = "Receive" + Message name and guard condition = LCC message Boolean conditions (line 13 to 19 of Table 6.3);

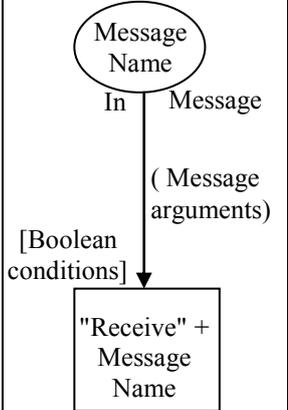
LCC (Receive a Message)	Conditions ← Message(Topic) ≤ a(RoleName(Arguments),AgentID)
CPNs Model	CPNXML Structure
<p><i>Receive message symbol</i></p> 	<ol style="list-style-type: none"> 1. <place id="ID1423689035"> 2. <text> <u>Message name</u> </text> 3. <type id="ID1423689036"> 4. <text tool="CPN Tools" version="2.9.11"> 5. <u>Message</u> </text> 6. </type> 7. <initmark id="ID1423689037"> 8. <text tool="CPN Tools" version="2.9.11"/> 9. </initmark> 10. <port id="ID1424205036" type="In"> 11. </port> 12. </place> 13. <trans id="ID1423689023"> 14. <text> <u>"Receive"+ message name</u> </text> 15. <cond > 16. <text tool="CPN Tools "version="2.9.11"> 17. <u>LCC Boolean conditions</u> </text> 18. </cond> 19. </trans> 20. <arc id="ID1424199627" 21. orientation="PtoT" order="1"> 22. <transend idref="New transition ID"/> 23. <placeend idref="New place ID"/> 24. <annot id="ID1424199628"> 25. <text tool="CPN Tools" version="2.9.11"> 26. <u>Messages arguments</u> 27. </text> 28. </annot> 29. </arc>

Table 6.3: LCC-CPNXML Transformation Table (Receive a message)

(3) One arc (input arc), which is used to connect the new place to the new transition, where the arc ID = unique identifier, the arc type= PtoT (input arc), the transition ID reference = the new transition ID, the place ID reference= the new place ID, the arc inscription = (Message arguments) (line 20 to 29 of Table 6.3).

LCC (LCC Then keyword followed by Changing Role statement)	then a(NewRoleName(Arguments),AgentID) ← ChangeRoleConditions
LCC CPNs Model	CPNXML Structure
<p style="text-align: center;"><i>Change role symbol</i></p> <p style="text-align: center;">(Changing role conditions + New role arguments)</p> <div style="text-align: center;"> </div>	<ol style="list-style-type: none"> 1. <place id="ID1423689035"> 2. <text> <u><i>New Role name</i></u> </text> 3. <type id="ID1423689036"> 4. <text tool="CPN Tools" version="2.9.11"> 5. <u><i>Role</i></u> </text> 6. </type> 7. <initmark id="ID1423689037"> 8. <text tool="CPN Tools" version="2.9.11"/> 9. </initmark> 10. <port id="ID1424205036" type="Out"> 11. </port> 12. </place> 13. <arc id="ID1423689049" 14. orientation="TtoP" order="1"> 15. <transend idref="<u><i>Last Message transition ID</i></u>"/> 16. <placeend idref="<u><i>New place ID</i></u>"/> 17. <annot id="ID1423689050"> 18. <text tool="CPN Tools version="2.9.11"> 19. <u><i>Role Arguments</i></u> 20. </text> 21. </annot> 22. </arc>

Table 6.4: LCC-CPNXML Transformation table (Then keyword and Change Role)

Table Four: LCC Recursive (Changing Role) Statement

The LCC Recursive code is transformed into a high-level Petri net by creating (as shown in Table 6.4):

- (1) One new place where the place ID = unique identifier, the place name= "ChangeRoleTo" + new role name, place colour set type = Role and place (port) type = Out (line 1 to 12 of Table 6.4);

- (2) One arc (out arc), which is used to connect the new place to the last message transition, where the arc ID = unique identifier, the arc type= TtoP (output arc), the transition ID reference = the last message transition ID, the place ID reference = the new place ID, the arc inscription = (Role arguments). Note that if the ChangeRoleConditions represents either add or subtract condition, it will appear in the Role arguments (line 13 to 22 of Table 6.4).

Table Five: LCC "or" Statement

The LCC "or" code is transformed into a high-level Petri net by creating (as shown in Table 6.5):

- (1) One new place where the place ID = unique identifier, the place name= main role name, place colour set type = Role and place (port) type = In (line 1 to 12 of Table 6.5);
- (2) One or more arcs. The number of arcs depends on the number of messages. These arcs are used to connect the new place to the message transitions. Each arc has an arc ID = unique identifier, the arc type= PtoT (input arc), the transition ID reference = the message transition ID, the place ID reference= the new place ID, the arc inscription = (Role arguments) (line 13 to 32 of Table 6.5).

Table Six: LCC Dialogue Topic Argument

The LCC Topic argument of the primary role (the first role in the LCC code which is responsible for opening the dialogue) is transformed into a high-level Petri net by creating (as shown in Table 6.6):

- (1) One new place where the place ID = unique identifier, the place name= "OpenDialogoue", the place colour set type = Topic and place (port) type= In (line 1 to 12 of Table 6.6);
- (2) One arc, which is used to connect the new place to the role message transition of the agent first role, where the arc ID = unique identifier, the arc type= PtoT

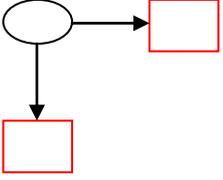
LCC (LCC or keyword)	or
LCC CPNs Model	CPNXML Structure
<p><i>Or symbol</i></p> 	<ol style="list-style-type: none"> 1. <code><place id="ID1423689035"></code> 2. <code><text> <i>Main role name</i> </text></code> 3. <code><type id="ID1423689036"></code> 4. <code><text tool="CPN Tools" version="2.9.11"></code> 5. <code> <i>Role</i> </text></code> 6. <code></type></code> 7. <code><initmark id="ID1423689037"></code> 8. <code><text tool="CPN Tools" version="2.9.11"/></code> 9. <code></initmark></code> 10. <code><port id="ID1424205036" type="In"></code> 11. <code></port></code> 12. <code></place></code> 13. <code><arc id="ID1423689049"</code> 14. <code> orientation="PtoT" order="1"></code> 15. <code><transend idref=" <i>First Message transition ID</i> "/></code> 16. <code><placeend idref=" <i>New place ID</i> "/></code> 17. <code><annot id="ID1423689050"></code> 18. <code><text tool="CPN Tools version="2.9.11"></code> 19. <code> <i>Role arguments</i></code> 20. <code></text></code> 21. <code></annot></code> 22. <code></arc></code> 23. <code><arc id="ID1423689049"</code> 24. <code> orientation="PtoT" order="1"></code> 25. <code><transend idref=" <i>Second Message transition ID</i> "/></code> 26. <code><placeend idref=" <i>New place ID</i> "/></code> 27. <code><annot id="ID1423689050"></code> 28. <code><text tool="CPN Tools version="2.9.11"></code> 29. <code> <i>Role arguments</i></code> 30. <code></text></code> 31. <code></annot></code> 32. <code></arc></code>

Table 6.5:LCC-CPNXML Transformation Table (Or keyword)

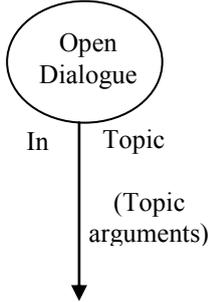
LCC (Dialogue Topic Argument)	a(RoleName(Arguments, Topic),AgentID)
LCC CPNs Model	CPNXML Structure
<p><i>Dialogue Topic symbol</i></p> 	<ol style="list-style-type: none"> 1. <place id="ID1423689035"> 2. <text> <u>OpenDialogue.</u> </text> 3. <type id="ID1423689036"> 4. <text tool="CPN Tools" version="2.9.11"> 5. . <u>Topic</u> </text> 6. </type> 7. <initmark id="ID1423689037"> 8. <text tool="CPN Tools" version="2.9.11"/> 9. </initmark> 10. <port id="ID1424205036" type="In"> 11. </port> 12. </place> 13. <arc id="ID1423689049"> 14. orientation="PtoT" order="1"> 15. <transend idref="<u>Role message transition ID</u>"/> 16. <placeend idref="<u>New place ID</u>"/> 17. <annot id="ID1423689050"> 18. <text tool="CPN Tools version="2.9.11"> 19. <u>Topic arguments</u> 20. </text> 21. </annot> 22. </arc>

Table 6.6: LCC-CPNXML Transformation table (Dialogue Topic) (input arc), the transition ID reference = the role message transition of agent first role's ID, the place ID reference= the new place ID, the arc inscription = (Topic argument) (line 13 to 22 of Table 6.6).

Table Seven: LCC Role Arguments

Each agent in the dialogue has one or more arguments. Our tool supplies these arguments by creating (as shown in Table 6.7):

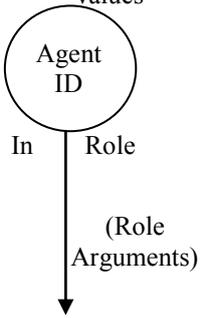
LCC Code (Starter Role Arguments)	a(RoleName(Arguments, Topic),AgentID)
LCC CPNs Model	CPNXML Structure
<p><i>Starter Role argument symbol</i></p> <p>Arguments initial values</p> 	<ol style="list-style-type: none"> 1. <code><place id="ID1423689035"></code> 2. <code><text> <u>Agent ID</u> </text></code> 3. <code><type id="ID1423689036"></code> 4. <code><text tool="CPN Tools" version="2.9.11"></code> 5. <code> <u>Role</u> </text></code> 6. <code></type></code> 7. <code><initmark id="ID1423689037"></code> 8. <code><text tool="CPN Tools" version="2.9.11"></code> 9. <code> <u>Arguments initial values</u></code> 10. <code></initmark></code> 11. <code><port id="ID1424205036" type="In"></code> 12. <code></port></code> 13. <code></place></code> 14. <code><arc id="ID1423689049"</code> 15. <code> orientation="PtoT" order="1"></code> 16. <code><transend idref=<u>Role main transition ID</u>></code> 17. <code><placeend idref=<u>New place ID</u>></code> 18. <code><annot id="ID1423689050"></code> 19. <code><text tool="CPN Tools" version="2.9.11"></code> 20. <code> <u>Role arguments</u></code> 21. <code></text></code> 22. <code></annot></code> 23. <code></arc></code>

Table 6.7: LCC-CPNXML Transformation table (Starter Role Arguments)

- (1) One new place where the place ID = unique identifier, the place name= agent ID, the place colour set type = Role and place (port) type = In (line 1 to 12 of Table 6.7);
- (2) One arc (input arc), which is used to connect the new place to the role message transition of agent first role, where the arc ID = unique identifier, the arc type= PtoT (input arc), the transition ID reference = the role message transition of an

agent first role's ID, the place ID reference = the new place ID, the arc inscription = (Role arguments) (line 13 to 22 of Table 6.7).

Table Eight: LCC "." End Statement

The LCC end statement represents the mark '.' after sending or receiving a message statement. It is transformed into a high-level Petri net by creating (as shown in Table 6.8):

- (1) One new place where the place ID = unique identifier, the place name= end, place colour set type = Role and place (port) type = Out (line 1 to 12 of Table 6.8);
- (2) One arc (output arc), which is used to connect the message transition to the new place, where the arc ID = unique identifier, the arc type= TtoP (output arc), the transition ID reference = the message transition ID, the place ID reference = the new place ID, the arc inscription = (Role arguments) (line 13 to 22 of Table 6.8).

Table nine: Get an Item from List Condition

The get an item form list condition is transformed into a high-level Petri net by creating (as shown in Table 6.9(a), Table 6.9(b) and Table 6.9(c)):

- (1) One new transition where the transition ID = unique identifier, the transition name = "getConditionTransition" and guard condition = "true" (line 1 to 7 of Table 6.9(a));
- (2) One new place where the place ID = unique identifier, the place name= the item name, place colour set type = the item type(by default the place colour set type= *Premise* which is defined as a string) (line 8 to 17 of Table 6.9(a));
- (3) One new place where the place ID = unique identifier, the place name= "flow", place colour set type = Role (line 18 to 27 of Table 6.9(a));

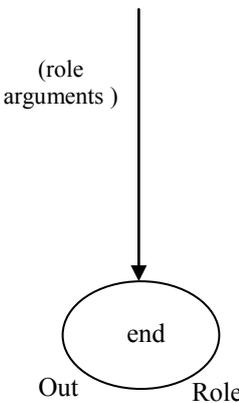
<p>LCC</p> <p>(.' After sending or receiving message statement)</p>	<p>Message(Topic) => a(RoleName(Arguments),AgentID) ← Conditions .</p> <hr/> <p>Conditions ← Message(Topic) <= a(RoleName(Arguments),AgentID) .</p>
<p>LCC CPNs Model</p>	<p>CPNXML Structure</p>
<p>End symbol</p> 	<ol style="list-style-type: none"> 1. <place id="ID1423689035"> 2. <text> <u>End</u> </text> 3. <type id="ID1423689036"> 4. <text tool="CPN Tools" version="2.9.11"> 5. <u>Role</u> </text> 6. </type> 7. <initmark id="ID1423689037"> 8. <text tool="CPN Tools" version="2.9.11"/> 9. </initmark> 10. <port id="ID1424205036" type="Out"> 11. </port> 12. </place> 13. <arc id="ID1423689049" 14. orientation="TtoP" order="1"> 15. <transend idref="Message transition ID"/> 16. <placeend idref="New place ID"/> 17. <annot id="ID1423689050"> 18. <text tool="CPN Tools" version="2.9.11"> 19. <u>Role arguments</u> 20. </text> 21. </annot> 22. </arc>

Table 6.8: LCC-CPNXML Transformation table (End Statement)

LCC Code (Get condition)	← GetConditions
CPNs Model	CPNXML Structure
<p>Get Condition symbol</p>	<ol style="list-style-type: none"> 1. <trans id="ID1423689023"> 2. <text> <u>"getConditionTransition"</u></text> 3. <cond > 4. <text tool="CPN Tools "version="2.9.11"> 5. <u>"true"</u> </text> 6. </cond> 7. </trans> 8. <place id="ID1423689035"> 9. <text> <u>Item name</u> </text> 10. <type id="ID1423689036"> 11. <text tool="CPN Tools" version="2.9.11"> 12. <u>Item Type</u> </text> 13. </type> 14. <initmark id="ID1423689037"> 15. <text tool="CPN Tools" version="2.9.11"/> 16. </initmark> 17. </place> 18. <place id="ID1423689036"> 19. <text> <u>flow</u> </text> 20. <type id="ID1423689036"> 21. <text tool="CPN Tools" version="2.9.11"> 22. <u>Role</u>. </text> 23. </type> 24. <initmark id="ID1423689039"> 25. <text tool="CPN Tools" version="2.9.11"/> 26. </initmark> 27. </place> 28. <arc id="ID1424199627" 29. orientation="PtoT" order="1"> 30. <transend idref="<u>getConditionTransition ID</u>"/> 31. <placeend idref="<u>Item Place ID</u>"/> 32. <annot id="ID1424199628"> 33. <text tool="CPN Tools" version="2.9.11">

Table 6.9 (a):LCC-CPNXML Transformation Table (Get an Argument Condition)

LCC Code (Get condition)	← GetConditions
CPNs Model	CPNXML Structure
	<p>34. <u>Item arguments</u></p> <p>35. </text></p> <p>36. </annot></p> <p>37. </arc></p> <p>38. <arc id="ID1424199687"</p> <p>39. orientation="PtoT" order="1"></p> <p>40. <transend idref="<u>getGonditionTransition ID</u>"/></p> <p>41. <placeend idref="<u>flow Place ID</u>"/></p> <p>42. <annot id="ID1424199618"></p> <p>43. <text tool="CPN Tools" version="2.9.11"></p> <p>44. <u>Role arguments</u></p> <p>45. </text></p> <p>46. </annot></p> <p>47. </arc></p> <p>48. <arc id="ID1424199684"</p> <p>49. orientation="TtoP" order="1"></p> <p>50. <transend idref="<u>getGonditionTransition ID</u>"/></p> <p>51. <placeend idref="<u>Message Place ID</u>"/></p> <p>52. <annot id="ID1424199638"></p> <p>53. <text tool="CPN Tools" version="2.9.11"></p> <p>54. <u>Message arguments</u></p> <p>55. </text></p> <p>56. </annot></p> <p>57. </arc></p> <p>58. <arc id="ID1424199664"</p> <p>59. orientation="TtoP" order="1"></p> <p>60. <transend idref="<u>Message transition ID</u>"/></p> <p>61. <placeend idref="<u>Item Place ID</u>"/></p> <p>62. <annot id="ID1424149638"></p> <p>63. <text tool="CPN Tools" version="2.9.11"></p> <p>64. <u>Get Condition</u></p> <p>65. </text></p> <p>66. </annot></p> <p>67. </arc></p>

Table 6.9 (b):LCC-CPNXML Transformation Table (Get an Argument Condition)

<i>LCC Code</i> (Get condition)	← GetConditions
CPNs Model	CPNXML Structure
	<p>68. <arc id="ID1424129684" 69. orientation="TtoP" order="1"> 70. <transend idref="<u>Message transition ID</u>" /> 71. <placeend idref="<u>flow Place ID</u>" /> 72. <annot id="ID1424299638"> 73. <text tool="CPN Tools" version="2.9.11"> <i>a. <u>Role Arguments</u></i> 74. </text> 75. </annot> 76. </arc></p>

Table 6.9 (c):LCC-CPNXML Transformation Table (Get an Argument Condition)

- (4) One arc, which is used to connect the item place to the new transition ("getConditionTransition"), where the arc ID = unique identifier, the arc type= PtoT, the transition ID reference = the new transition ID, the place ID reference= the item place ID, the arc inscription = (the item arguments e.g. *Premise*) (line 27 to 37 of Table 6.9(a) and Table 6.9(b));
- (5) One arc, which is used to connect the flow place to the new transition ("getConditionTransition"), where the arc ID = unique identifier, the arc type= PtoT, the transition ID reference = the new transition ID, the place ID reference= the flow place ID, the arc inscription = (Role arguments) (line 38 to 47 of Table 6.9(b));
- (6) One arc, which is used to connect the new transition ("getConditionTransition") to the message place, where the arc ID = unique identifier, the arc type= TtoP, the transition ID reference = the new transition ("getConditionTransition") ID, the place ID reference= the message place ID, the arc inscription=(Message Arguments) (line 49 to 57 of Table 6.9(a) and Table 6.9(b));
- (7) One arc, which is used to connect the role message transition to the item place, where the arc ID = unique identifier, the arc type= TtoP, the transition ID reference = the role message transition ID, the place ID reference= the item

place ID, the arc inscription = (GetCondition) (line 58 to 67 of Table 6.9(a) and Table 6.9(b));

- (8) One arc, which is used to connect the role message transition to the flow place, where the arc ID = unique identifier, the arc type= TtoP, the transition ID reference = the role message transition ID, the place ID reference= the flow place ID, the arc inscription = (Role arguments) (line 68 to 76 of Table 6.9(c));

See Figure C.14 in appendix C which shows an example of get item from list condition CPN model.

6.1.3 Generation of a CPN Superpage

The second step for transforming an LCC protocol into the CPNXML file is to generate one CPN superpage. The CPN superpage is composed of:

- (1) More than one substitution transition (see chapter 2, section 2.3.1.2) where each substitution transition represents one LCC role.
- (2) More than one place and arc which is used to connect the CPN subpages generated in the first step to the CPN superpage and to create the CPN model of the LCC protocol. These places and arcs represent the interaction relations between roles (subpages).

The final result of this step, which is used to connect all the CPN subpages, is a high-level CPN model. The resulting CPN model is the formal representation of the LCC protocol and can be used to analyse the dynamic behaviour of the LCC protocol.

Generation of a CPN Superpage Steps

Each LCC role is transformed into a high-level Petri net by creating (as shown in Table 6.10):

- (1) One new substitution transition where the transition ID = unique identifier, the transition name= role name, subpageinfo ID = corresponding subpage ID, subpageinfo name = corresponding subpage name, and portsock= (socket ID,

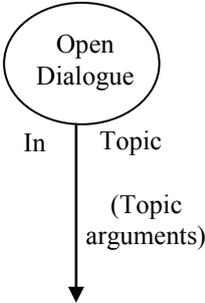
LCC (Dialogue Topic Argument)	a(RoleName(Arguments, Topic),AgentID)
LCC CPNs Model	CPNXML Structure
<p data-bbox="339 306 608 338"><i>Dialogue Topic symbol</i></p> 	<ol style="list-style-type: none"> 1. <place id="ID1423689035"> 2. <text> <i>OpenDialogue.</i> </text> 3. <type id="ID1423689036"> 4. <text tool="CPN Tools" version="2.9.11"> 5. . <i>Topic</i> </text> 6. </type> 7. <initmark id="ID1423689037"> 8. <text tool="CPN Tools" version="2.9.11"/> 9. </initmark> 10. <port id="ID1424205036" type="<i>In</i>"> 11. </port> 12. </place> 13. <arc id="ID1423689049"> 14. orientation="<i>PtoT</i>" order="1"> 15. <transend idref="<i>New substitution transition ID</i>"/> 16. <placeend idref="<i>New place ID</i>"/> 17. <annot id="ID1423689050"> 18. <text tool="CPN Tools version="2.9.11"> 19. <i>Topic arguments</i> 20. </text> 21. </annot> 22. </arc>

Table 6.11: LCC-CPNXML Transformation table (Dialogue Topic in the superpage)

b) Create one arc (input arc), which is used to connect the new place to the new substitution transition, where the arc ID = unique identifier, the arc type = PtoT (input arc), the transition ID reference = the new substitution transition ID, the place ID reference = the new place ID, the arc inscription = (Topic argument) (line 13 to 22 of Table 6.11).

(4) If this role is the agent's primary role, then:

a) One new place where the place ID = unique identifier, the place name = agent ID, the place colour set type = Role and place (port) type = In (line 1 to 12 of Table 6.12);

b) One arc (input arc), which is used to connect the new place to the role message transition of agent first role, where the arc ID = unique identifier, the arc type = PtoT (input arc), the transition ID reference = the new

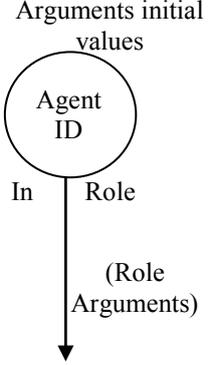
LCC Code (Starter Role Arguments)	a(RoleName(Arguments, Topic),AgentID)
LCC CPNs Model	CPNXML Structure
<p><i>Starter Role argument symbol</i></p>  <p>Arguments initial values</p>	<ol style="list-style-type: none"> 1. <place id="ID1423689035"> 2. <text> <u>Agent ID</u> </text> 3. <type id="ID1423689036"> 4. <text tool="CPN Tools" version="2.9.11"> 5. <u>Role</u> </text> 6. </type> 7. <initmark id="ID1423689037"> 8. <text tool="CPN Tools" version="2.9.11"/> 9. <u>Arguments initial values</u> 10. </initmark> 11. <port id="ID1424205036" type="In"> 12. </port> 13. </place> 14. <arc id="ID1423689049" 15. orientation="PtoT" order="1"> 16. <transend idref="New substitution transition ID "/> 17. <placeend idref="New place ID"/> 18. <annot id="ID1423689050"> 19. <text tool="CPN Tools" version="2.9.11"> 20. <u>Role arguments</u> 21. </text> 22. </annot> 23. </arc>

Table 6.12: LCC-CPNXML Transformation table (Agent's Starter Role Arguments in superpage)

substitution transition ID, the place ID reference = the new place ID, the arc inscription = (Role arguments) (line 13 to 23 of Table 6.12)

Appendices A and C illustrate detailed examples of CPN subpages and the superpage of a negotiation dialogue and a persuasion dialogue, respectively.

6.2 Step Two: Construction of State Space

The second step of the verification method is to construct state space. In the CPN Tool, state spaces can be constructed by:

- (1) Using the following CPN SML functions:

CalculateOccGraph();

CalculateScgGraph();

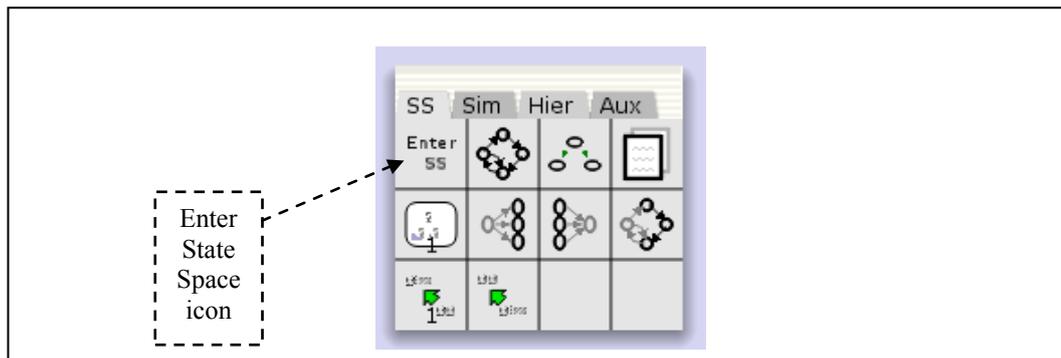


Figure 6.2: State Space Tool Palette

- (2) Or, using the CPN State Space (SS) tool palette: constricting the state space is simple. The user needs to:
- a) Open the CPN Tool;
 - b) Select the state space tool palette (as shown in Figure 6.2);
 - c) Select the Enter State Space (Enter SS) in the SS tool palette, and apply it to one of the pages in the CPN model.

For more information about using the state space tools see [Jensen *et al.*, 2002]. In our approach, the user can construct the state space of the generated CPNXML file (the generated CPN model) using the CPN state space tool palette (see chapter 8, section 8.3.2).

Appendices A and C illustrate detailed examples of the State Spaces of the CPN models corresponding to a negotiation dialogue and a persuasion dialogue, respectively.

The State Space Explosion Problem in the CPN Tool

In general, verification techniques suffer from state space explosion problem [Ding and Su, 2008]. The main reason for this problem is running out of memory before finishing to compute the state space of a complex model.

Ding and Su [Ding and Su, 2008] compares different techniques for dealing with the state space explosion problem in the CPN Tool. In this thesis we did not deal with this problem.

However, we cannot guarantee that our verification method will not encounter a state space explosion problem. In fact, the generated CPN model could obtain an infinite number of state space nodes which cause the state space explosion. This is because the CPN model could be defined for finite number of agents (e.g. two agents) but still the agents could be involved in infinite loops. Consequently, we cannot guarantee that there will be absolutely no state space explosion in our verification model process.

6.3 Step Three: Automated Creation of DID Properties Files

The third step of the verification method is to create a DID properties file. The extraction of the protocol properties from the DID diagram and the creation of DID properties files are automatic. These files can be used by our tool to obtain all the information about the behaviour of the DID diagram (e.g. Starting message information). When the tool dynamically generates each file, it uses the property name as the file name and stores the file on the tool path.

Nine property files are automatically created by our tool:

- (1) Possible Locutions file: contains the set of permitted messages;
- (2) Reply Locutions file: contains the set of legal reply locutions in terms of the available moves that an agent can select to follow on from the previous move;
- (3) Starting Locutions file: contains message names which are used to begin the dialogue;
- (4) Intermediate Locutions file: contains message names which are used to remain in the dialogue,

- (5) Termination Locutions file: contains message names which are used to terminate the dialogue;
- (6) Termination Locutions Effect CS and Effective CS files: contain the effect of the termination message to the sender commitment store *CS*;
- (7) Player Types file: contains dialogue game player types (e.g. opponent or proponent);
- (8) Player IDs file: contains dialogue game player IDs;
- (9) Termination Role Names file: contains player termination role names.

6.4 Step Four: Applying Verification Model

In step two we explained how to construct a state space graph and in step three we explained how to create DID property files. Therefore, the next task is to automatically verify the DID properties over the synthesised LCC protocol represented as a state space graph.

Verification Model Properties

The verification process is carried out by checking five basic properties, which are independent of any dialogue games types:

- (1) Dialogue opening property: to check that the LCC protocol begins with a proper Starting Locution;
- (2) Termination of a dialogue property: to determine if the LCC protocol terminates with a proper Termination Locution;
- (3) Turn taking between agents property: to guarantee that in the LCC protocol the turn-taking switches to the next agent after the current agent sends a message;
- (4) Message sequencing property: to check that the LCC protocol message exchange respects the DID;

- (5) Recursive message property: to verify that the LCC protocol recurs when an agent sends a message with an Intermediate DID Locution.

In general, to verify each property, we use the following approach:

- (1) Create a new text file for each property and use the property name as the file name;
- (2) Extract the needed information from the state space graph and write this information in the property text file;
- (3) Get the information of a DID diagram from the DID property file (created in the previous step three);
- (4) Call the CPN SML property function, where the function inputs are the DID diagram information (DID property file) and the LCC protocol state space information (property text file);
- (5) Create a new text file (property result file) and write the CPN SML property function result in the property result file;
- (6) Repeat steps 1 to 5 for each property;
- (7) Present a report to the user indicating which properties are satisfied and which are unsatisfied.

The following subsections give a detailed description of each of these properties as well as the corresponding CPN SML function.

Property-1 Dialogue Opening

This property should guarantee that the LCC protocol will start if, and only if, a proposal agent sends a Starting DID Locution. Figure 6.3 shows the CPN SML specification of this property:

- (1) Line 1: Read the state space graph information from the Property1 text file and save this information in the *SS* variable.

```

1. Read&Save   SS=State Space information
2. Read&Save   DIDOpenDialogueMessages =DID information
3. Call       CheckProperty1
4. Input      (SS,DIDOpenDialogueMessages)
5. Extract    message1
6.   val checkODM =
7.     compare(DIDOpenDialogueMessages,message1)
8.     if (checkODM ) then
9.       "Property 1(Dialogue opening) is Satisfied"
10.    else
11.      "Property 1(Dialogue opening) is not Satisfied"
12. end       CheckProperty1
13. Create&Save Property1 result file

```

Figure 6.3: Property 1 as an SML Function

- (2) Line 2: Read the information of a DID diagram from the Starting Locutions' DID property file and save this information in the *DIDOpenDialogueMessages* variable.
- (3) Line 3: Call *CheckProperty1* function.
- (4) Line 4: *CheckProperty1* function inputs are *SS* and *DIDOpenDialogueMessages*.
- (5) Line 5: Extract the first message from the *SS* (*message1*)
- (6) Lines 6 and 7: Compare the first exchanged message in the state space graph with the Starting Locution from the DID where:
 - a) *compare* function is used to compare the first message;
 - b) *checkODM* variable represents the *compare* function result. It is considered true if the first message in the state space graph is the same as the Starting Locution of the DID.
- (7) Lines 8 to 11: Check the result of the comparison. A positive (negative) result indicates that Property 1 is satisfied (unsatisfied).

- (8) Line 13: Create a Property1 result file and write the result of *CheckProperty1* in this file.

Property-2 Termination of a Dialogue

This property should guarantee that the LCC protocol will end when an agent sends a DID Termination Locution. It should also check that the agent's commitment store has changed properly after termination, and that the role of the agent that finishes the dialogue is the expected one (based on the recorded sequence of moves). Figure 6.4 shows the algorithm of the CPN SML specification of this property:

- (1) Line 1: Read the state space graph Termination nodes information from the Property2 text file and save this information in *TNodes* variable.
- (2) Line 2: Read the DID termination messages information from the Termination Locutions and the Effective CS DID property files and save this information in the *TDID* variable.
- (3) Line 3: Call function *CheckProperty2*.
- (4) Line 4: Function inputs are *TNodes* and *TDID*.
- (5) Line 5: Extract the needed information from *TNodes* where:
 - a) *message* represents termination message;
 - b) *topic* represents dialogue topic;
 - c) *premise* represents dialogue topic premises;
 - d) *sender* represents termination message sender ID;
 - e) *receiver* represents termination message receiver ID;
 - f) *sCS* represents sender commitment store;
 - g) *rCS* represents receiver commitment store;

```

1. Read&Save  TNodes = state space termination nodes information
2. Read   TDID =DID termination nodes information
3. Call   CheckProperty2
4. Input  (TNodes,TDID)
5. Extract (message, topic , premise, sender, receiver, sCS, rCS)
6. Extract (DIDTL, DIDEf, DIDAID,DIDS)
7. val checkSR = checkSenderReceiver(message,sender,receiver,
8.                                     opponent,proponent,DIDAID,DIDS)
9. val csContant = checkTheContantofCS(role, message, rCS,topic,premise,rCSsize,
10.                                     topicSize,premiseSize, DIDTL,DIDEf)
11. val lengthofRest= length restStateSpace
12. if (lengthofRest >= 4) andalso (csContant= true) andalso (checkSR=true) then
13.     CheckPropert2(restStateSpace, DID)
14.     else
15.         if (csContant) andalso (checkSR ) then
16.             "Property 2(Termination of a Dialogue) is Satisfied"
17.         else
18.             if not (csContant) then
19.                 "Property 2(Termination of a Dialogue) is not Satisfied: There is a
20.                 problem in the agent's commitment store"
21.             else
22.                 "Property 2(Termination of a Dialogue) is not Satisfied: There is a
23.                 problem in the who to terminated the dialogue"
24. End CheckProperty2
25. Create&Save Property2 result file

```

Figure 6.4: Property 2 as an SML Function

h) *proponent* represents the sender agent in the initial node (the sender agent ID of the first role in the LCC code which is responsible for opening the dialogue);

i) *opponent* represents the receiver agent in the initial node (the receiver agent ID of the second role in the LCC code which is responsible for receiving the opening [starting] dialogue message).

(6) Line 6: Extract one termination message information from the *TDID* where:

- a. *DIDTL* represents the expected termination message for the specific role;
- b. *DIDef* represents the effect of the termination message to the sender commitment store *CS* (e.g. *DIDef*= "Add Topic");
- c. *DIDAID* represents the expected agent ID of the termination message sender;
- d. *DIDS* represents the expected agent type (e.g. opponent or proponent) of the termination message sender.

(7) Lines 7 and 8: Check that the sender and the receiver of the termination message in the state space are the expected sender and receiver. Then compare the sender and receiver of the termination message in the state space with the sender and receiver of the same termination message in the DID where:

- a. *checkSenderReceiver* function is used to compare the sender and receiver of the termination message;
- b. *proponent* and *opponent* variables are used to check the expected values of the sender and receiver (which agent must send this message and which agent must receive this message);
- c. *checkSR* variable represents the *checkSenderReceiver* function result. It is considered true if the sender and receiver of the termination message in the State Space are identical to the sender and receiver of the same termination message in the DID.

(8) Lines 9 and 10: Compare the content of the CS in the termination message of the sender agent in the state space with the content of the same termination message of the sender agent in the DID where:

- a. *checkTheContentofCS* function is used to compare the content of the CSs;

- b. *csContant* represents the *checkTheContantofCS* function result. It is considered true if the content of the CS in the termination message of the sender agent in the state space is identical to the content of the same termination message of the sender agent in the DID.
- (9) Lines 11 to 13: Check if there is another termination node in the state space; then recall the *CheckPropert2* function.
 - (10) Lines 14 to 23: Check the result of the comparison. A positive (negative) result indicates that Property 2 is satisfied (unsatisfied).
 - (11) Line 25: Create Property2 result file and write the result of *CheckProperty2* in this file.

Property-3 Turn Taking between Agents

This property checks that in the LCC protocol the turn-taking between agents switches after each move (after an agent sends a message). Figure 6.5 shows the algorithm of the CPN SML specification of this property:

- (1) Line 1: Read the state space graph information from the Property3 text file and save this information in *SS* variable.
- (2) Line 2: Call function *CheckPropert3AllTN*.
- (3) Line 3: Function input is *SS*.
- (4) Line 4: Extract the arcs information from the *ArcsList*. *ArcsList* represents all arcs information in the *SS*.
- (5) Line 5: Function *CheckPropert3AllTN* calls the function *checkProperty3Part1* which is used to check the turn-taking between agents by comparing the state space nodes information. It compares two nodes at one time. It compares the odd numbers of the nodes since every two nodes represent the sender and the receiver function of the same locution (message). It begins by comparing node

```

1. Read&Save SS= state space graph information
2. Call CheckPropert3AllTN
3. Input (SS)
4. Extract (ArcsList)
5. Call turnTaking = checkProperty3Part1
6.     Input (ArcsList)
7.     Extract (n1,role1, senderM1, receiverM1, n2,role2, senderM2, receiverM2)
8.     val restLength= length restArcsList
9.     if (restLength >= 3) andalso (not (role1 = role2))
10.        andalso (senderM1 = receiverM2) andalso (receiverM1 = senderM2)
11.        then checkProperty3Part1(restArcsList)
12.    else
13.        if (restLength >= 3) andalso ((role1 = role2))
14.            andalso ((senderM1 = senderM2) andalso (receiverM1 = receiverM2 ))
15.            then checkProperty3Part1(restArcsList)
16.        else
17.            if (not (role1 = role2)) andalso (senderM1 = receiverM2)
18.                andalso (receiverM1 = senderM2)
19.                then true
20.            else
21.                false
22.        End checkProperty3Part1
23. Return Back to CheckPropert3AllTN
24. if (turnTaking= true) then
25.     "Property 3(Turn Taking) is Satisfied"
26. else
27.     "Property 2(Turn Taking) is not Satisfied"
28. end CheckPropert3AllTN
29. Create&Save Property3 result file
    
```

Figure 6.5: Property 3 as an SML Function

1 with node 3. Note that the result of function *checkProperty3Part1* is saved in the *turnTaking* variable.

(6) Line 6: Function *checkProperty3Part1* input is *ArcsList*.

(7) Line 7: Extract two nodes' information from *ArcsList* where:

- a) *n1* represents the first node;
 - b) *role1* represents the role name of the first node;
 - c) *senderM1* represents the sender agent ID of the first node;
 - d) *receiverM1* represents the receiver agent ID of the first node;
 - e) *n2* represents the second node;
 - f) *role2* represents the role name of the second node;
 - g) *senderM2* represents the sender agent ID of the second node;
 - h) *receiverM2* represents the receiver agent ID of the second node;
- (8) Line 8: Get the lengths of the remaining nodes information in the *restArcsList* and save it in *restLength*.
- (9) Lines 9 and 21:
- a) Compare the first node's information (*role1*, *senderM1* and *receiverM1*) with the second node's information (*role2*, *senderM2* and *receiverM2*);
 - b) If there are other nodes in the *restArcsList*, then recall the *checkProperty3Part1* function (recurs).
- (10) Line 23: Return the control back to *CheckProperty3AllTN* function.
- (11) Lines 24 to 27: Check the result of the comparison (*turnTaking* variable). A positive (negative) result indicates that Property3 is satisfied (unsatisfied).
- (12) Line 29: Create the Property3 result file and write the result of *CheckProperty3AllTN*.

Property-4 Message Sequence

This property is used to verify that the LCC protocol message exchange respects the DID. For instance, for the DID depicted in Figure 4.3 in chapter 4 one thing that should be checked is that after an agent makes a claim the other agent can only answer with a "concede" or a "why" locution. Figure 6.6 shows the CPN SML specification of this property:

- (1) Line 1: Read the state space graph information from the Property4 text file and save this information in the *SS*.
- (2) Line 2: Read the information of the DID diagram from the Possible Locutions and Reply Locutions DID properties files and save this information in the *DIDPosM* and *DIDRepM* where:
 - a) *DIDPosM* represents the set of possible locutions in the DID;
 - b) *DIDRepM* represents the set of legal reply locutions in the DID.
- (3) Line 3: Call function *CheckPropert4* which is used to compare the message exchange in the *SS* with the message sequence in the DID (*DIDPosM* and *DIDRepM*).
- (4) Line 4: Function inputs are *SS*, *DIDPosM* and *DIDRepM*.
- (5) Line 5: Extract the arcs information from *SS*. *AllArcs* represents the All arcs information in the *SS*.
- (6) Line 6: Compare the message sequence in the state space graph (*AllArcs*) with the message sequence in the DID (*DIDPosM* and *DIDRepM*) where:
 - a) *checkMessageS* function is used to compare messages;
 - b) *messageSeq* represents the *checkMessageS* function result. It is considered true if the message sequence in the state space graph is identical to the message sequence in the DID.

```

1. Read&Save SS= state space information
2. Read&Save (DIDPosM,DIDRepM)
3. Call CheckPropert4
4. Input (SS, DIDPosM,DIDRepM)
5. Extract (allArcs)
6. Val messageSeq = checkMessageS(allArcs,DIDPosM,DIDRepM)
7. if (messageSeq= true) then
8.     "Property 4(Message Sequence) is Satisfied"
9. else
10.    "Property 4(Message Sequence) is not Satisfied"
11. end CheckPropert4
12. Create&Save Property4 result file
    
```

Figure 6.6: Property 4 as an Standard ML Function

(7) Lines 7 to 10 are used to check the result of the comparison. A positive (negative) result indicates that Property 3 is satisfied (unsatisfied).

(8) Line 12: Create the Property4 result file and write the result of *CheckProperty4* in this file.

Property-5 Recursive Message

This property is defined to verify that the LCC protocol recurs when an agent sends a message with an Intermediate DID Locution. Figure 6.7 shows the CPN SML specification of this property:

(1) Line 1: Read the state space graph information from the Property5 text file and save this information in *SS*.

(2) Line 2: Read the DID recursive locution information from the Intermediate Locutions DID property file and save this information in *DIDRecursiveMessages*.

(3) Line 3: Call function *CheckProperty5*. This function gets the expected recursive locutions from DID and attempts to prove that these locutions are also recursive locutions in the state space by proving the following:

```

13. Read&Save   SS= state space information
14. Read&Save DIDRecursiveMessages
15. Call CheckProperty5
16. Input (SS, DIDRecursiveMessages)
17. Extract (Openingmessage,TNodes)
18. val checkopeningDM = findElementInTheList(DIDRecursiveMessages,
19.                                           Openingmessage)
20. val checkTerminationM =
21.     checkAllTerminatedMessags(DIDRecursiveMessages,TNodes)
22. if (not (checkopeningDM )) andalso (not (checkTerminationM)) then
23.     "Property 5(Recursive Message) is Satisfied"
24. else
25.     "Property 5(Recursive Message) is not Satisfied"
26. end CheckProperty5
27. Create&Save Property5 result file
    
```

Figure 6.7: Property 5 as an Standard ML Function

- a) The target locution is not the starting or opening locution in the state space;
- b) The target locution is not the terminating locution in the state space.

(4) Line 4: Function inputs are *SS* and *DIDRecursiveMessages*.

(5) Line 5: Extract the starting locutions information from *SS* and save this information in *Openingmessage*. Then extract the termination locutions information from *SS* and save this information in *TNodes*.

(6) Lines 6 and 7: Check if the recursive locution in the DID is a Starting Locution in the state space, where:

- a) *findElementInTheList* function is used to check if the recursive locution in the DID is a Starting Locution in the state space;
- b) *checkopeningDM* represents the *findElementInTheList* function result. It is considered true if the recursive locution in the DID is a Starting Locution in the state space.

(7) Lines 8 and 9: Check if the recursive locution in the DID is a Termination Locution in the state space, where:

- a) *checkAllTerminatedMessages* function is used to check if the recursive locution in the DID is a Termination Locution in the state space;
- b) *checkTerminationM* represents the *checkAllTerminatedMessages* function result. It is considered true if the recursive locution in the DID is a Termination Locution in the state space.

(8) Lines 10 to 13: Check the result of the comparison(*checkopeningDM* and *checkTerminationM*). A positive (negative) result indicates that Property5 is satisfied (unsatisfied).

(9) Line 15: Create Property5 result file and write the result of *CheckProperty5* in this file.

These five properties are provided by our verification model system. However, the system allows users to add and run more properties. Appendix A shows more properties, which are different from these five properties and are dependent on the dialogue types.

6.5 Summary

This chapter has explained how we perform the automatic validation of LCC protocols based on their DID properties. It describes in detail the four stages of the verification model approach: (1) automatically transforming the LCC specification into an equivalent CPNXML file; (2) construction of state space graph from the resulting CPNXML file; (3) automatically creating DID properties; (4) automatically verifying the satisfaction of the CPN SML specification in the state-space graph computed from the LCC protocol by applying a verification model. The proposed validation tool can be used to analyse the correctness of LCC.

As proof of this concept, in the next chapter we will describe the implemented LCC argumentation protocol automated synthesis and validation tool.

Chapter 7

Design and Implementation

This chapter ties together all of the separate sections of the thesis. It discusses the architecture of our systems and the implementation of the *GenerateLCCProtocol* tool that has been developed as part of this thesis. As explained in chapters 5 and 6, this tool enables the user to automatically generate LCC protocols from DID specifications, along with semi-automatically checking the correctness of the generated LCC protocols.

As shown in Figure 7.1, the *GenerateLCCProtocol* tool receives a DID as an input and returns:

- (1) The LCC argumentation agent protocol resulting from applying LCC–Argument patterns over the DID given as input (as explained in chapters 4 and 5).
- (2) The result of verifying if the resulting LCC protocol satisfy the DID properties (as explained in chapter 6).

This chapter begins by providing a brief overview of the system architecture in section 7.1. Section 7.2 discusses, in detail, an example of use of the tool. Lastly, Section 7.3 summarises this chapter.

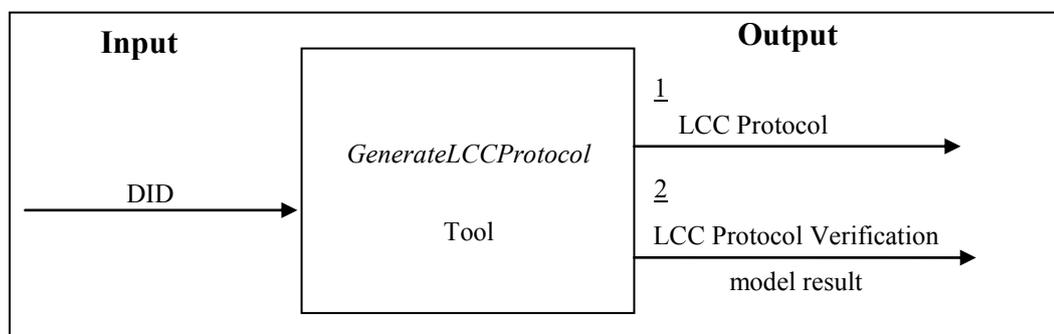


Figure 7.1: *GenerateLCCProtocol* Tool

7.1 Architecture

The synthesis protocol tool (*GenerateLCCProtocol* tool) has been designed and implemented in the Java programming language. The tool consists of two parts, as shown in Figure 7.2.

7.1.1 Part One: Synthesis of Concrete Protocols Architecture

Part one of the thesis architecture (as shown in Figure 7.2) is used to bridge the gap between AIF and LCC using transformational synthesis. Part one, explained in detail in chapters 4 and 5, was built in two stages:

- (1) Specification of multi-agent protocols in a new high level control flow specification language called Dialogue Interaction Diagram (DID), which extends the AIF. The DID is provided in chapter 4;
- (2) Automatic synthesis of concrete LCC protocols from DID specifications by recursive application of LCC-Argument patterns. The fully automated synthesis is provided in chapter 5.

7.1.2 Part Two: Verification Model Architecture

Part two of the system architecture (as shown in Figure 7.2) provides a verification methodology based on CPN and SML language to verify the semantics of the DID specification against the semantics of the synthesised LCC protocol. The verification methodology is provided in chapter 6. It was built in four stages:

- (1) Automatically transforming the LCC specification (the resulting LCC protocol from part one) into an equivalent Coloured Petri Net (CPN). The formal semantics of the CPN model allows us to prove that certain (un)desirable properties are (un)satisfied in the LCC protocol. The proof of properties in the LCC protocols mapped into CPNs is supported by a state-space technique, which is used to compute exhaustively all possible execution states;
- (2) Manual construction of the state space by the user (as explained in chapter 6);

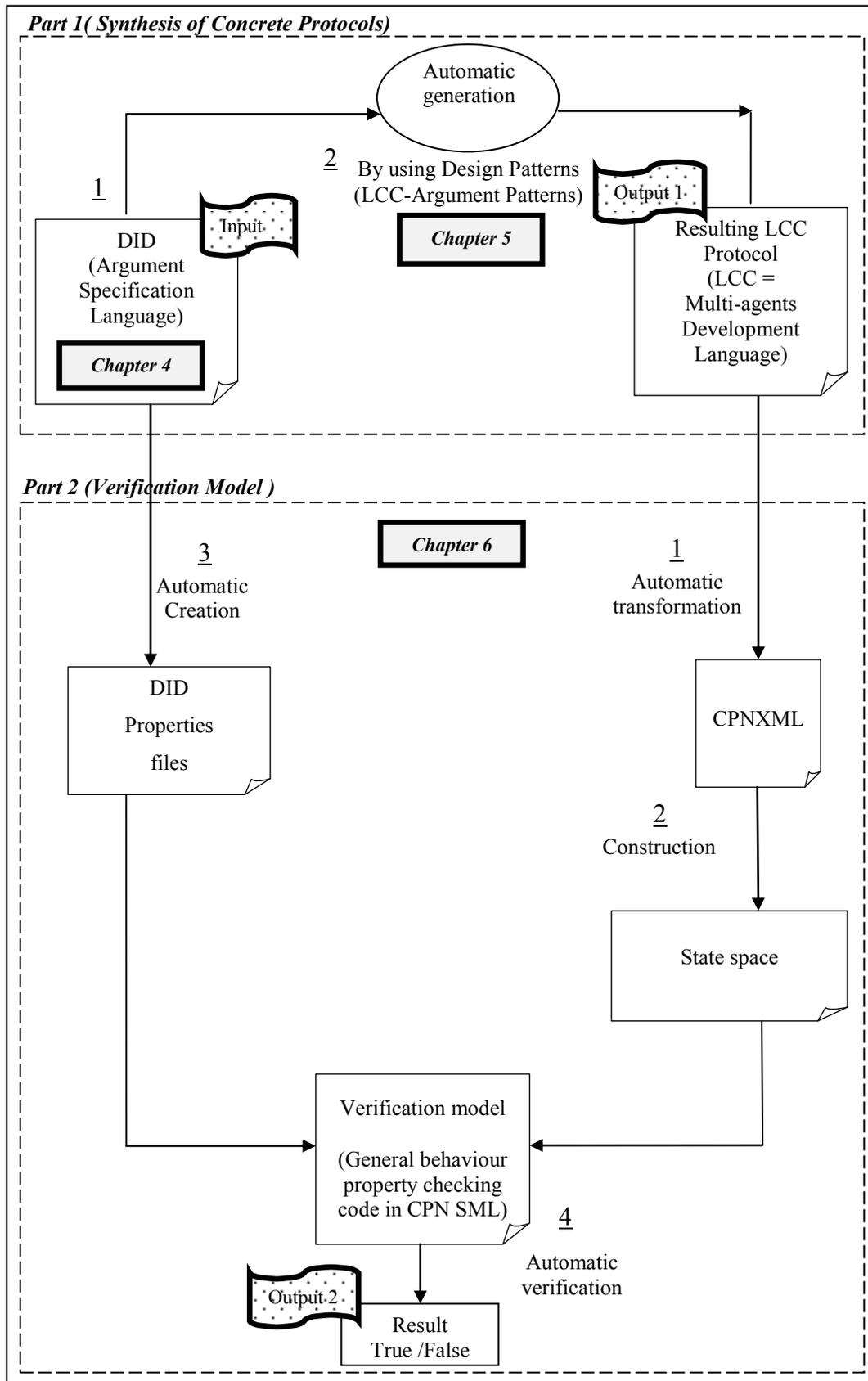


Figure 7.2: Overall Architecture

- (3) Automatically creating DID (DID diagram from part one) property files;
- (4) Automatically verifying the satisfaction of the CPN SML specifications in the state-space graph computed from the LCC protocol.

7.2 An Example Scenario

This section presents an example scenario (Figure 7.3) which demonstrates how, by using the *GenerateLCCProtocol* tool, the process of creating a DID diagram, the process of synthesising concrete LCC protocols and the verification process can be applied. This sections does not provide details of the underlying implementation. For more information about the *GenerateLCCProtocol* tool and to see the options in each windows, please see appendix E.

1. Creating Dialogue Interaction Diagram Process

In order to create a DID diagram for a persuasion dialogue (see chapter 3, section 3.4), the user needs to use the *create new DID diagram* screen (as shown in Figure 7.4). Using this screen, the user can create the DID by writing one piece of locution icon information at a time:

- (1) The first step is to identify the persuasion dialogue game locutions: there are five locutions: *claim*, *argue*, *why*, *concede* and *retract*;
- (2) The next step is to write one piece of locution icon information beginning from the locution in the top of the DID. In this example, we must begin with *claim* (as shown in Figure 7.4):
 - a) Locution Type = *Starting*. Note that if locution Type= *Intermediate* or *Termination*, the user has to select one locuion from the 'reply to' locution list (structural rules which represent the previous locution name) (as shown in Figure 7.5);
 - b) Locution Name= *claim(T)*;

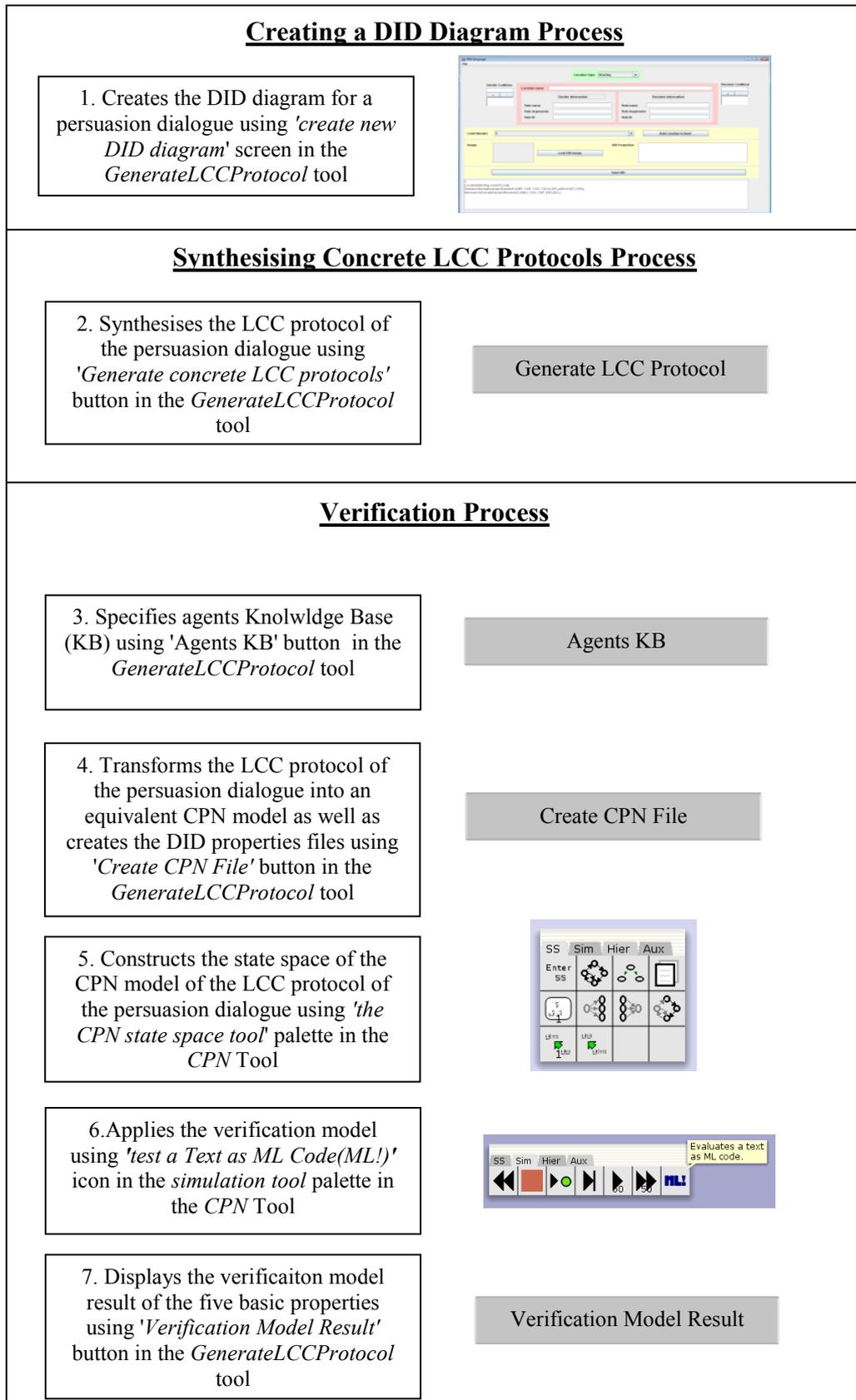


Figure 7.3: An Example Scenario of *GenerateLCCProtocol* Tool

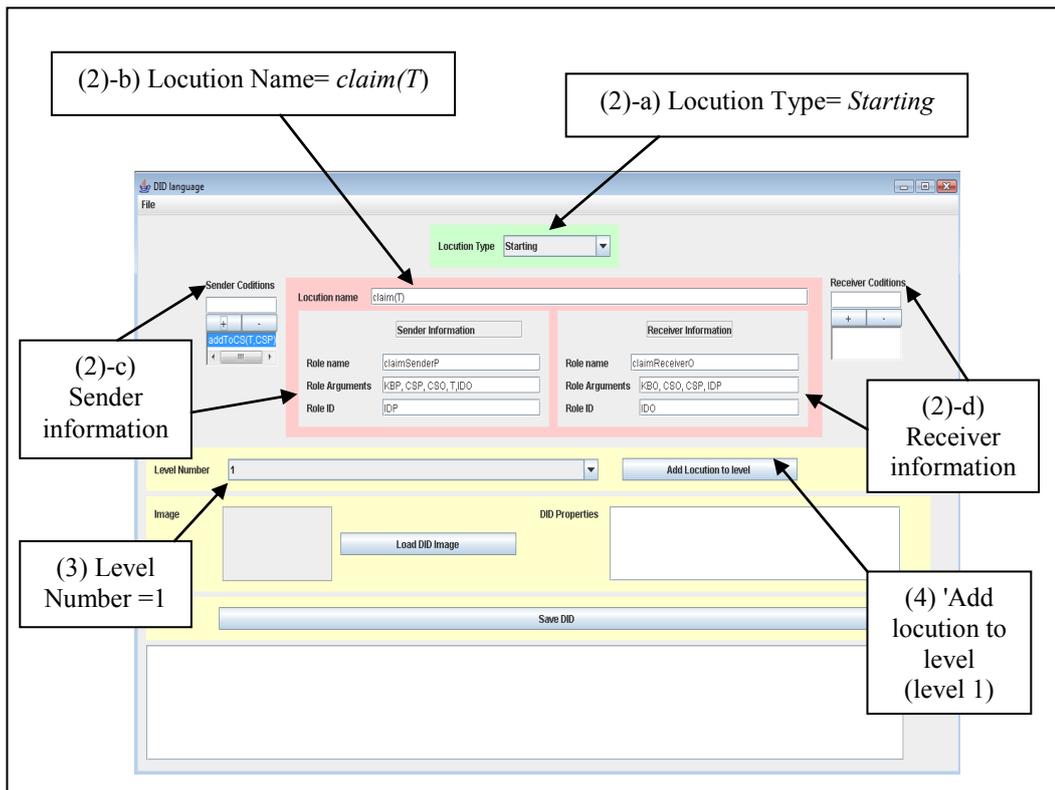


Figure 7.4: Create New Dialogue Interaction Diagram Example (Claim Locution Icon)

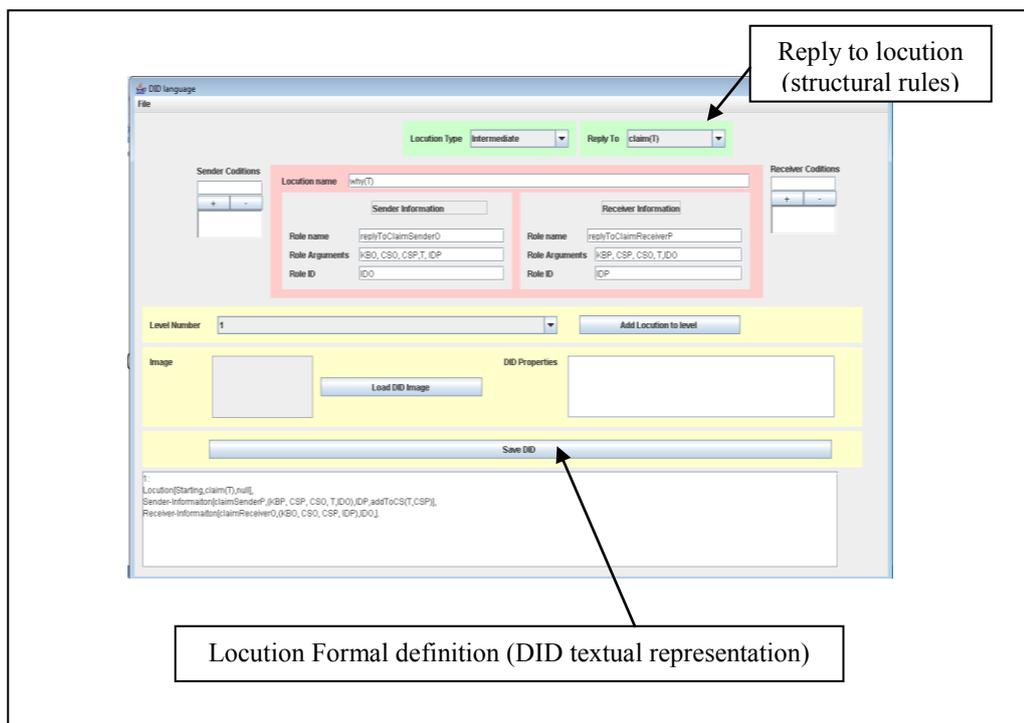


Figure 7.5: Create New Dialogue Interaction Diagram Example (Add Locution Formal Definition to DID)

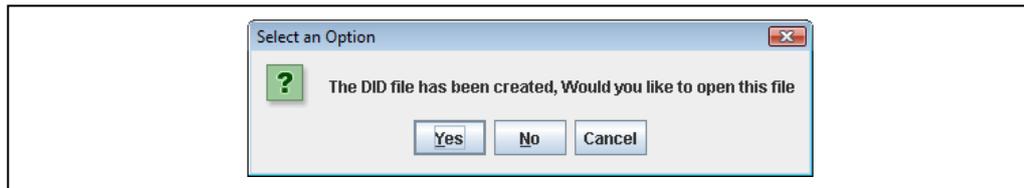


Figure 7.6: Open DID File Dialogue Box

- c) Sender information: Role name = $claimSender_P$; Role arguments = $KB_P, CS_P, CS_O, T, ID_O$; Agent ID = ID_P and Role conditions = $addTopicToCS(T, CS_P)$.
- d) Receiver information: Role name = $claimReceiver_O$; Role arguments = KB_O, CS_O, CS_P, ID_P ; Agent ID = ID_O ; and Role conditions = $null$.
- (3) Following this, we must select a locution level number (in this example, select 1);
- (4) After that, we click on 'Add locution to level' button. Note that clicking on this button adds the locution icon's information to the DID textual representation (as shown in Figure 7.5). See appendix E for more information about the DID textual representation;
- (5) Then, we move to the next locution icon and repeat steps 2, 3 and 4 (section 4.2.3 in chapter 4 describes in detail a persuasion dialogue)
- (6) Finally, when adding the last locution icon in the DID (see appendix E):
- a) Write the DID's properties in the properties text field;
 - b) Load the DID image by clicking on the 'Load DID image' (if there is an image or graphical representation for this dialogue);
 - c) Click on 'Save DID' button to save the DID. When the user clicks on this button a dialogue box will appear asking whether the user would like to open the DID file (see Figure 7.6). The DID file's textual representation screen will appear when the user clicks on 'Yes' button (see Figure 7.7).

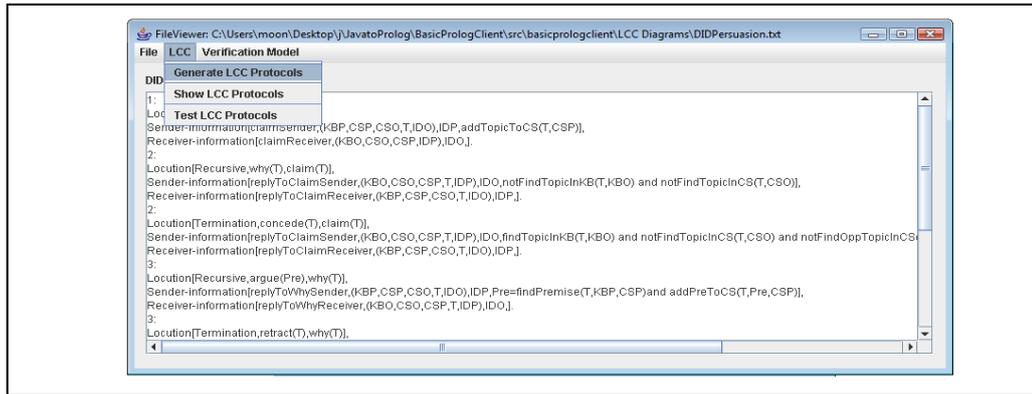


Figure 7.7: The DID Textual Representation of the Persuasion Dialogue

2. *Synthesising Concrete LCC Protocol Process*

In order to synthesise LCC protocol from the DID of the persuasion dialogue by recursively applying the LCC-Argument patterns, the user needs to click on the 'Generate LCC Protocol' button (on the LCC menu bar in the DID textual representation screen of a persuasion dialogue in Figure 7.7). See appendix E for more information.

In this example, when the user clicks on the 'Generate LCC Protocol' button, the tool will ask the user for an LCC protocol file name and then generate the LCC protocol. After that the LCC file dialog box will appear. The user has to click on the 'Yes' button to display the generated LCC protocol (as shown in Figure 7.8). This process is fully automatic (requiring no human assistance). The LCC-Argument patterns and the automated synthesis process are exhibited in chapter 5 and appendix C gives a detailed description of how to transfer a DID of a persuasion dialogue to an LCC protocol by using LCC-Argument patterns.

3. *Verification Process*

In order to verify the generated LCC protocol of the persuasion, the user needs to:

1-Specify agents' Knowledge Base (KB)

In order to verify the generated LCC protocol, the tool needs to work with a specific example. In other words, the user must provide the tool with the agents Knowledge Base (KB).

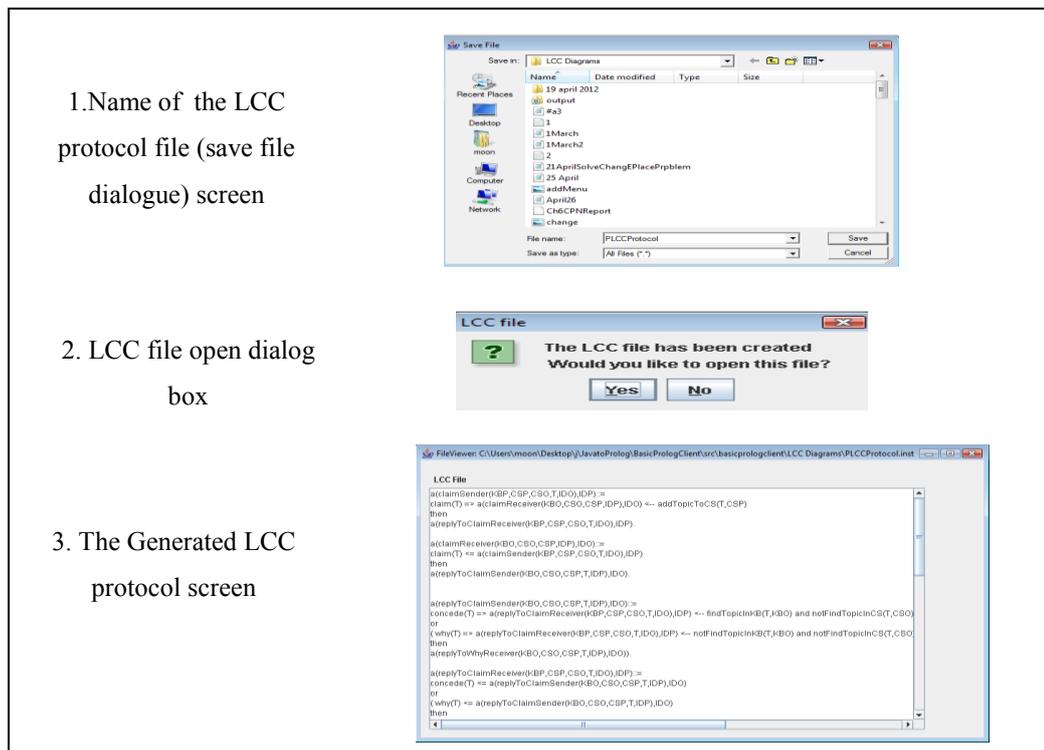


Figure 7.8: Synthesis of LCC Protocol of the Persuasion Dialogue

In this example, when the user clicks on the 'Agents KB' button (on the *Verification Model* menu bar in the DID textual representation screen of a persuasion dialogue in Figure 7.9), the tool will show a message dialogue screen which informs the user when he/she is able to add the agent's KB information. The user has to click on the 'Ok' button to display the Agent Knowledge Base screen (as shown in Figure 7.9). Then, the user has to add the knowledge base (add one element at a time to the agent KB list) for both agents (agent 1 and agent 2). After that, the user has to click on the 'Add Agent1 and Agent 2 KB' button to save the KB list for both agents (as shown in Figure 7.9). In this example, the agent1's KB= ["The car is safe", "it has an airbag"]) and the agent2's KB= ["it has an airbag", "The car is safe"]) (see Appendix C).

Please note that the user can only add agen's KB lists using *GenerateLCCProtocol* Tool before creating the CPN file. Otherwise, the user can add the agent's KB list manually using the CPN Tool (edit the *initial marking* of the role argument places). See Jensen *et al.* [Jensen *et al.*, 2007] for more information about place *initial marking*.

Bridging the Specification Protocol Gap in Argumentation

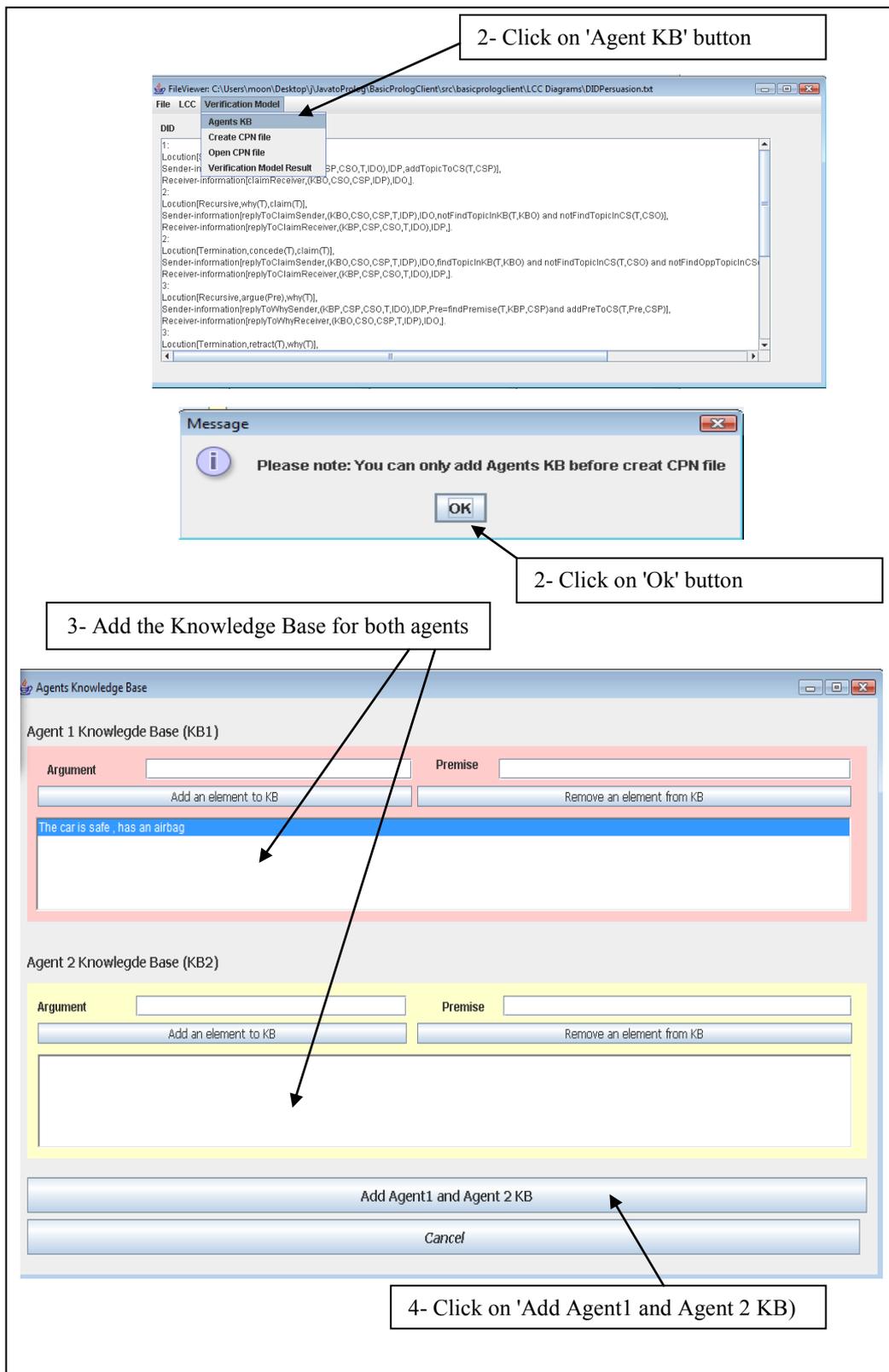


Figure 7.9: Specifying Agents Knowledge Base Screens

2- Transform the LCC Protocol into an Equivalent CPN Model

To transform the LCC protocol of the persuasion dialogue into an equivalent CPN model (CPNXML file), as well as to create the DID properties files, the user must click on the '*Create CPN File*' button (on the *Verification Model* menu bar in the DID textual representation screen of a persuasion dialogue in Figure 7.10).

In this example, when the user clicks on the '*Create CPN File*' button, the tool will ask the user for a CPN model file name and then generate the CPN model (CPNXML) file as well as the DID property files (see chapter 6 and appendix C). After that the CPN model dialogue will appear. The user has to click on the 'Yes' button to display the topic input dialogue (as shown in Figure 7.10).

Following this action the user must enter the topic. Then, the CPN model opens a dialog box. This box asks the user if he/she would like to open the CPN model file. The generated CPN model file screen will appear when the user clicks on 'Yes' button. This process is fully automatic. The automated transformation of an LCC protocol into an equivalent CPN model (CPNXML file) is examined in chapter 6 and appendix C gives a detailed description of how to transfer an LCC protocol of a persuasion dialogue to a CPN model.

3- Construct the State Space of the CPN Model

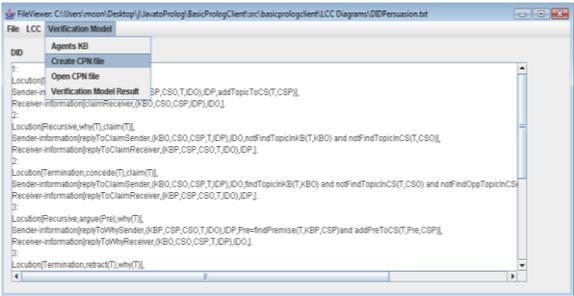
After creating the CPN model file, the user needs to click on the 'Instruction' button in the Generated CPN model (CPNXML file) screen in Figure 7.10. An instruction screen (see Figure 7.11) will appear asking the user to perform eight manual steps in order to construct the state space and to apply the verification model.

In order to construct the state space of the CPN model of the persuasion dialogue, the user needs to follow the first four steps which appears in the instruction screen (see chapter 6, section 6.2.1 and appendix C):

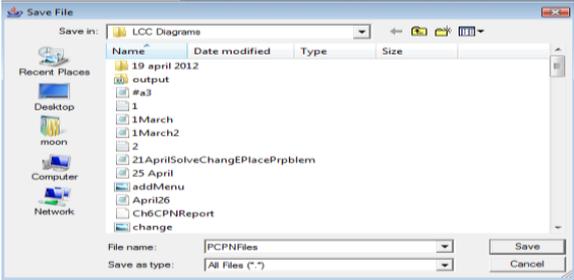
- (1) Open CPN Tool.
- (2) Open CPN file of the generated LCC file.

Bridging the Specification Protocol Gap in Argumentation

1. The 'Create CPN File' Button on the Verification Model Menu Bar



2. Name of the CPN model file (save file dialogue) screen



3. CPN model dialog box



4. Topic input dialog box



5. CPN model open dialog box



6. The Generated. CPN model (CPNXML file) screen

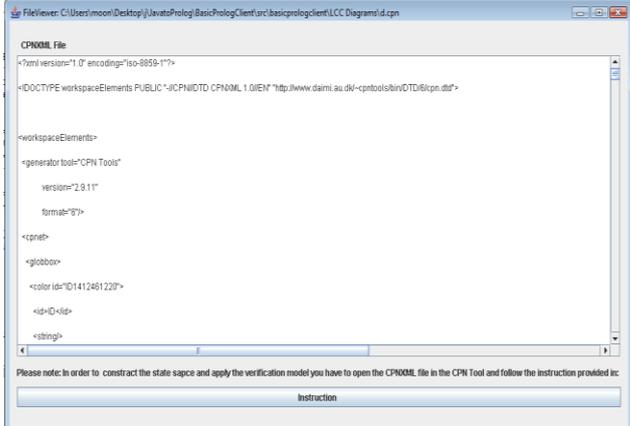


Figure 7.10: Transforming LCC Protocol into an Equivalent CPN Model Screens

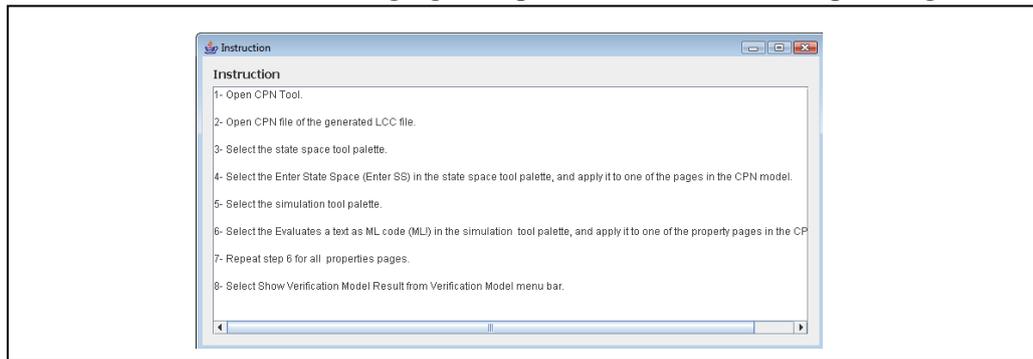


Figure 7.11: Instruction Screen

- (3) Select the state space tool palette.
- (4) Select the Enter State Space (Enter SS) in the state space tool palette, and apply it to one of the pages in the CPN model.

4- Apply the Verification Model

To Apply the verification model, the user needs to follow the steps numbered 5, 6 and 7 which appears in the instruction screen (see chapter 6 and appendix C):

- Step 5: Select the simulation tool palette.
- Step 6: Select the 'Evaluates a text as ML code (ML!)' in the simulation tool palette, and apply it to one of the property pages in the CPN model.
- Step 7: Repeat step 6 for all properties pages.

5- Display the Verificaiton Model Result

To display the verificaiton model result, the user needs to follow step numbered 8 which appears in the instruction screen (see chapter 6 and appendix C). The user needs to click on the '*Verification Model Result*' button (on the *Verification Model* menu bar in the DID textual representation screen of a persuasion dialogue in Figure 7.12).

In this example, when the user clicks on the '*Verification Model Result*' button, the reminder dialog box will appear to remind the user to construct the state space and to apply the verification model activities. The user has to click on the 'Yes' button to display the verification model result screen (as shown in Figure 7.12).

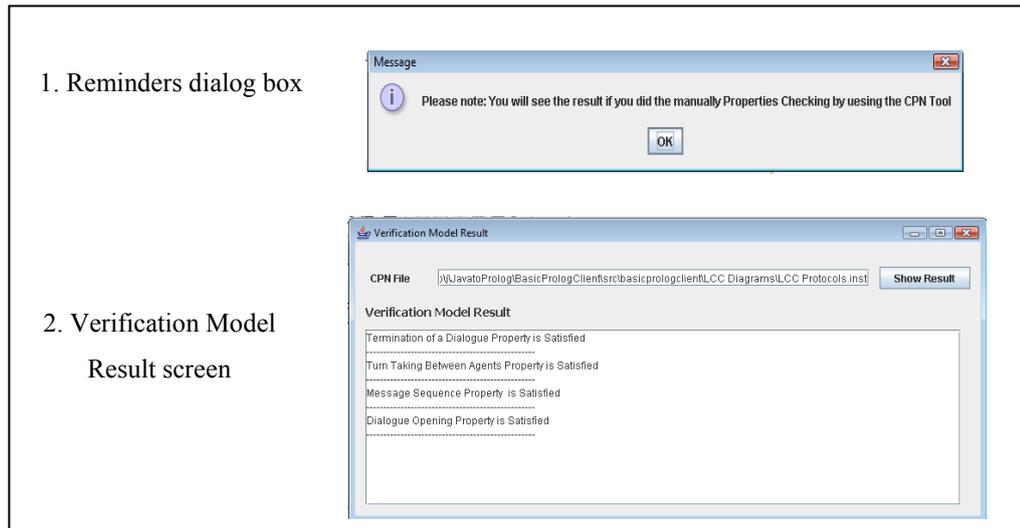


Figure 7.12: Verification Model Result Screen

7.3 Summary

This chapter has given an overview of the architecture of the thesis. It also has discussed an example which illustrates how the *GenerateLCCProtocol* tool is used to create DID diagrams, synthesis the concrete LCC protocols, and verify the synthesised protocols.

Chapter 8

Evaluation and Discussion

This chapter discusses and summarises the main contributions of this thesis. It also points out limitations of the thesis. Discussions on the synthesiser (synthesis of concrete protocols), the verification method and the *GenerateLCCProtocol* tool are given in Sections 8.1, 8.2 and 8.3, respectively. Lastly, Section 8.4 summarises this chapter.

8.1 Synthesis of Concrete Protocols

The purpose of this thesis, as mentioned in chapter 1, has been to bridge the gap between argument specification and protocol implementation using an extension of the Argument Interchange Format (AIF), that we called Dialogue Interaction Diagram (DID), as the specification language and the Lightweight Coordination Calculus (LCC) as an implementation language.

Both chapter 4 and 5 as well as appendices A, B and C have demonstrated how automated synthesis method can connect argumentation to MAS interaction protocols in a process language. This, potentially, could allow developers of argumentation systems to use specification languages to which they are accustomed (in our case AIF/DID) to generate systems capable of direct deployment on open infrastructures (in our case LCC).

The following subsections discuss the relation between DID and AIF, the difference between DID and related languages (AIF extensions) and the limitations of the synthesis methods (including DID and LCC-Argument patterns).

8.1.1 Relation between DID and AIF

The synthesis of concrete protocols approach presented in this thesis began with AIF. However, as mentioned in chapter 3 and 4, a fully automated synthesis beginning

from the AIF is not possible because AIF is an abstract language that does not capture dialogue game concepts (e.g. locutions, starting rules and turn taking rules), neither it captures some protocol implementation concepts (e.g. sender and receiver agent's roles concept) that are needed to support the interchange of arguments between agents. An example of AIF obstacle is shown in chapter 3 section 3.8.3.

The only two studies which have attempted to solve the AIF obstacle are Modgil and McGinnis [Modgil and McGinnis, 2007] and Reed et al. [Reed et al., 2008 ; Reed et al., 2010]. The limitations of these two approaches are demonstrated with examples in chapter 3. Modgil and McGinnis' [Modgil and McGinnis, 2007] work extends AIF to represent argumentation-based dialogues. [Reed et al., 2008] extended AIF to AIF+ so that it can handle argumentation dialogue games as well as represent the relation between the locution (in AIF+) and its propositional content (in AIF). However, similarly to AIF, AIF+ is used to represent data, not to process data. In fact, both Modgil and McGinnis [Modgil and McGinnis, 2007] and Reed et al. [Reed et al., 2010; Reed et al., 2008] attempted to solve the dialogue game problem of AIF (by adding dialogue games concepts to AIF), while failing to address the implementation problem (adding protocol concepts to AIF). See Section 8.1.2 for more details.

To remedy this, this thesis proposes a new intermediate language between the AIF and LCC called DID, which requires additional information that cannot be deduced from AIF. In practice, DID is a new layer on top of AIF. DID is used to represent interaction protocol rules between two agents. It has the dialogue games concepts (locutions, participants commitment store and commitment rules, structural rules, turn taking rules, pre-condition rules, post-condition and locution types) and protocol implementation concepts (sender and receiver agent's roles concept). The definition of DIDs and the example of DIDs are provided in chapter 4.

As mentioned above, this research attempts to close the gap between standard argument specification and protocol implementation by automating the synthesis of

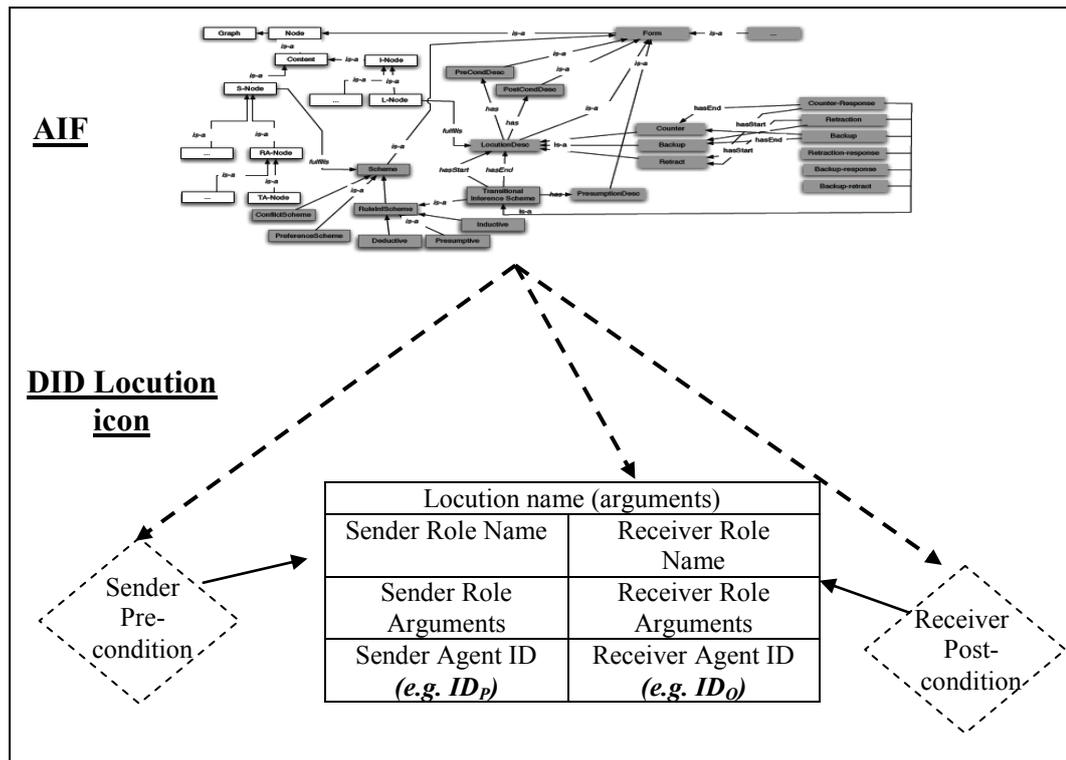


Figure 8.1: The Relationship between AIF and DID Locutions Icon

protocols in LCC from argument specifications written in the AIF. However, by the time we get to the DID, little of the AIF remains.

In fact, AIF could be embedded inside the agent and used by agent to express his knowledge and check the satisfaction of the message constraints. Therefore, DID is not an extension of AIF. It is important to point out that DID can work with any argument format (written in the AIF, or in other argumentation-based formalism) where DID coordinates arguments exchange between agents and the argument format (such as AIF) express the agent knowledge for the constraints.

DID is more than an argumentation language. It can be used to describe all argumentation systems that can be described as a sequence of turn taking recursive steps terminating in a base case.

The Relationship Between the DID and AIF Example

The relationship between DID and AIF is that DID arguments could be expressed in AIF (see Figure 8.1). The following example in Figure 8.2(a) and Figure 8.2(b) concerns the flying abilities of birds and penguins (see chapter 3 for more details)

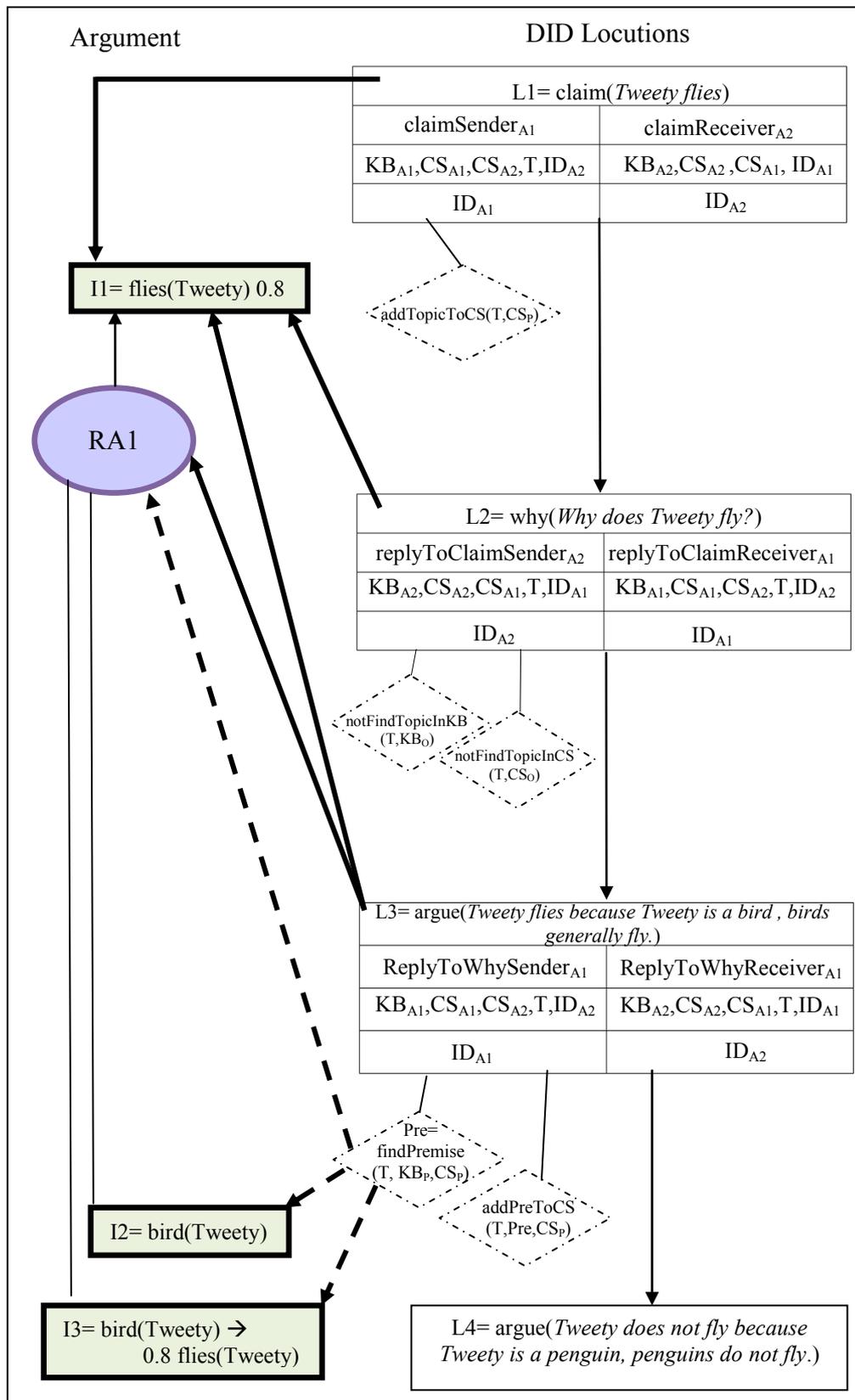


Figure 8.2 (a): Illustrating the Link between Argument (AIF Nodes) and DID Locutions

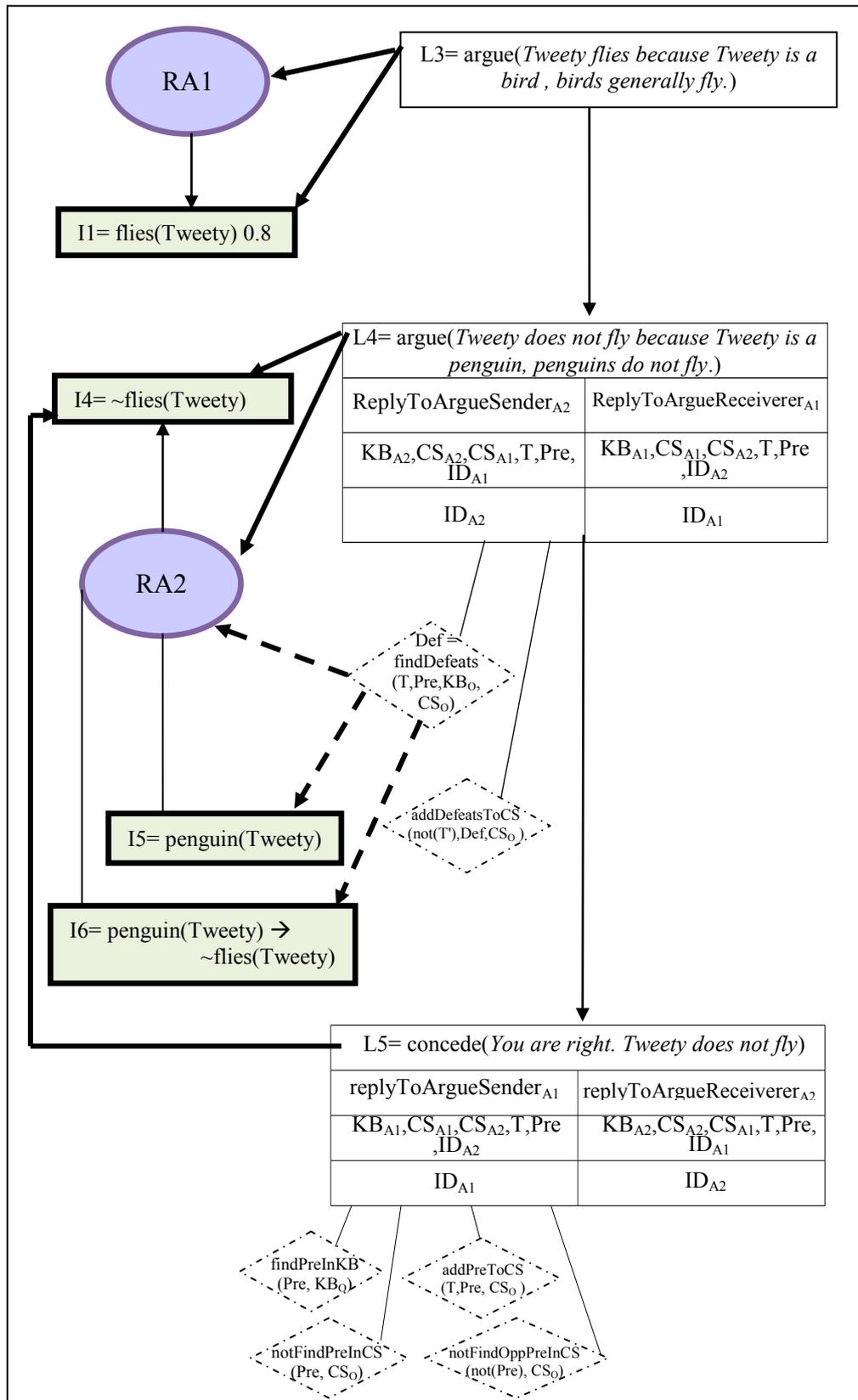


Figure 8.2 (b): Illustrating the Link between Argument (AIF Nodes) and DID Locutions

and shows the relationship between DID diagram (in Figure 4.3 in chapter 4) and the AIF diagram (in Figure 3.3 in chapter 3) (please note that this relationship is not added automatically):

In this dialogue between $A1$ and $A2$, the dialogue game consists of five locutions which are represented by $L1, L2, L3, L4$ and $L5$ icons. The argument consists of six propositions which are represented by $I1, I2, I3, I4, I5$ and $I6$ nodes. The interaction between the argument (AIF diagram) and the dialogue game (DID diagram) is described by the thick arrows and the relation between the argument (AIF diagram) and the constraint in the dialogue game (DID diagram) is described by the dotted arrows. The $L1$ and $L2$ have a direct link with the propositional content $I1$ (see Figure 8.2(a)). The links between $L3$ with $I2$ and $I3$ (see Figure 8.2(a)) are represented by $RA1$ node (the $RA1$ node connects $I1$ "*flies(P)*" with its premises $I2$ and $I3$).

The $RA2$ node links $L4$ and its propositional content $I5$ and $I6$ (the $RA2$ node connects $I4$ "*~flies(P)*" with its premises $I5$ and $I6$). Finally, $L5$ has a direct link with $I4$ (see Figure 8.2(b)).

In this example:

- (1) $A1$ opens the discussion by sending *claim(I1)* in $L1$ locution.
- (2) DID diagram (in Figure 4.3 in chapter 4) specifies that $A2$ can reply with *why(T)* or *concede(T)*.
- (3) $A2$ sends *why(I1)* in $L2$.
- (4) DID diagram (in Figure 4.3 in chapter 4) specifies the legal replies *argue(β)* where β 's conclusion is T , or *retract(T)*.
- (5) $A1$ responds to the challenge by declaring the supporting premises $I2$ and $I3$ for $I1$ [sends *argue(I2 and I3)* in $L3$ node]. Note that $A1$ satisfies the *argue* message constraint $Pre=findPremise(T, KB_p, CS_p)$ using AIF which describes the relation between $A1$'s argument ($T=Tweety\ flies$ in $I1$) and the its supporting

premises ($Pre = Tweety\ is\ a\ bird, birds\ generally\ fly$ in $I2$ and $I3$) (see Figure 8.2(a)).

(6) *DID* diagram (in Figure 4.3 in chapter 4) specifies the legal replies $why(\beta)$, $argue(\beta)$ where β 's conclusion is T , or $concede(T)$.

(7) $A2$ responds by declaring its supporting premises $I5$ and $I6$ for $I4$ [sends $argue(I5\ and\ I6)$ in $L4$ node]. Note that $A2$ satisfies the $argue$ message constraint $Def = findDefeats(T, Pre, KB_O, CS_O)$ using AIF which describes the relation between $A2$'s argument ($T = Tweety\ does\ not\ flies$ in $I4$) and the its supporting premises ($Def = Tweety\ is\ a\ penguin, penguins\ do\ not\ fly$ in $I5$ and $I6$) (see Figure 8.2(b)).

(8) *DID* diagram (in Figure 4.3 in chapter 4) specifies the legal replies $why(\beta)$, $argue(\beta)$ where β 's conclusion is T , or $concede(T)$.

(9) $A1$ responds by sending $I4$ [sends $concede(I4)$ in $L5$ node].

This example shows that the *DID* can work with argument formats written in the AIF.

8.1.2 The Difference between *DID* and AIF Extension

As explained in detail in chapter 3 section 3.8.5, two studies have attempted to solve the AIF problem by extending the AIF to handle some dialogue game concepts:

- (1) Modgil and McGinnis [Modgil and McGinnis, 2007];
- (2) Reed et al. [Reed et al., 2008; Reed et al., 2010] (Please note that the AIF+ is still an ongoing work and our research was developed in parallel to this work).

Table 8.1 summarises the major differences between these two studies and *DID*:

(1) Locution Concept (Figure 8.3):

- *DID*: locutions are represented in the form of Locution icon (see chapter 4);

	DID	Modgil and McGinnis [Modgil and McGinnis, 2007]	Reed et al. [Reed et al., 2008]
Represent Locutions	Locutions are represented in the form of Locution icon.	Expands I-nodes content to represent locution	Locutions are represented in the form of L-nodes
Dialogue Games Concepts	DID has the following dialogue games concepts: locutions, participants, commitment store, commitment rules (post-conditions), structural rules, turn taking rules, pre-condition rules and locution types.	Modgil and McGinnis work has the following dialogue games concepts: locutions, pre-conditions and structural rules.	AIF+ has the following dialogue games concepts: locutions, pre and post conditions and structural rules.
Protocol automated synthesis	The user can perform fully automated synthesis of multi-agent protocols using LCC–Argument patterns	The user cannot perform fully automated synthesis of multi-agent protocols	The user cannot perform fully automated synthesis of multi-agent protocols
Argument Format	DID can work with any argument format	Only AIF	Only AIF

Table 8.1: Differences between Modgil and McGinnis [Modgil and McGinnis, 2007], Reed et al. [Reed et al., 2008] and DID

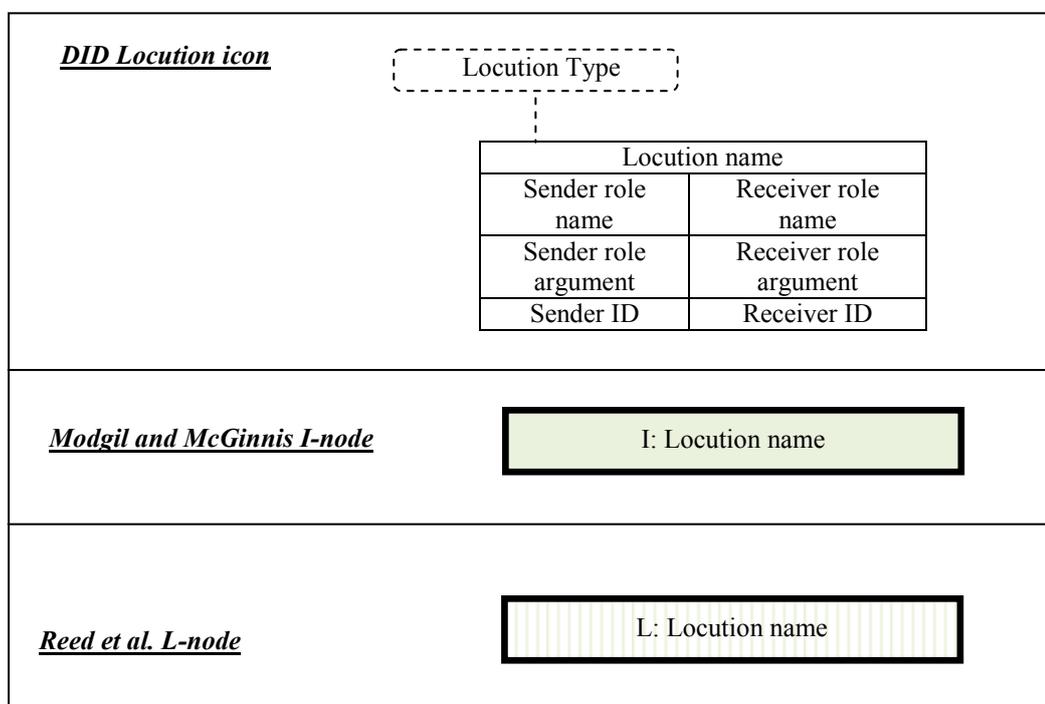


Figure 8.3: Locution Concepts

- Modgil and McGinnis [Modgil and McGinnis, 2007] expand Information nodes (I-nodes) content to represent locution (see chapter 3, section 3.8.5 for more detail about I-node);
- Reed et al. [Reed et al., 2008]: locutions are represented in the form of Locution nodes (L-nodes), a subclass of Information nodes (I-nodes) (see chapter 3, section 3.8.5 for more detail).

(2) Dialogue Game Concepts (Figure 8.4(a) and (b)):

- DID represents eight concepts of the dialogue games [Locutions; Pre-condition rules; Post-condition rules; Structural rules; Participants Commitment Store and Commitment rules; Locution types (Starting rules and Termination rules which are used to specify when the dialogue starts and when the dialogue ends); Turn Taking rules] using the locution icon (see chapter 4);
- Modgil and McGinnis [Modgil and McGinnis, 2007] represent three dialogue game concepts (as shown in Figure 8.4 and Figure 8.5):
 - a) Locutions: are represented by an I-node;
 - b) Pre-conditions: are represented by PIA-node (see chapter 3, section 3.8.5 for more detail about PIA-node);
 - c) Structural rules: are represented by PIA-node;
- AIF+ (by Reed et al. [Reed et al., 2010]) represents four dialogue games concepts (as shown in Figure 8.4 and Figure 8.6):
 - a) Locutions: are represented by an L-node;
 - b) Pre- and post-conditions: are represented by a Locution Description (LDesc-nodes) nodes [Reed et al., 2010]. In AIF+, for each locution, represented by an L-node, there is a corresponding LDesc-node. Each LDesc-node is linked to a corresponding PreCondDesc node (it describes

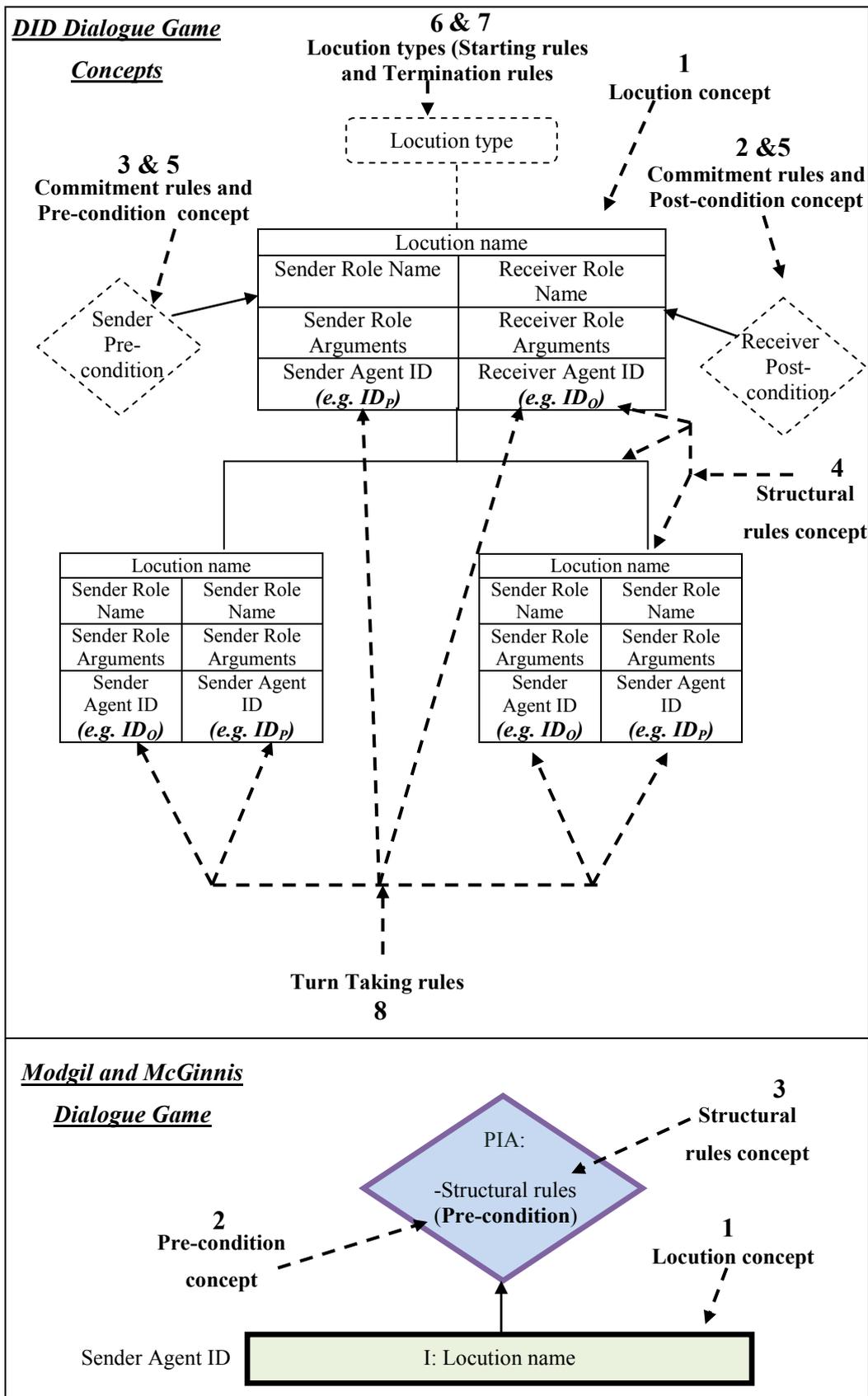


Figure 8.4 (a): Dialogue Games Concepts

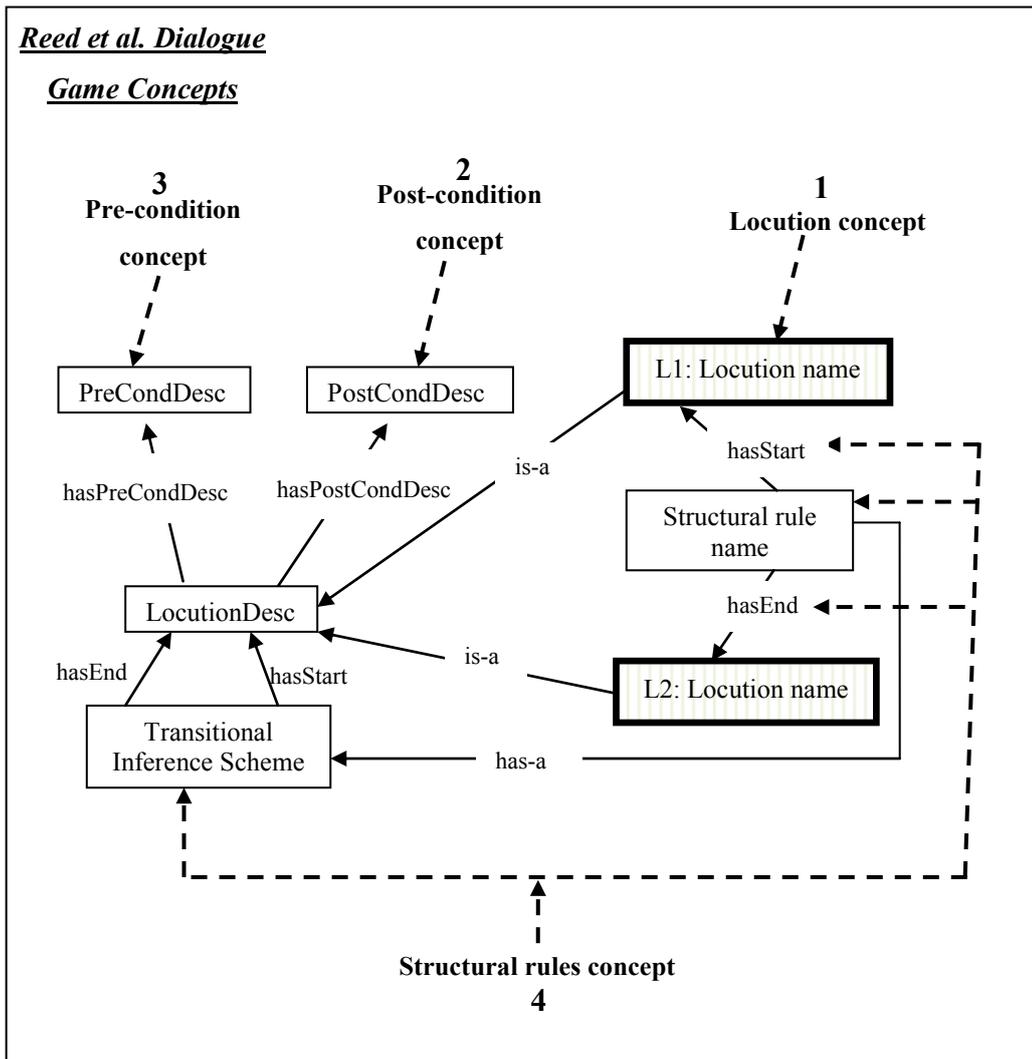


Figure 8.4 (b): Dialogue Games Concepts

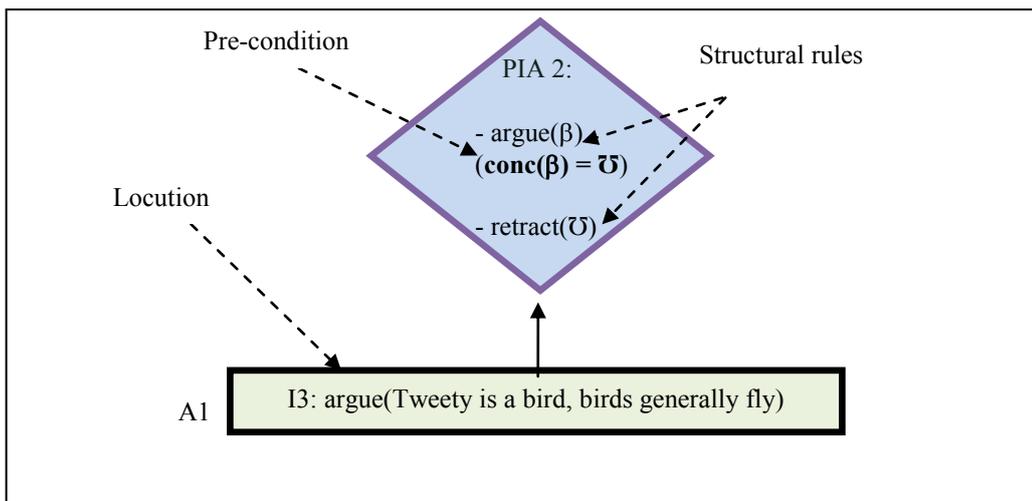


Figure 8.5: Modgil and McGinnis Example of Dialogue Games Concepts

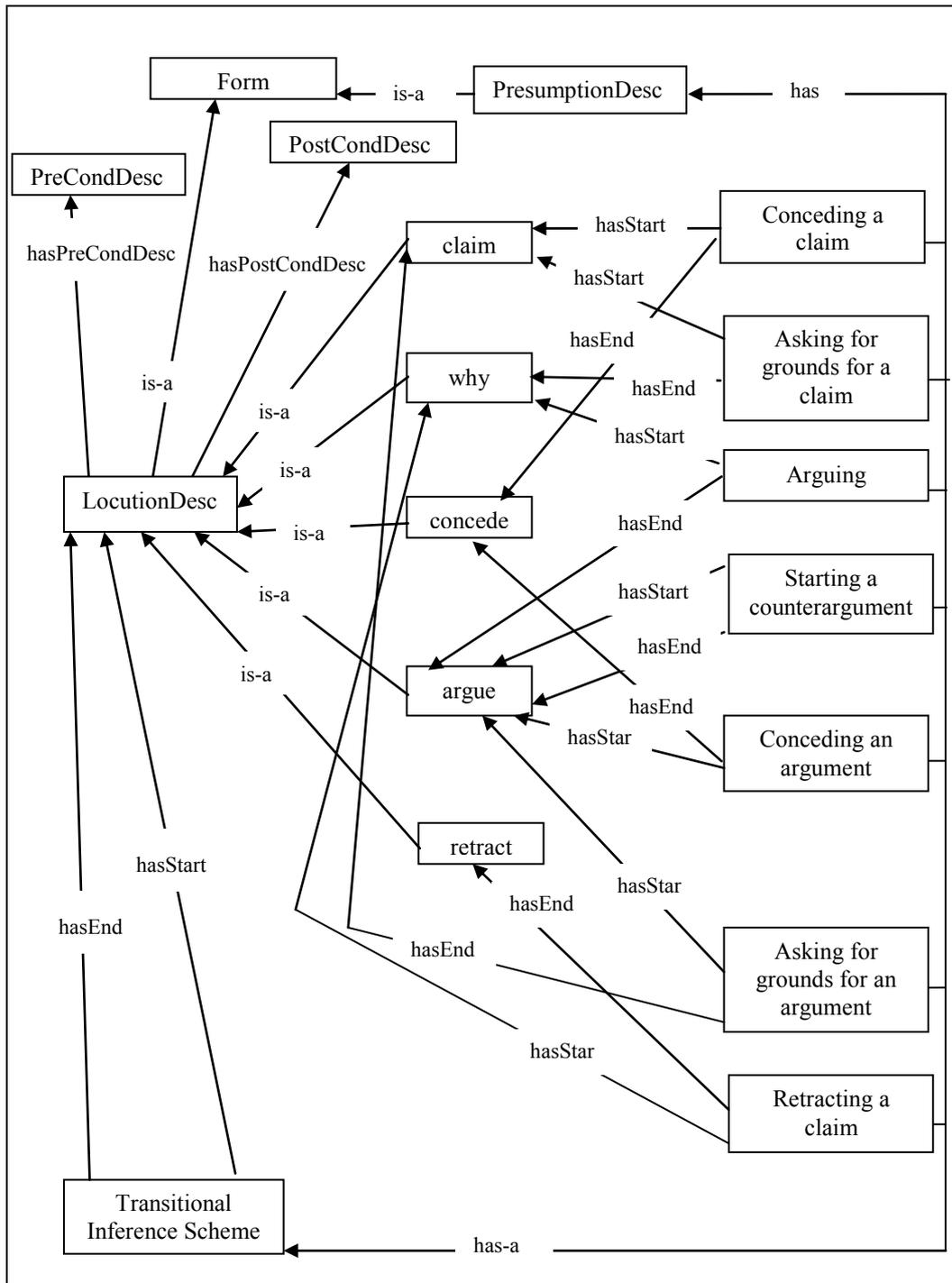


Figure 8.6: AIF+ Description of Persuasion Dialogue Games

the pre-conditions of locution) and PostCond-Desc nodes (it describes the post-conditions of locution). In the AIF+ (Figure 8.6) representation of persuasion dialogue in chapter 3, there are five LDesc-nodes corresponding to the five locutions: *claim*, *why*, *concede*, *argue* and *retract*;

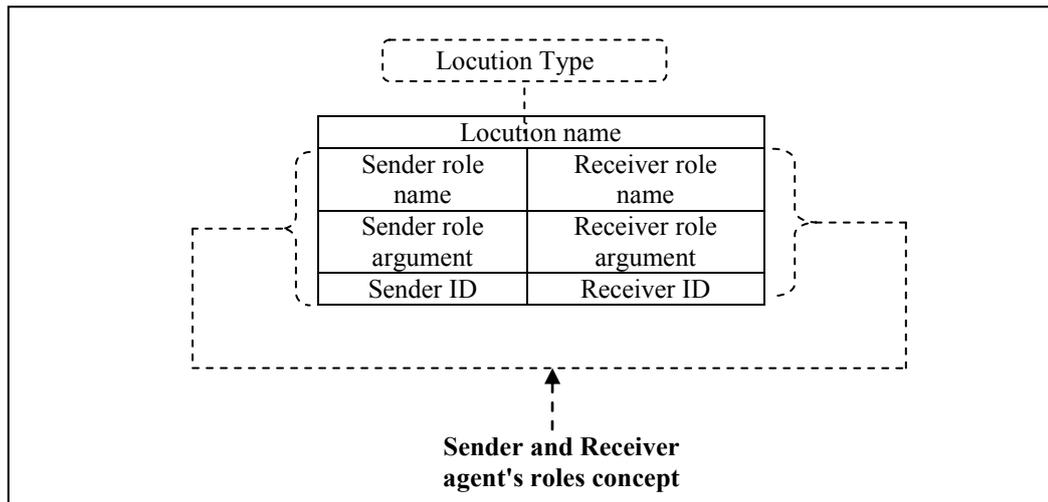


Figure 8.7: DID Protocol Implementation Concepts

- c) Structural rules: the structural rules (the order sequence of locutions) are represented by a transitional inference schemes node which describes, for a given locution, the available locutions that a participant can select to follow from the previous locution. In the AIF+ (Figure 8.6) representation of persuasion dialogue in chapter 3, there are seven transitional inference scheme nodes which describe the available responding persuasion dialogue locutions for an uttered locution (e.g. the *why* locution may be followed by either an *argue* or a *retract* locution);

(3) Protocol automated synthesis:

- DID: The user can perform a fully automated synthesis of multi-agent protocols using LCC–Argument patterns since DID represents dialogue game protocols [it has eight dialogue games concepts as well as protocol implementation concept (Sender and receiver agents roles) as shown in Figure 8.7] (see chapter 5 for more detail);
- Modgil and McGinnis [Modgil and McGinnis, 2007]: The user cannot perform a fully automated synthesis of multi-agent protocols. Their work does not present all concepts which are needed in order to perform the automated synthesis: (1) Post-conditions (helps to control agent behaviour); (2) Turn Taking rules (help to control agent behaviour); (3) Starting rules (help to control the starting of a dialogue); (4) Termination rules (help to control the

ending of a dialogue); (5) Sender and receiver agents roles (help to control the way the dialogue proceeds).

- AIF+ (by Reed et al. [Reed et al., 2008]): The user cannot perform a fully automated synthesis of multi-agent protocols. AIF+ does not present all concepts which are needed in order to perform the automated synthesis: (1) Turn Taking rules (help to control agent behaviour); (2) Starting rules (help to control the starting of a dialogue); (3) Termination rules (help to control the ending of a dialogue); (4) Sender and receiver agents roles (help to control the way the dialogue proceeds).

(4) Argument Format

- DID can work with any argument format written in the AIF, or in other argumentation-based formalism such as The Legal Knowledge Interchange Format (LKIF) [Gordon, 2008]. See section 8.1.1.
- Modgil and McGinnis's approach [Modgil and McGinnis, 2007] can only work with AIF.
- AIF+ (by Reed et al. [Reed et al., 2008]) can only work with AIF.

8.1.3 DID Limitation

The DID can model large classes of argumentation systems (dialogue games) that can be described as a sequence of turn taking recursive steps terminating in a base case such as persuasion and negotiation dialogues (see chapter 4 and appendices A and C). However, the DID has two limitations:

(1) Two agents:

We limited the DID diagram to two agents since the DID for N-agent needs more concepts (e.g. recursive-conditions and recursive-arguments) which could make the DID too close to an agent protocol and make the drawing of the DID diagram for N-

agent more difficult than writing the agent protocol in LCC notation (see chapter 4, section 4.4.5).

However, in chapter 4, section 4.4.4 we were able to extend the DID locution icon to represent N-agent dialogue games (see appendix B for more detail), although this is not the most elegant solution (it is too complex for the user to create, understand and edit). In doing so, we showed that it is possible to extend DID diagram.

To overcome the complexity of drawing the DID for N-agent, we hid the details of DID diagrams for N-agent in a reusable black box and we used parameters to get the information needed from the user. Besides, we performed automated synthesis of the protocol and used a specific type of LCC-Argument patterns called *broadcasting pattern* (see chapter 5, section 5.2.2). See section 8.1.4 for more detail.

(2) Unique-moves and Immediate-reply:

We restricted an agent's moves to unique-moves (an agent can make a single reply for each possible move of the other agent. In other words, agents are not able to send more than one message in one round of turn taking) and immediate-reply moves (the turn taking between agents switches after each move and each agent must reply to the move of the previous agent) (see chapter 4, section 4.2).

Although, many current systems [Prakken, 2005] enforce control structure (unique-moves and immediate-reply), sometimes agents in dialogue games must have freedom to explore multiple moves and alternative replies in one turn, returning to earlier choices or to postpone replies. For example, unique-moves and immediate-reply dialogue games are more appropriate when a quick decision has to be reached, since this restriction forces agents to move their strongest arguments without wasting time on other choices [Prakken, 2005]. However, multi-moves (when agents can make several moves before the turn taking between agents switches) and non-immediate-reply (the turn taking between agents may switch after each move or may switch later) dialogue games are more appropriate when the quality of the outcome is more important than the time spent on it [Prakken, 2005].

We chose to enforce this restriction in order to be able to perform protocol automated synthesis directly from a DID specification. However, if we want to use the DID to model multi-moves and non-immediate-reply dialogue games, we do not need to change the DID. We need to add new set of LCC-Argument patterns to our library to allow the synthesiser to generate LCC argumentation protocols for multi-moves and non-immediate-reply dialogue games. See section 8.1.4 for more detail.

8.1.4 LCC-Argument Patterns Limitations

The LCC-Argument patterns can be used with the DID to generate agent protocols for many standard types of argumentation systems such as persuasion and negotiation dialogues (see chapter 5 and appendices A, B and C). However, the LCC-Argument patterns have some limitations:

LCC-Argument Patterns for Two Agents

Two patterns (Starting pattern and Termination-Intermediate Pattern) were proposed to synthesise LCC protocols, for two agents, automatically. At this stage we could claim that we have a full set of patterns to synthesis LCC argumentation protocols for two agents. However, as explained in section 8.1.3, these protocols are limited to unique-moves and immediate-reply dialogue games.

We believe that if we want to provide a solution for multi-moves and non-immediate-reply dialogue games, we will need to add a new set of LCC-Argument patterns (which may contain a lot of detailed information) to our library. For example, if we want to allow a Termination-Intermediate Pattern to work with multi-moves, we have to add a set of *Rewriting methods* which have the ability to consider all different collections of possible sequences of moves (locutions).

Let us consider the example in Figure 8.8 (some details are omitted from Figure 8.8 for clarity). In this example, Level 3 has 3 locutions which means there are 15 different collections of possible sequences of reply moves to locution icon *argue* in

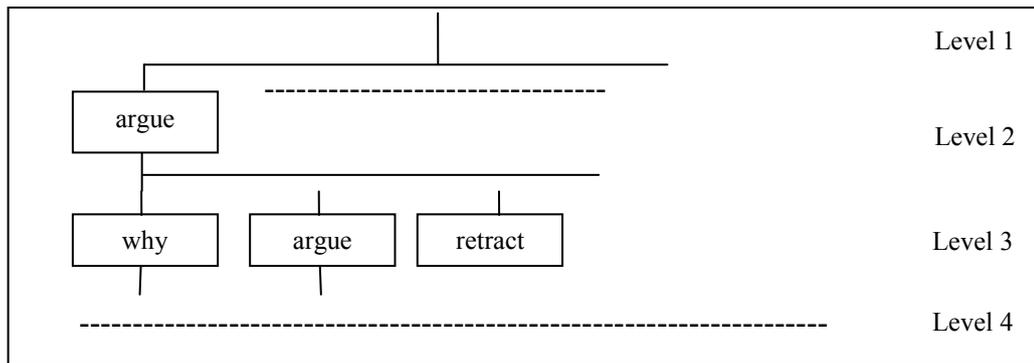


Figure 8.8: Partial DID Diagram

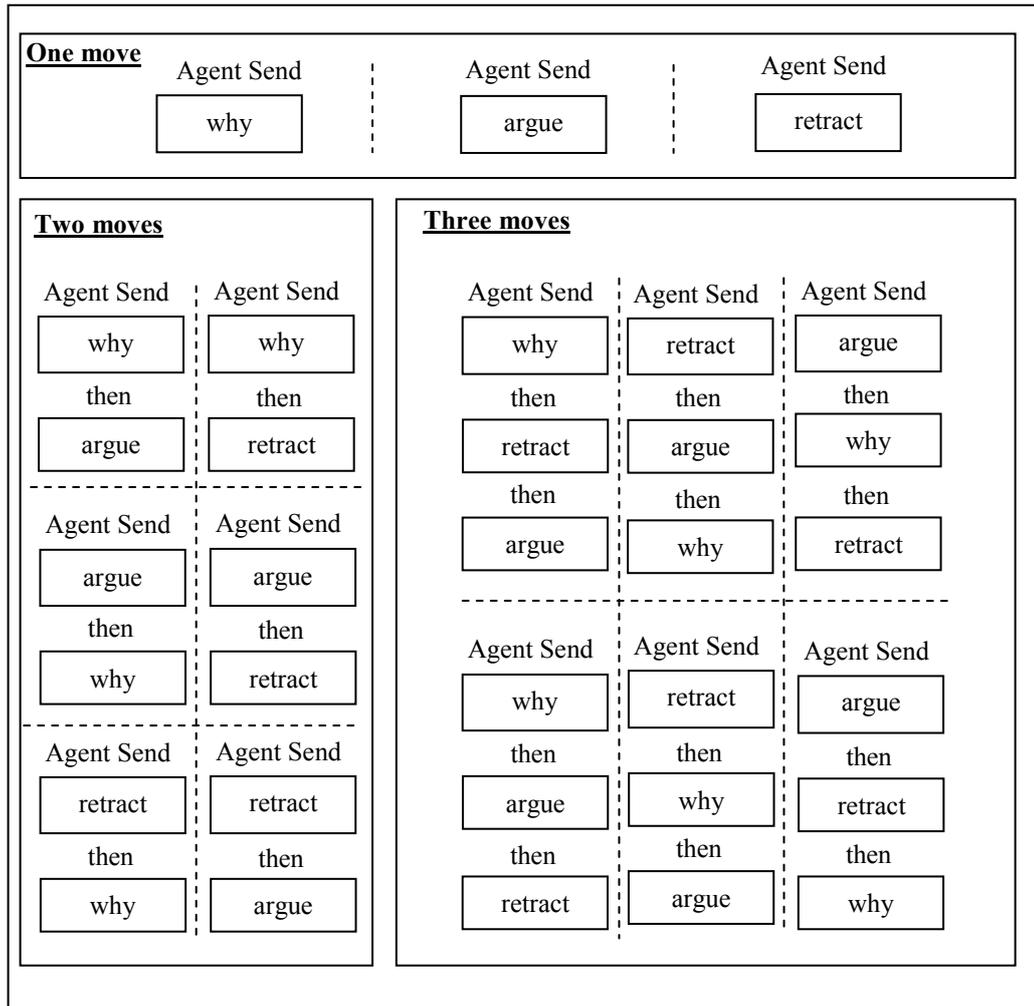


Figure 8.9: Possible Sequence Of Reply Moves

level 2 (Figure 8.9 shows 15 possible sequence of reply moves for locution icon *argue* in level 2). This means that the new *Rewriting methods* must be able to:

- (1) Use a specific mathematical function to find the number of possible sequence of reply moves;

- (2) Provide a way (an algorithm) to select the correct next move(s).
- (3) Provide a way (an algorithm) to avoid repeating the same sequences of moves (locutions).

Therefore, it would require adding algorithmic information to this pattern, which could be very difficult to edit by non-technical users (see chapter 9 for more detail).

LCC-Argument Patterns for N-agent

Part of our research focused on dialogue games involving more than two agents. However, we generated one type of LCC argumentation protocols for N-agent. Practically, our automated synthesis method uses an LCC-argumentation broadcasting pattern to divide agents into groups composed of two agents. Then it follows the automated synthesis process of two agents' protocols (see chapter 5, section 5.2.2) to generate the LCC protocols, which allows pairs of group to communicate with each other.

This means that our tool limits the LCC argumentation protocol for N-agent to a broadcasting pattern. However, it is interesting to consider what it would be like to actually build more patterns that can deal with any type of N-agent protocol. Can we have a full set of patterns to synthesis LCC argumentation protocols for N-agent? To do this we would need to either:

- (1) Create one pattern and add more detail to it (to be able to work with different types of N-agent protocols), which could make it very difficult for non-technical users to edit.
- (2) Add more detailed patterns to the LCC-Argument patterns library. It is true that more detailed patterns are more useful than abstract ones, in the sense that they can model more dialogue games. However, detailed patterns usually become too specific and are less likely to occur frequently.
- (3) Extend the DID diagram to represent N-agent diagrams (be able to represent recursive concepts) and add more patterns to the library in order to work with

the new notations in the DID diagram which we previously performed (see appendix B). Although this is not the most elegant solution (it is too complex for the user to create, understand and edit DID diagram for N-agent), we showed that it is possible to extend the DID diagram and synthesise N-agent protocols. However, it would appear that in the case of N-agents, we cannot obtain a complete set of LCC-Argument patterns. Furthermore, there are some limitations in the LCC language itself. The LCC language supports only sequential definitions of roles. For example, if an agent in a given role wants to send the same message to a group of agents all at exactly the same time, LCC cannot model that, although it could send a number of copies of the same message in sequence. In fact, it would appear that, in the case of N-agent, we cannot obtain a complete set of LCC-Argument patterns.

8.2 Verification Method based on Coloured Petri Net and SML

This thesis explained a verification method based on CPNs and SML (see chapter 6). Given the DID and the LCC specification, our verification tool could answer the question: Does the LCC specification satisfy the DID behaviour properties? To answer this question, the tool performs the following tasks:

- (1) Automatically transforms the LCC specification into an equivalent CPNXML file;
- (2) Constructs from the CPNXML file the state space;
- (3) Automatically creates DID properties files;
- (4) Automatically verifies the satisfaction of the DID properties in the state-space graph computed from the LCC protocol by applying a verification model.

The next subsections discuss the limitations of the four steps of our verification method.

8.2.1 Limitations of Transforming the LCC Specification into an Equivalent CPNXML File

Our verification method generates a hierarchical CPN model from an LCC specification by using a set of transformational rules. Although many steps of our approach are automatic, our approach is not able to automatically transform LCC parameters to colour set types of the CPN model which is a result of LCC being an untyped language. This means that the user needs to manually supply colour set types information to the generated CPNXML file.

By default our verification tool defines three types of colour set and thirteen functions (see chapter 6, section 6.1 for more detail) and saves them in the *Global Declaration* file. The user does not need to know about them unless he/she needs to define new types or functions. That means the user needs to learn CPN colour sets and function concepts as well as the CPN SML language in order to supply this information to the generated CPNXML file. However, the user does not need to become a CPN SML programmer in order to supply this information. He/she needs only to learn how to declare colour sets (data types) and variables along with knowing how to compare one data (datum) value with another.

8.2.2 Limitations of Constructing of the State Space

The second step of the verification method is to construct from the CPN model its state space (directed graph, which represents all possible executions of the CPN model). The fourth step of the verification method concerns the full state space analysis which is possible if the state space of the CPN model has a fixed size (i.e. the state space graph has a finite number of nodes). Although, we have not experienced a state space explosion problem with the persuasion and negotiation dialogues examples (appendices C and A) as explained in chapter 6, our verification method is likely to encounter the state space explosion problem (state space analysis will be prohibited because of the infinite number of the state space graph nodes). This is because the CPN model could have a finite number of agents but for instance it could describe an LCC protocol where agents can be involved in an infinite loop.

8.2.3 Limitations of the Verification Method

Our verification method identifies five basic properties, which are independent of any dialogue games types (Dialogue opening property, Termination of a dialogue property, Turn taking between agents property, Message sequencing property and Recursive message property). See chapter 6, section 6.4 for more detail. If the user needs to verify different properties than these five properties, the user needs to manually add the new properties to the generated CPNXML file (Appendix A describes how to add new properties to the generated CPNXML file with examples). That means that the user needs to learn the CPN SML language (in other words, become a CPN SML programmer) in order to write the new property code.

8.3 GenerateLCCProtocol Tool

The *GenerateLCCProtocol* tool (see chapter 7 for more detail) enables the user to synthesise LCC protocols automatically from DID specifications and verify the semantics of the DID specification against the semantics of the synthesised LCC protocol automatically.

This tool has been designed and implemented to perform two tasks:

- (1) Synthesis of concrete protocols;
- (2) Model verification.

The next two subsections discuss the limitations of these two tasks.

8.3.1 Task One: Synthesis of Concrete Protocols

The *GenerateLCCProtocol* tool receives a DID as an input and returns the corresponding LCC specification protocol. One advantage of the DID is that it is a high-level graphical language (see chapter 4, section 4.2 for more detail) and people in the agent community are familiar with high-level language or graphical notation

languages like Agent Unified Modelling Language (UML)¹⁹ [Bauer et.al., 2001]. Also, specifying argumentation protocols using programming-level protocol languages is error-prone, and a higher-level graphical language can help avoiding low-level errors.

Unfortunately, our tool does not have a graphical representation for all DID diagrams files. In fact, the tool allows the user to create the DID diagram by providing one locution icon information at a time using locution icon graphical representation (see chapter 7, Figure 7.14). For each locution icon the tool generates a textual representation for it and saves it in the DID diagram file (see chapter 7 for more detail). Then, if the user needs to edit the DID diagram file, the user has to edit the DID textual representation. This means that the user has to know the formal representation of the DID as well as the graphical notation of the DID diagram.

To avoid this problem it would be useful for the user to create, review and edit the DID diagram in a graphical way which means that more work is needed to improve our tool (see chapter 9 for more detail).

8.3.2 Task Two: Model Verification

For this task, the *GenerateLCCProtocol* tool receives a DID and the LCC specification protocol as an input, verifies them and then answers the question: Does the LCC specification satisfy the DID properties? This is explained in chapter 6 and section 8.2. Four steps are needed to answer this question:

- (1) Transforming the LCC specification into an equivalent CPNXML file. This step is processed by the *GenerateLCCProtocol* tool in a fully automatic way;
- (2) Constructing the state space. Unfortunately, the *GenerateLCCProtocol* tool is not able to construct the state space in an automatic way. The user needs to open

¹⁹ UML is a graphical language which consists of a set of graphic symbols. It is used to create, process, and model agent-based software, object-oriented software and workflows.

the CPNXML file using the CPN Tool and construct the state space in a manual way (see chapter 6, section 6.2.1 and chapter 7). In fact, the CPN Tool team created the Access/CPN [Westergaard and Kristense, 2009] tool to connect the CPN tool with external applications (e.g. Java applications) which could help to construct the state space in a fully automatic way. Unfortunately, we were not able to use the Access/CPN tool to connect the CPN Tool with the *GenerateLCCProtocol* because there are some problems in the Access/CPN tool itself²⁰.

- (3) Creating DID properties files. This step is processed by the *GenerateLCCProtocol* tool in a fully automatic way;
- (4) Verifying the satisfaction of the CPN SML specification in the state-space graph computed from the LCC protocol by applying a verification model. This step is processed by the *GenerateLCCProtocol* tool in a semi-automatic way. The *GenerateLCCProtocol* tool generates the CPN SML code of the five basic properties (chapter 6, section 6.4). To verify these five basic properties, the user needs to:
 - Open the generated CPNXML file;
 - Select in the CPN Tool the simulation tool palette;
 - Select the *Evaluates a Text as ML Code(ML!)* icon in the simulation tool palette and apply it to one of the property pages;
 - Repeat these steps for all properties pages;
 - Select *Verification Model Result* from the verification menu bar in the *GenerateLCCProtocol* tool.

²⁰ We spent three months trying to connect the CPN Tool with the *GenerateLCCProtocol* tool using the Access/CPN tool. We contacted the CPN tool team and they acknowledged the bugs we found in the tool.

8.4 Summary

This chapter has discussed the thesis findings and contributions as well as provided an overview of the limitations of this thesis. Our evaluation also highlighted areas where more work is needed. The next chapter will discuss how the work could be improved and outline directions for future research.

Chapter 9

Conclusion and Future Work

This chapter summarises the thesis contributions in Section 9.1 and also outlines directions for future research in Section 9.2.

9.1 Summary of Contributions

This thesis, as mentioned in chapter 1, has investigated the problem of the gap between argument specification languages and multi-agent implementation languages. One way of addressing this issue is through an automated synthesis method, so the specific question that we asked is whether a generic argumentation representation (acting as a high-level specification language) could be used to automate the synthesis of executable specifications in a protocol language capable of expressing a class of multi-agent social norms. As our argumentation language we have chosen the Argument Interchange Format (AIF). As our protocol language we have chosen the Lightweight Coordination Calculus (LCC).

Fully automated synthesis starting only from the AIF, as mentioned in chapter 3, is not possible because AIF is an abstract language that does not capture some concepts that are related to the interchange of arguments between agents (e.g. sequence of argument, locutions and pre- and post-conditions for each argument). An example of this obstacle is shown in chapter 3.

To remedy this obstacle, in chapter 4, we extended the AIF diagrammatic notation to give a new, intermediate recursive visual high-level language between the AIF and LCC called a Dialogue Interaction Diagram (DID). DID provides mechanisms to represent, in an abstract way, the dialogue game protocol rules by giving an overview of the permitted moves (messages) and their relationship to each other. It restricts agent moves to unique-moves and immediate-reply moves. This restriction is quite strict but it still allows to include a large class of argumentation systems in the

synthesizer, for instance all argumentation systems that can be described as dialogue games. In general, we can synthesise arguments that can be described as a sequence of turn taking recursive steps (each of which involves turn taking between the pair of agents) terminating in a base case. Given the turn-taking assumption, we can synthesise LCC protocols (which are executable) directly from DID specifications. However, a DID cannot explain how two or more agents can cooperate and interact with each other in situations where more complex protocols involving more than turn taking are required.

To overcome this problem, in chapter 5, we supplied LCC-Argument patterns, which are re-usable, parameterisable LCC specifications that can be embedded in automated synthesis tools and used with DID to support agent protocol development. By re-using design patterns repeatedly it is possible to reduce the effort of building complex argumentation protocols. The set of these more complex design patterns is, in theory, unbounded (for the same reason that design patterns in traditional software engineering are unbounded) but in practice families of interaction patterns occur. We have focused on those involving more than two agents where synthesized LCC protocols specify broadcasting methods to divide agents into groups composed of two agents (with these two-agent dialogues then being specified using DID).

Because design patterns could introduce errors in the synthesis process (since a poorly designed interaction pattern may result in an inappropriate LCC protocol even with a perfect synthesis mechanism), in chapter 6, we provided a verification methodology. The proposed verification strategies are based on SML and CPN to check the semantics of the DID specification used as a starting point against the semantics of the synthesised LCC protocol.

In conclusion, although the resulting synthesis and verification system is not an industry-strength specification tool, it demonstrates how automated synthesis methods can connect argumentation to MAS interaction protocols in a process language. This, potentially, could allow developers of argumentation systems to use specification languages to which they are accustomed (in our case AIF/DID) to

generate systems capable of direct implementation on open infrastructures (in our case LCC).

9.2 Improvements and Future Work

The results of this thesis point to several interesting directions for future work, in the hope of introducing further improvements to the DID, the automated synthesis method and the semi-automated verification method:

9.2.1 DID Future Work

So far, we have developed a high-level dialogue game protocol abstract language called DID. This language can represent any argument (dialogue game) system that can be described as a sequence of turn taking recursive steps terminating in a base case. DID can be used with LCC-Argument patterns for the automatic synthesis of LCC agent protocols, which means that users do not need to learn LCC language. But despite this fact, there are still several open issues and we want to point out two of them:

- **Natural Language:**

Although the DID language can model a large class of argumentation systems, it is interesting to consider who is likely to be able to use the DID notation. Will some users be able to use the DID notation while others cannot? Unfortunately, we do not know those answers ourselves since we did not test that. However, we assume that some users may have some problems working with DID notation. DID diagrams can become complicated simply because of the complexity of the modelled argumentation system. That means we need to find new ways to make DID easier to use. One way of addressing this issue is through connecting DID (formal language) with natural language, which might reduce the effort and time needed to build a DID diagram. In the future we would like to investigate the use of the natural language to get the dialogue game protocol information from the user.

- **Graphical Representation:**

As indicated in Chapter 8, the *GenerateLCCProtocol* tool does not have a graphical representation for all DID diagrams files. Although the user creates the DID diagram by providing one locution icon information at a time in graphical way, the user needs to learn the formal representation of the DID in order to be able to edit the DID diagram. In other words, more work is needed to improve the *GenerateLCCProtocol* tool to enable the user to create, review and edit the DID diagram in a graphical way.

9.2.2 Automated Synthesis Method Future Work

- **Deductive Synthesis:**

A DID cannot explain how two or more agents can cooperate and interact with each other, therefore we cannot go directly from DID to LCC. To overcome this problem, this thesis used structured synthesis method (pattern based approach). However, it is interesting to check whether this approach (structure synthesis) is the right way to address DID problem. Is there another way to solve this problem?

In fact, another way to generate the LCC agent protocol from the DID would be to use deductive synthesis²¹ methods, where the protocol generation task is viewed as a problem of proving a mathematical theorem. As a future work we would like to investigate the use of the deductive synthesis method to generate the LCC agent protocols. In other words, we would like to answer the following question: Is a deductive synthesis method easier and more effective than our structured synthesis method?

²¹ A deductive approach [Manna and Waldinger, 1980] "is presented for the construction of recursive programs. This approach regards program synthesis as a theorem-proving task and relies on a theorem-proving method that combines the features of transformation rules, unification, and mathematical induction within a single framework".

- **LCC-Argument Pattern Library:**

Currently, the LCC-argument pattern library is limited (as explained in Chapter 8) to two agent dialogue games, unique-moves and immediate-reply dialogue games and a broadcasting approach for N-agent dialogue games. This means that the investigation of new LCC-argument pattern is needed to improve our tool. Such improvements involve a better understanding of dialogue games, the LCC language and LCC-argument patterns.

One of the common patterns we would like to add is non-immediate-reply dialogue games (these systems do not typically require agents to reply immediately to the other agents' messages).

9.2.3. Semi-automated Verification Method Future Work

At this moment, our semi-automated verification method has some limitations (as explained in Chapter 8). The most important one is a verified properties issue.

The verification has succeeded in verifying five basic properties (Dialogue opening property, Termination of a dialogue property, Turn taking between agents property, Message sequencing property and Recursive message property) which are general properties that may be applied to several dialogue games. However, if the user needs to verify different properties, the user needs to specify these properties and feed them to the generated CPNXML file manually. Therefore, we believe further research needs to be carried out to address this issue. In fact, we intend to investigate three questions: Can the user modify the available properties to suit their specific dialogue game using the *GenerateLCCProtocol* tool? Can the *GenerateLCCProtocol* tool specify new properties in an automated manner? Can the *GenerateLCCProtocol* tool take the new properties information from the user using a constrained form of natural language?

9.2.4. Other Future Work

Because we had to extend the AIF to get a language that has enough information in it to generate the MAS protocols, we ended up with more versatile language called DID. We believe that the DID can represent things beyond arguments but we have not investigated this aspect. Perhaps a more immediate direction for future work is the investigation of applying the automated synthesis and verification method to different fields (besides argumentation).

Appendix A

Negotiation Dialogue

This appendix presents an example of the negotiation dialogue [Sadri et. al., 2001; Sadri et. al., 2002]. The summary of the paper is presented in Section A.1. Section A.2 represents the DID formal definition of the negotiation dialogue. Section A.3 represents the DID of the negotiation dialogue. Section A.4 represents the picture hanging example of the negotiation dialogue. Section A.5 represents the generated LCC protocol from the automated agent protocol synthesis tool "*GenerateLCCProtocol*". Finally, Section A.6 represents the CPN model and verification model properties of the negotiation dialogue.

A.1 Negotiation Dialogue Example

Sadri et. al [Sadri et. al., 2001; Sadri et. al., 2002] work focuses on negotiation dialogue (see chapter 3 section 3 for more details) which allows two agents to request resources or knowledge, propose resource exchanges and suggest alternative resources. Practically, it provides a language as well as a protocol for negotiation dialogues in the domain of resource exchanging that allows each agent in the dialogue to achieve his main goal.

In this negotiation dialogue, there are only two agents. Each agent has only one goal G , one missing resource R , and they have only one plan P to get the missing resource and to achieve its goal. During the dialogue, both agents will try to get the missing resources. In order to achieve this they may suggest alternative plans and resources to each other.

In fact, an agent can open a negotiation dialogue by making a *request* move with the topic (missing resource) R . To terminate a negotiation dialogue an agent must send either *accept* or *refuse* moves [Sadri et. al., 2001; Sadri et. al., 2002].

A.2 DID Formal Definition of the Negotiation dialogue

(7) Players:

In this dialogue, there are two participant: 'A' and 'B'.

Players={A,B}

(8) There are six locutions (Acts):

Acts = {request(R), challenge(R), accept(R), refuse(R), justify(R,S),
promise(R'',R')}

(9) ActType(Act):

Act	ActType (Act)
request	{Starting}
challenge	{ Intermediate }
accept	{Termination}
refuse	{Termination}
justify	{Intermediate}
promise	{Intermediate}

(10) Replies(Act):

In the persuasion dialogue the Replies rules are as follows:

Act	Replies(Act)	Note
request(R)	{challenge(R), accept(R), refuse(R)}	R= missing resource for the speaker
challenge(R)	{justify(R,S)}	S= support for R
accept(R)	∅	
refuse(R)	∅	
justify(R,S)	{refuse(R), promise(R'',R')}	R'= missing resource for the speaker and R''= new resource for new plan for the audience
promise(R'',R')	{accept(R'',R'), refuse(R'',R')}	

(11) **PreC(Act,KB,CS):**

Let Player = A

In the negotiation dialogue the Pre-conditions are as follows:

Act	PreC(Act,KB,CS)	Note
request(R)	miss(KB _A ,R) = true	<ul style="list-style-type: none"> • <i>miss</i> function returns true if agent <i>A</i> misses a resource <i>R</i> for a plan <i>P</i> to achieve a goal <i>G</i>.
challenge(R)	((have(KB _A ,R) and need (KB _A ,R) = true) or notHave(KB _A ,R) = true or missResource(KB _A , P,G) = true)	<ul style="list-style-type: none"> • <i>have</i> function returns true if agent <i>A</i> has a resource <i>R</i>. • <i>need</i> function returns true if agent <i>A</i> has a resource <i>R</i> needed for a plan <i>P</i> to achieve a goal <i>G</i>. • <i>notHave</i> function returns true if agent <i>A</i> does not have a resource <i>R</i>. • <i>missResource</i> function returns true if agent <i>A</i> needs <i>R'</i> resource for a plan <i>P</i> to achieve a goal <i>G</i>.
accept(R)	have (KB _A ,R) = true and notNeed (KB _A ,R) = true and notmissResource(KB _A ,P,G) = true and gaveAway(CS _A ,R) = true	<ul style="list-style-type: none"> • <i>have</i> function returns true if agent <i>A</i> has a resource <i>R</i>. • <i>notNeed</i> function returns true if agent <i>A</i> has a resource <i>R</i> which is not needed for a plan <i>P</i> to achieve a goal <i>G</i>. • <i>notmissResource</i> function returns true if agent <i>A</i> does not miss a resource <i>R'</i> for a plan <i>P</i> to achieve a goal <i>G</i>. • <i>gaveAway</i> function always returns true and results in agent <i>A</i> giving away a resource <i>R</i> (agent <i>A</i> subtract <i>R</i> from its commitment store <i>CS_A</i>).
refuse(R)	(notHave(KB _A ,R) or need(KB _A ,R) = true) and notmissResource(KB _A , P,G) = true	These pre-conditions must be satisfied in order for <i>A</i> to move <i>refuse</i> after <i>request</i> move where, <ul style="list-style-type: none"> • <i>notHave</i> function returns true if agent <i>A</i> does not have a resource <i>R</i>. • <i>need</i> function returns true if agent <i>A</i> has a resource <i>R</i> needed for a plan <i>P</i> to achieve a goal <i>G</i>. • <i>notmissResource</i> function returns true if agent <i>A</i> does not miss a resource <i>R'</i> for a plan <i>P</i> to achieve a goal <i>G</i>.

Bridging the Specification Protocol Gap in Argumentation

Act	PreC(Act,KB,CS)	Note
refuse(R)	missResource(KB _B ,P,G) = true and notExistAlternativePlane(G, without(R,R')) = true	These pre-conditions must be satisfied in order for <i>A</i> to move <i>refuse</i> after <i>justify</i> move where, <ul style="list-style-type: none"> • missResource function returns true if agent <i>A</i> needs <i>R'</i> resource for a plan <i>P</i> to achieve a goal <i>G</i>. • notExistAlternativePlane function returns true if agent <i>A</i> cannot find an alternative plan for agent <i>B</i>'s goal without <i>R</i> and <i>R'</i>.
justify(R,S)	miss(KB _A ,R) = true and getPlan(KB _A ,P) = true and getGoal(KB _A ,G) = true	<ul style="list-style-type: none"> • miss function returns true if agent <i>A</i> needs <i>R</i> resource for a plan <i>P</i> to achieve a goal <i>G</i>. • getPlan function returns true if agent <i>A</i> is able to find a plan <i>P</i> in its Knowledge Base <i>KB_A</i> (<i>A</i> needs <i>R</i> resource for a plan <i>P</i> to achieve a goal <i>G</i>). • getGoal function returns true if agent <i>A</i> is able to find a goal <i>G</i> in its Knowledge Base <i>KB_A</i> (<i>A</i> needs <i>R</i> resource for a plan <i>P</i> to achieve a goal <i>G</i>).
promise(R'',R')	missResource (R', P, G) = true and have (KB _A ,R'') = true and notNeed (KB _A ,R'') = true and choosealternativeplane (KB _A ,G,NewPlan,without(R,R'),with (R'')) = true	<ul style="list-style-type: none"> • R''= missing resource for the speaker <i>A</i> and R''= new resource for new plan for the audience <i>B</i> • missResource function returns true if agent <i>A</i> needs <i>R'</i> resource for a plan <i>P</i> to achieve a goal <i>G</i>. • have function returns true if agent <i>A</i> has a resource R''. • notNeed function returns true if agent <i>A</i> has a resource R'' which is not needed for a plan <i>P</i> to achieve a goal <i>G</i>. • choosealternativeplane function returns true if agent <i>A</i> finds a new and different plan <i>NewPlan</i> for other agent <i>B</i>'s goal that requires neither of R and R' and needs R''.
refuse(R'',R')	miss(KB _A ,R) = true and notChooseBetterPlan(KB _A ,G, NewPlan, oldPlan, without(R,R'), with(R'')) = true	These pre-conditions must be satisfied in order for <i>A</i> to move <i>refuse</i> after <i>promise</i> move where, <ul style="list-style-type: none"> • miss function returns true if agent <i>A</i> has a resource <i>R</i> needed for a plan <i>OldPlan</i> (<i>P</i>) to achieve a goal <i>G</i>.

		<ul style="list-style-type: none"> • <i>notChooseBetterPlan</i> function compare the <i>OldPlan</i> and the <i>NewPlan</i> and returns true if <i>NewPlan</i> is not acceptable.
accept(R'',R')	miss(KB _A ,R) = true and have(KB _A ,R') = true and notNeed(KB _A ,R') = true and chooseBetterPlan(KB _A ,G,NewPlan, OldPlan,without(R,R'),with(R'')) = true and gaveaway(CS _A , R') = true and obtained(CS _A ,R'')= true	These pre-conditions must be satisfied in order for <i>A</i> to move <i>accept</i> after <i>promise</i> move where, <ul style="list-style-type: none"> • <i>miss</i> function returns true if agent <i>A</i> has a resource <i>R</i> needed for a plan <i>OldPlan</i> (<i>P</i>) to achieve a goal <i>G</i>. • <i>have</i> function returns true if agent <i>A</i> has a resource <i>R'</i>. • <i>notNeed</i> function returns true if agent <i>A</i> has a resource <i>R'</i> which is not needed for a plan <i>P</i> to achieve a goal <i>G</i>. • <i>chooseBetterPlan</i> function compare the <i>OldPlan</i> and the <i>NewPlan</i> and returns true if agent the <i>NewPlan</i> (that requires neither of <i>R</i> and <i>R'</i> and needs <i>R''</i>) is acceptable. • <i>gaveAway</i> function always returns true and results in agent <i>A</i> giving away a resource <i>R'</i> (agent <i>A</i> subtract <i>R'</i> from its commitment store <i>CS_A</i>). • <i>obtained</i> function always returns true and results in agent <i>A</i> obtaining a resource <i>R''</i> (agent <i>A</i> adding <i>R''</i> to its commitment store <i>CS_A</i>).

(12) PostC(Act,KB,CS):

let Player(M_t)= *A* and NextPlayer =*B*,

In a negotiation dialogue the Post-Conditions (conditions for receiver player *B* of M_t) are as follows:

Act	PostC(Act,KB,CS)	Note
request(R)	true	
challenge(R)	true	
accept(R)	obtained (CS _B ,R) = true	<ul style="list-style-type: none"> • <i>obtained</i> function always returns true and results in agent <i>B</i> obtaining a resource <i>R</i> (agent <i>B</i> adding <i>R</i> to its commitment store <i>CS_B</i>).

Act	PostC(Act,KB,CS)	Note
refuse(R)	true	
justify(R,S)	true	
promise(R'',R')	true	
accept(R'',R')	obtained(CS _B , R') = true and gaveaway(CS _B ,R'') = true	These post-conditions must be satisfied in order for <i>A</i> to move <i>accept</i> after <i>promise</i> move where, <ul style="list-style-type: none"> • <i>gaveAway</i> function always returns true and results in agent <i>B</i> giving away a resource <i>R''</i> (agent <i>B</i> subtract <i>R''</i> from its commitment store <i>CS_B</i>). • <i>obtained</i> function always returns true and results in agent <i>B</i> obtaining a resource <i>R'</i> (agent <i>B</i> adding <i>R'</i> to its commitment store <i>CS_B</i>).
refuse(R'',R')	true	

(13) LegalMoves(M_t , CS_A, CS_B)

From Figure A.1 the negotiation dialogue, we can see that:

- Dialogues open by making a *request* move

M₁ = initial move, ActType(Act(M₁)) = Starting and Act(M₁)= {request}

- In the negotiation dialogue, the argument terminates once the agents send *accept* or *refuse*. In other words, both *accept* and *refuse* ∈ {Termination}. There is no reply move to these moves (there are no arrows coming out from these moves).
- *Challenge*, *justify* and *promise* ∈ {Intermediate}. There are several moves to these moves (there are arrows coming out from these moves).
- The turn-taking between participants switches after each move:

a) if M₁ then Player = A,

b) else NextPlayer = B iff Player = A

and NextPlayer = A iff Player = B

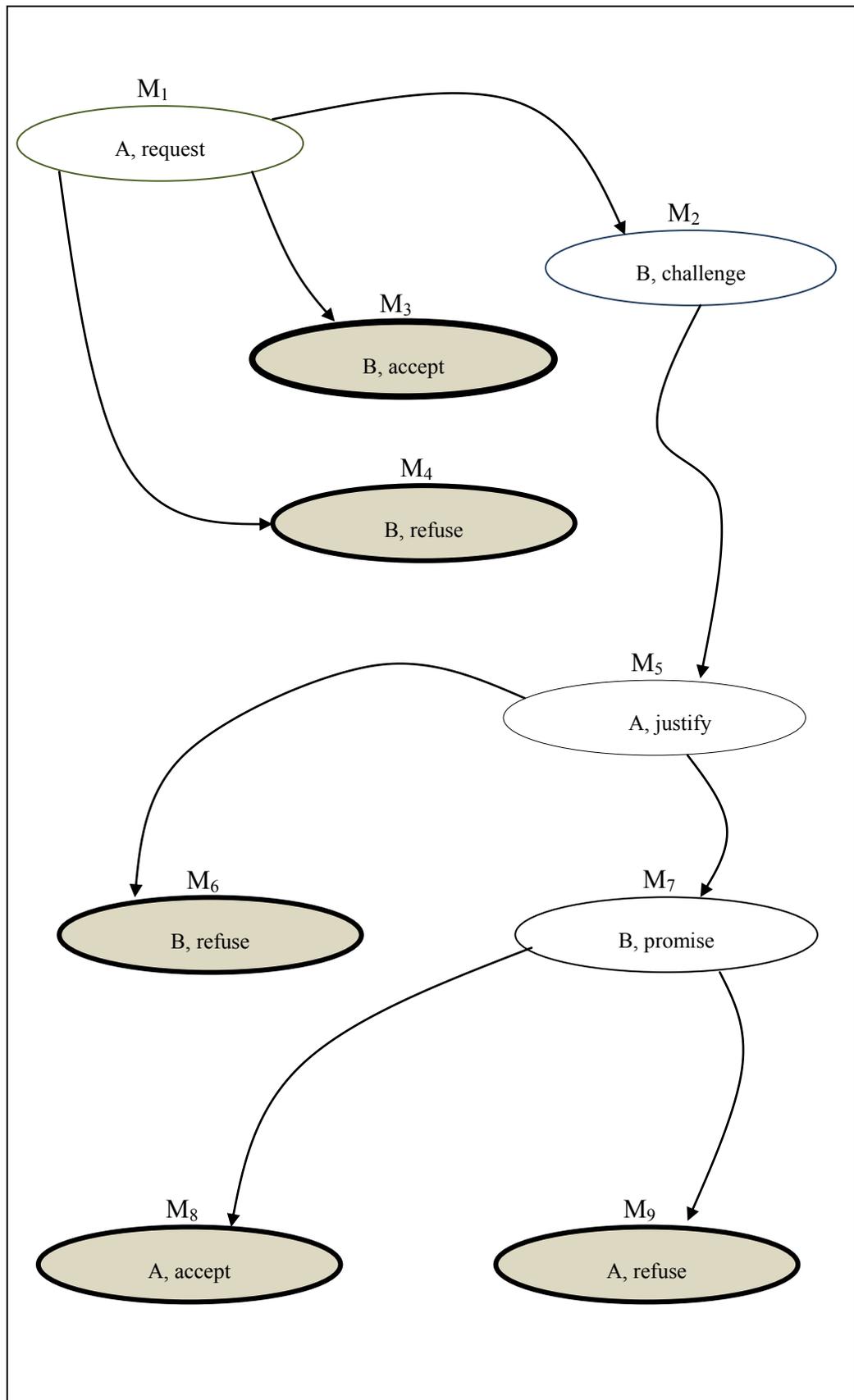


Figure A.1: The Negotiation Dialogue Legal Moves

A.3 DID of the Negotiation Dialogue

Figure A.2 illustrates a DID structure of a negotiation dialogue (Note that pre-conditions and post-conditions for locutions are not shown in this figure. Rather, it is shown in Figure A.3(a), Figure A.3(b), Figure A.3(c), and Figure A.3(d). In Figure A.2, there are six locutions: *request*, *challenge*, *accept*, *refuse*, *justify* and *promise* locutions (a subset of locutions in [Amogud et.al. 2000]²²). There are three types of locution: starting (*request*), termination (*accept* and *refuse*), and intermediate (*challenge*, *justify* and *promise*) locution.

In this example, a dialogue always starts with a *request* and ends with an *accept* or *refuse* locution. *A* can open the dialogue by sending a *request*(R) locution if he is able to satisfy the condition which is connected to the sender role of this locution. Then, turn-taking switches to *B*. *B* has to choose between three different possible reply locutions: *challenge*(R), *accept*(R) or *refuse*(R). *B* will make his choice using the conditions which appear in the rhombus shape (for example, in order to choose *challenge* (R), *B* must be able to satisfy the two conditions which connect with *challenge*). After that, the turn switches to *A*, and so on. The argument terminates once an agent sends either an *accept* or *refuse* locution.

A.4 The Picture Hanging Example

Figure A.4 represents the negotiation dialogue graph of the picture hanging example (adapted from [Parsons et al., 1998; Maudet et al., 2007]) (see chapter 3 for more details):

- (14) Dialogue takes place between two agents, *A* and *B*.
- (15) *A* has KB_A and CS_A , and *B* has KB_B and CS_B (Note that the agent's knowledge bases are shown at the top of the figure).

²² In this example, we follow the Commitment rules in Amogud et.al [Amogud et.al. 2000] work).

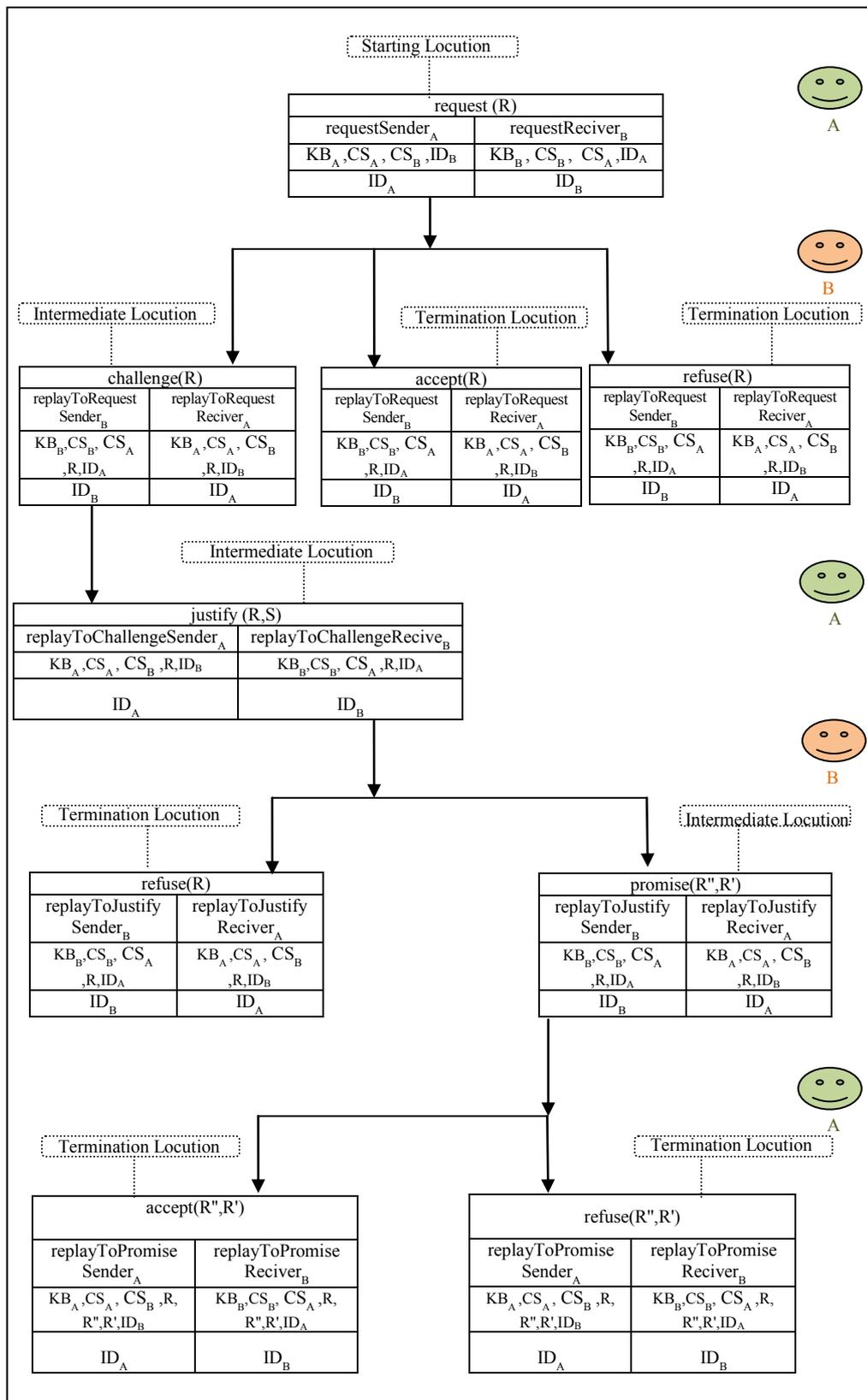


Figure A.2: DID Structure of a Negotiation Dialogue

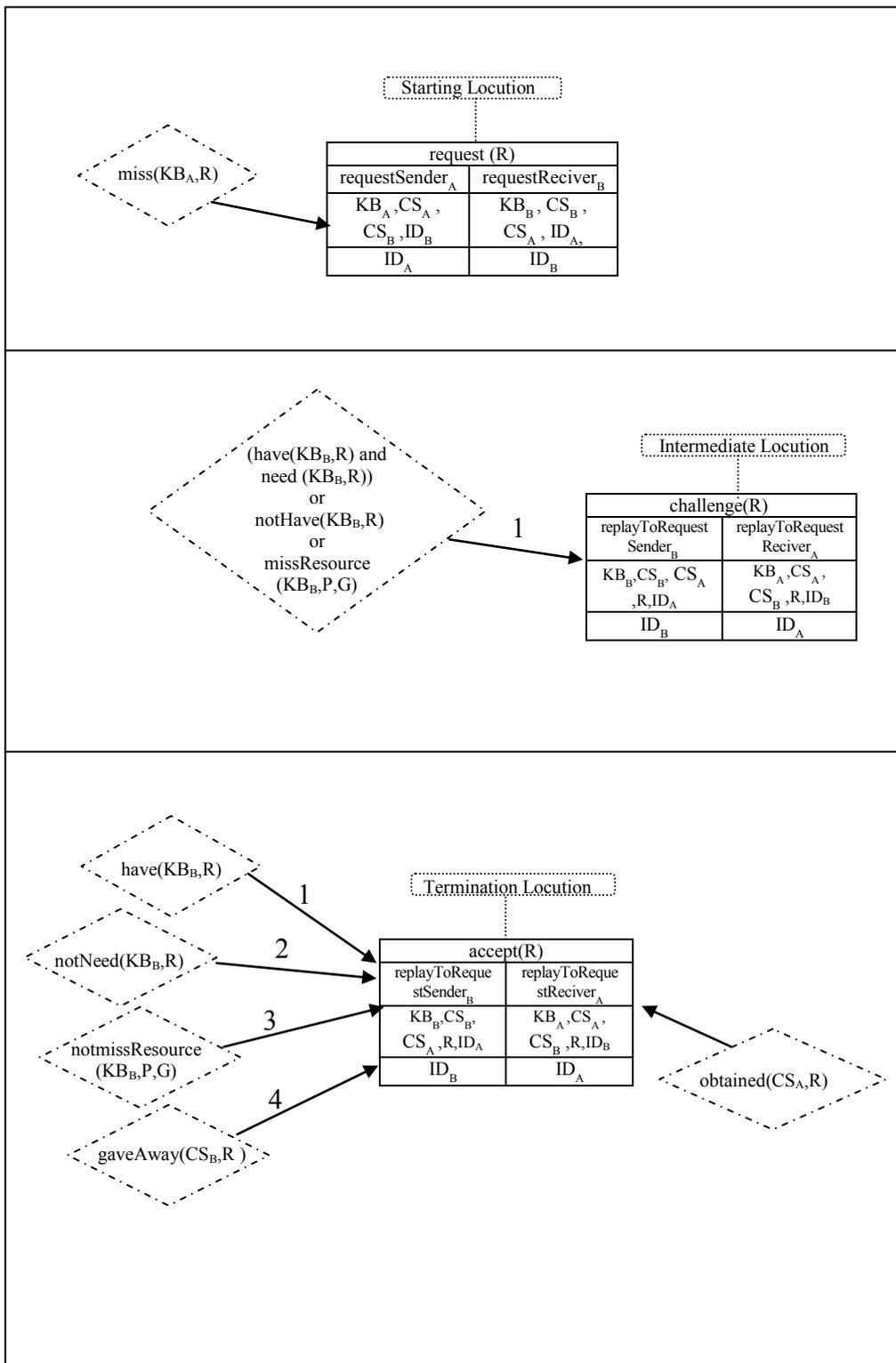


Figure A.3(a): Negotiation Dialogue Locutions Pre-Conditions and Post-Conditions

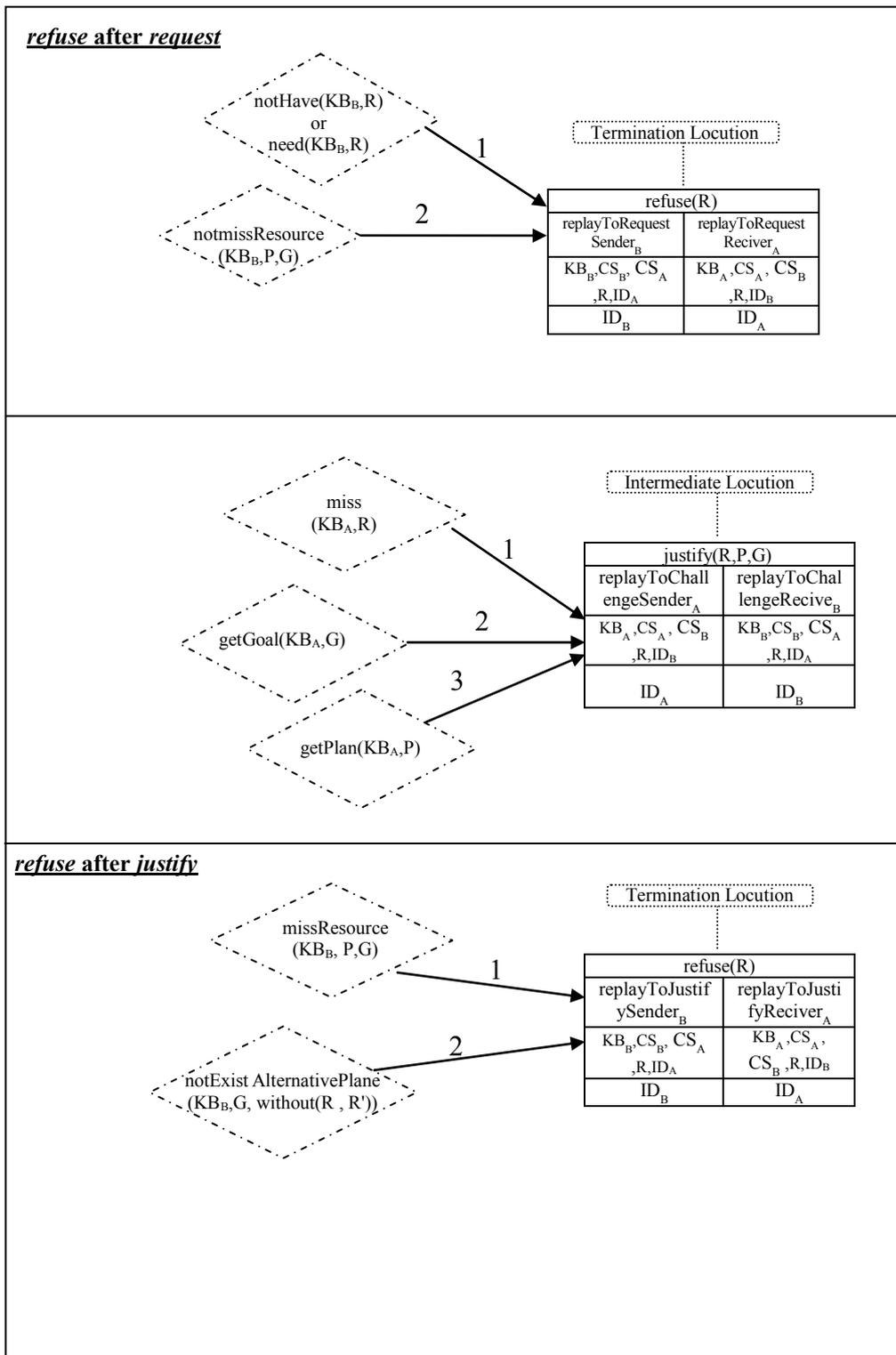


Figure A.3 (b): Negotiation Dialogue Locutions Pre-conditions and Post-conditions

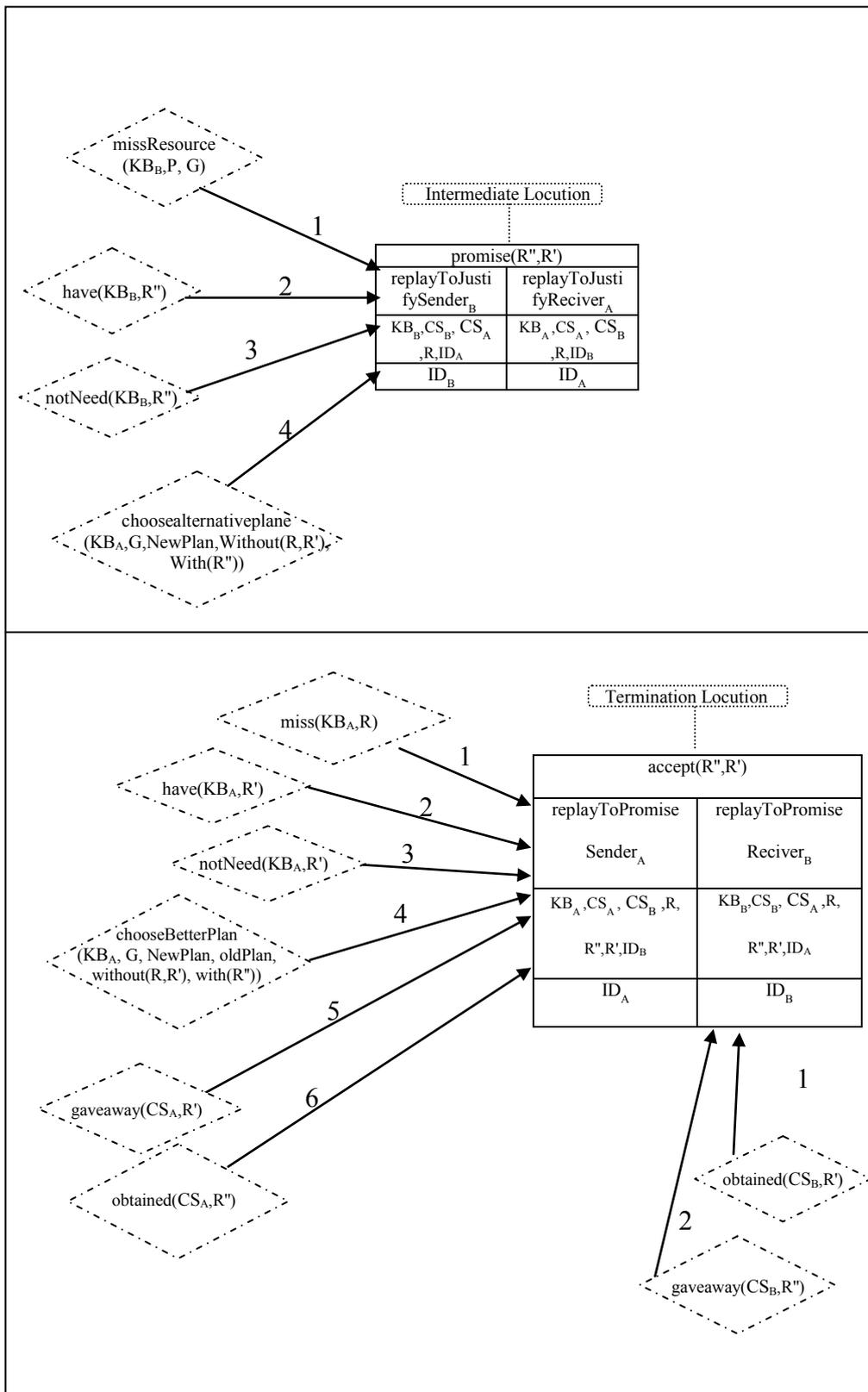


Figure A.3 (c): Negotiation Dialogue Locutions Pre-conditions and Post-conditions

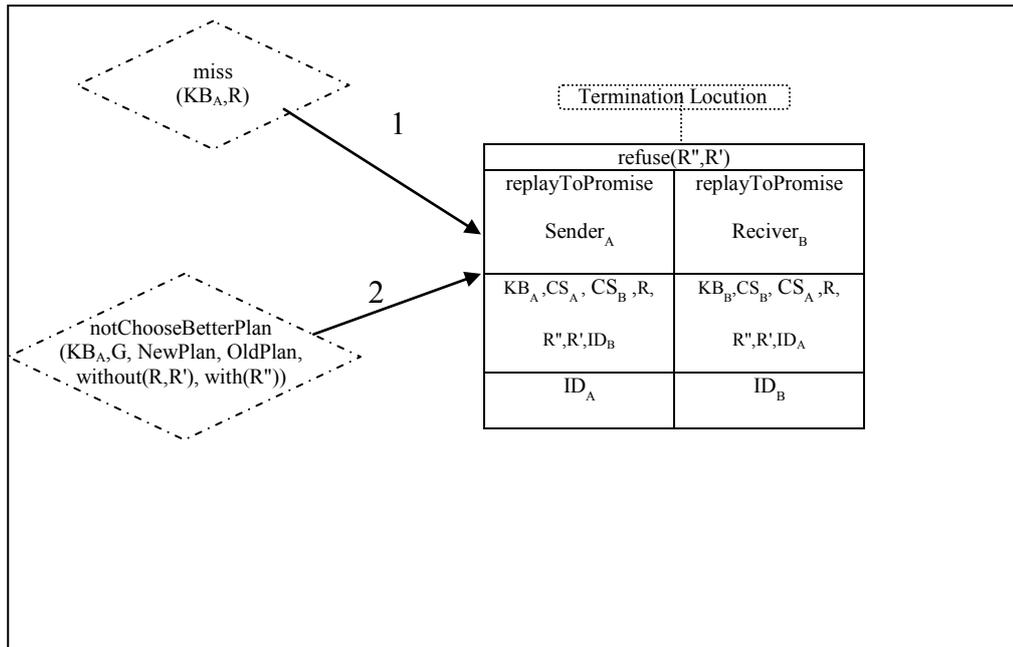


Figure A.3 (d): Negotiation Dialogue Locutions Pre-conditions and Post-conditions

- (16) A and B can access CS_A and CS_B .
- (17) The goal of the dialogue is to exchange knowledge (resources), since an agent's knowledge is not sufficient to achieve its own goals. The goal of A is to hang a picture and the goal of B is to hang a mirror. A has a hammer. However, to hang the picture A needs a nail in addition to the hammer. In contrast, B has a nail, screw and screw-driver. B needs a hammer, in addition to the nail, to hang the mirror. A plans to get a nail from B and B plans to get a hammer from A .
- (18) A begin the discussion by sending *request*("Can you please give me a nail?").
- (19) B consults its argumentation system AS_B ($AS_B = \{KB_B, CS_B\}$) whether he has a nail or not, and if he has a nail does he need it. In this example, B finds that he has a nail and needs to hang a mirror.
- (20) B challenges "Can you please give me a nail?". In others words, he asks the reason behind A 's request of "a nail". In this example, B will challenge "Can you please give me a nail?" by sending *challenge*("why do you need a nail?") locution.

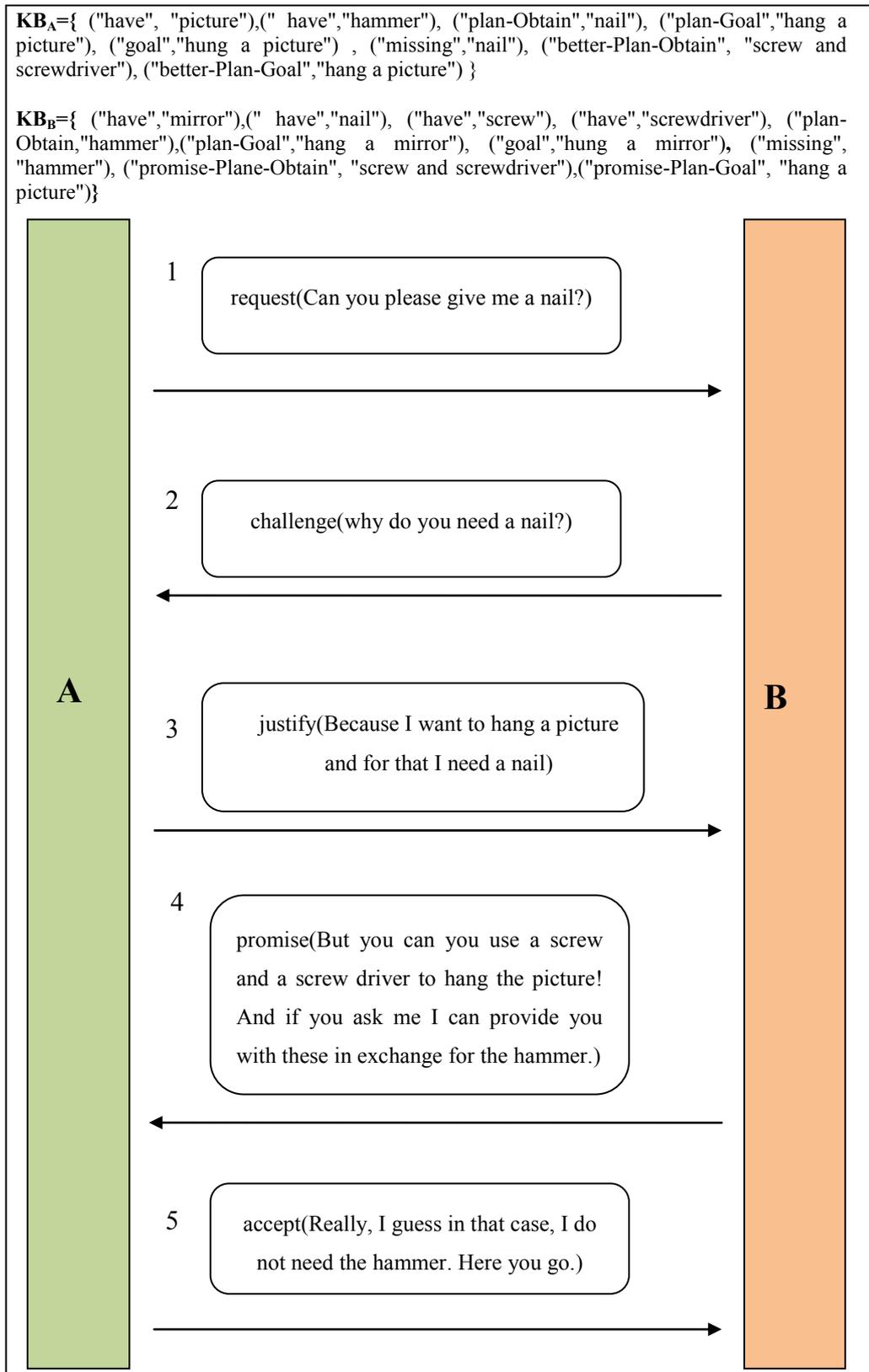


Figure A.4: The Picture Hanging Example

- (21) *A* responds to the challenge by declaring the supporting premises *S* (*S*=*A*'s goal and *A*'s plan) for "*Can you please give me a nail?*". In this example, *A* offers a reason for the *request* by sending *justify*("Because I want to hang a picture and for that I need a nail") locution.
- (22) *B* checks with its argumentation system AS_B whether he could provide an alternate plan for *A* that allows both *A* and *B* to achieve their goal. In this example *B* finds a new plan for *A*'s goal and sends *promise*("But you can you use a screw and a screw driver to hang the picture! And if you ask me I can provide you with these in exchange for a hammer") locution.
- (23) *A* checks with its argumentation system AS_A whether the new plan is acceptable (whether the new plan is better than the old plan or not). In this example, *A* finds that it is acceptable and accepts the new plan by sending *accept*("Really, I guess in that case, I do not need the nail. Here you go") locution.
- (24) The commitment stores of *A* and *B* at the end of the dialogue are:
- $CS_A = \{("gaveAway", "hammer"), ("obtained", "screw and screwdriver")\}$
 - $CS_B = \{("obtained", "hammer"), ("gaveAway", "screw and screwdriver")\}$

A.5 LCC Synthesis Protocol of the Negotiation Dialogue

This section represents the generated LCC protocol from the automated agent protocol synthesis tool "*GenerateLCCProtocol*". In this example, the tool receives as input the DID of the negotiation dialogue, which is shown in Figure A.2, and then the tool generates the negotiation dialogue LCC protocol by using LCC-Argument patterns. The final LCC protocol is illustrated in Figure A.5(a) and Figure A.5(b):

- (1) The tool begins with the locution icon at the top of the DID of the negotiation dialogue, which is *request*(*R*).

Bridging the Specification Protocol Gap in Argumentation

Agent A	Agent B
<p>a(requestSender_A(KB_A,CS_A,CS_B,ID_B),ID_A))::=</p> <p>request(R) => a(requestReceiver_B(KB_B,CS_B,CS_A,ID_A),ID_B) \leftarrow miss(KB_A,R)</p> <p>then</p> <p>a(replyToRequestReceiver_A(KB_A,CS_A,CS_B,R,ID_B),ID_A).</p>	<p>a(requestReceiver_B(KB_B,CS_B,CS_A,ID_A),ID_B))::=</p> <p>request(R) <= a(requestSender_A(KB_A,CS_A,CS_B,ID_B),ID_A)</p> <p>then</p> <p>a(replyToRequestSender_B(KB_B,CS_B,CS_A,R,ID_A),ID_B).</p>
<p>a(replyToRequestReceiver_A(KB_A,CS_A,CS_B,R,ID_B),ID_A))::=</p> <p>obtained(CS_A,R) \leftarrow accept(R) <=</p> <p>a(replyToRequestSender_B(KB_B,CS_B,CS_A,R,ID_A),ID_B)</p> <p>or</p> <p>refuse(R) <=</p> <p>a(replyToRequestSender_B(KB_B,CS_B,CS_A,R,ID_A),ID_B)</p> <p>or</p> <p>(</p> <p>challenge(R) <=</p> <p>a(replyToRequestSender_B(KB_B,CS_B,CS_A,R,ID_A),ID_B)</p> <p>then</p> <p>a(replyToChallengeSender_A(KB_A,CS_A,CS_B,R,ID_B),ID_A).</p>	<p>a(replyToRequestSender_B(KB_B,CS_B,CS_A,R,ID_A),ID_B))::=</p> <p>accept(R) =></p> <p>a(replyToRequestReceiver_A(KB_A,CS_A,CS_B,R,ID_B),ID_A)</p> <p>\leftarrow have(KB_B,R) and notNeed(KB_B,R) and notmissResource(KB_B,P,G) and gaveAway(CS_B,R)</p> <p>or</p> <p>refuse(R) =></p> <p>a(replyToRequestReceiver_A(KB_A,CS_A,CS_B,R,ID_B),ID_A)</p> <p>\leftarrow (notHave(KB_B,R) or need(KB_B,R)) and notmissResource(KB_B,P,G)</p> <p>or</p> <p>(</p> <p>challenge(R) =></p> <p>a(replyToRequestReceiver_A(KB_A,CS_A,CS_B,R,ID_B),ID_A) \leftarrow (have(KB_B,R) and need (KB_B,R)) or notHave(KB_B,R)) and missResource(KB_B,P,G)</p> <p>then</p> <p>a(replyToChallengeReceiver_B(KB_B,CS_B,CS_A,R,ID_A),ID_B).</p>
<p>a(replyToChallengeSender_A(KB_A,CS_A,CS_B,R,ID_B),ID_A))::=</p> <p>justify(R,P,G) =></p> <p>a(replyToChallengeReceiver_B(KB_B,CS_B,CS_A,R,ID_A),ID_B) \leftarrow miss(KB_A,R) and getGoal(KB_A,G) and getPlan(KB_A,P)</p> <p>then</p> <p>a(replyToJustifyReceiver_A(KB_A,CS_A,CS_B,R,ID_B),ID_A).</p>	<p>a(replyToChallengeReceiver_B(KB_B,CS_B,CS_A,R,ID_A),ID_B))::=</p> <p>justify(R,P,G) <=</p> <p>a(replyToChallengeSender_A(KB_A,CS_A,CS_B,R,ID_B),ID_A)</p> <p>then</p> <p>a(replyToJustifySender_B(KB_B,CS_B,CS_A,R,ID_A),ID_B).</p>

Figure A.5(a): Generated LCC Protocol

Agent A	Agent B
<p>a(replyToJustifyReceiver_A(KB_A,CS_A,CS_B, R, ID_B), ID_A))::=</p> <p>refuse(R) <= a(replyToJustifySender_B(KB_B,CS_B, CS_A,R, ID_A), ID_B)</p> <p>or</p> <p>(</p> <p>addToCS(CS_A,R") ← promise(R",R') <=</p> <p>a(replyToJustifySender_B(KB_B,CS_B, CS_A,R, ID_A), ID_B)</p> <p>then</p> <p>a(replyToPromiseSender_A(KB_A,CS_A, CS_B, R, R",R', ID_B), ID_A)).</p>	<p>a(replyToJustifySender_B(KB_B,CS_B, CS_A,R, ID_A), ID_B))::=</p> <p>refuse(R) => a(replyToJustifyReceiver_A(KB_A,CS_A, CS_B, R, ID_B), ID_A)</p> <p>←</p> <p>(</p> <p>missResource(KB_B, P, G)</p> <p>and</p> <p>notExistAlternativePlane(KB_B, G, without(R, R'))</p> <p>)</p> <p>or</p> <p>(</p> <p>promise(R",R') =></p> <p>a(replyToJustifyReceiver_A(KB_A,CS_A, CS_B, R, ID_B), ID_A)</p> <p>←</p> <p>(</p> <p>missResource (KB_B, P, G)</p> <p>and have(KB_B, R")</p> <p>and notNeed(KB_B, R") and choosealternativeplane (KB_B, G, NewPlan, Without(R, R'), With(R"))</p> <p>)</p> <p>then</p> <p>a(replToPromiseReceiver_B(KB_B,CS_B, CS_A, R, R",R', ID_A), ID_B)).</p>
<p>a(replyToPromiseSender_A(KB_A,CS_A,CS_B, R, R",R', ID_B), ID_A))::=</p> <p>(</p> <p>accept(R",R') =></p> <p>a(replToPromiseReceiver_B(KB_B,CS_B, CS_A, R, R",R', ID_A), ID_B)</p> <p>←</p> <p>(</p> <p>miss(KB_A, R) and</p> <p>have(KB_A, R') and notNeed(KB_A, R') and</p> <p>chooseBetterPlan</p> <p>(KB_A, G, NewPlan, oldPlan, without(R, R'), with(R"))</p> <p>and giveaway(CS_A, R') and obtained(CS_A, R")</p> <p>)</p> <p>or</p> <p>(</p> <p>refuse(R",R') =></p> <p>a(replToPromiseReceiver_B(KB_B,CS_B, CS_A, R, R",R', ID_A), ID_B) ← miss(KB_A, R) and</p> <p>notChooseBetterPlan</p> <p>(KB_A, G, NewPlan, OldPlan, without(R, R'), with(R"))</p> <p>)</p>	<p>a(replToPromiseReceiver_B(KB_B,CS_B, CS_A, R, R",R', ID_A), ID_B))::=</p> <p>(</p> <p>(obtained(CS_B, R') and giveaway(CS_B, R"))</p> <p>← accept(R",R') <=</p> <p>a(replyToPromiseSender_A(KB_A,CS_A, CS_B, R, R",R', ID_B), ID_A)</p> <p>)</p> <p>or</p> <p>(</p> <p>refuse(R",R') <=</p> <p>a(replyToPromiseSender_A(KB_A,CS_A, CS_B, R, R",R', ID_B), ID_A)</p> <p>)</p>

Figure A.5(b): Generated LCC Protocol

("goal", "hung mirror"), ("missing", "hammer"), ("promisePlanObtain", "screw"), ("promisePlanGoal", "hang picture")], "requestReceiverB", "", "", [], "IDA", "", "", ""). This place represents agent *B* arguments.

Step One: Automated Transformation from LCC to CPN/XML

The generated LCC protocol for negotiation dialogue in Figures A.5(a) and A.5(b) was used as input to the verification tool. The verification tool generated a negotiation dialogue CPN/XML file which has:

- (1) Ten CPN subpages generated by the *GenerateLCCProtocol* tool (subpage for each LCC role in the Figures A.5(a) and A.5(b)). See Figures A.6, A.7, A.8, A.9, A.10, A.11, A.12, A.13, A.14 and A.15.
- (2) One CPN superpage generated by the *GenerateLCCProtocol* tool. This page connects the ten CPN subpages (*requestSenderA*, *requestReceiverB*, *replyToRequestSenderB*, *replyToRequestSenderB*, *replyToChallengeSenderA*, *replyToChallengeReceiverB*, *replyToJustifySenderB*, *replyToJustifyReceiverA*, *replyToPromiseSenderA* and *replyToPromiseReceiverB*) together and describes the interaction between these ten subpages. See Figure A.16.

The CPN model generated by the verification tool for the negotiation dialogue was not completed. It needed manual translations of LCC protocol message conditions to guards (SML conditions) in the CPN model. These translations had to be done manually because the LCC conditions code is not in the LCC protocol file [Robertson, 2004; Hassan et.al., 2005].

Step Two: Construction of State Space

After finishing manual translations of the LCC protocol message in the last step, the state space (shown in Figure A.17) for the CPN model of an LCC protocol for a negotiation dialogue was generated using the SS tool palette in CPN Tools (see chapter 6, section 6.2).

Bridging the Specification Protocol Gap in Argumentation

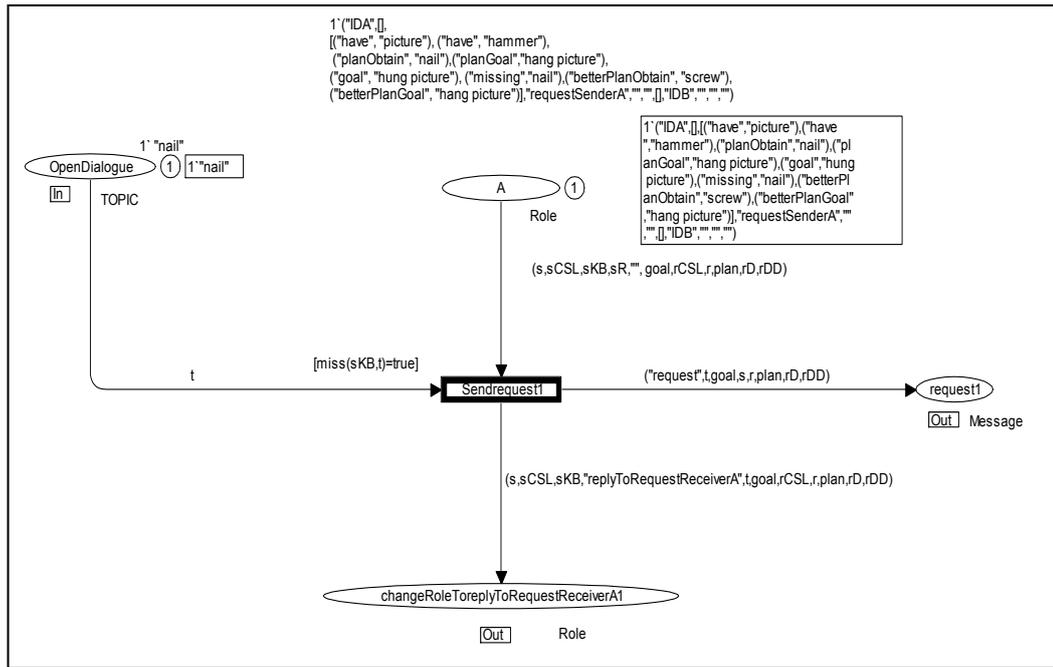


Figure A.6: The requestSenderA CPN Subpage

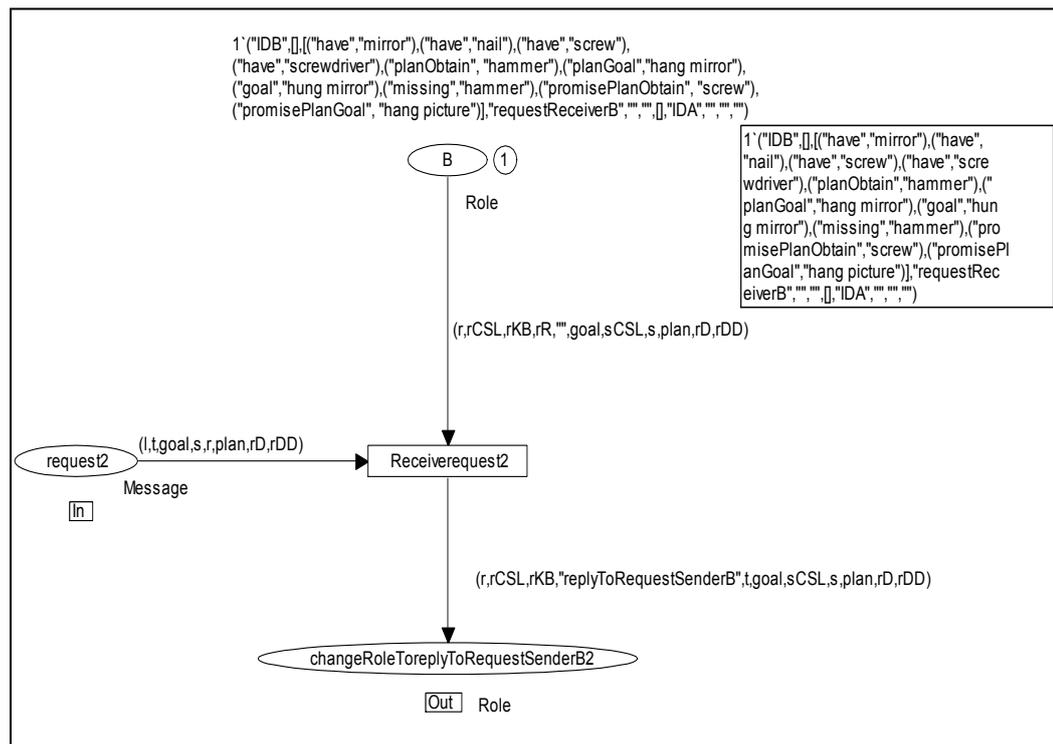


Figure A.7: The requestReceiverB CPN Subpage

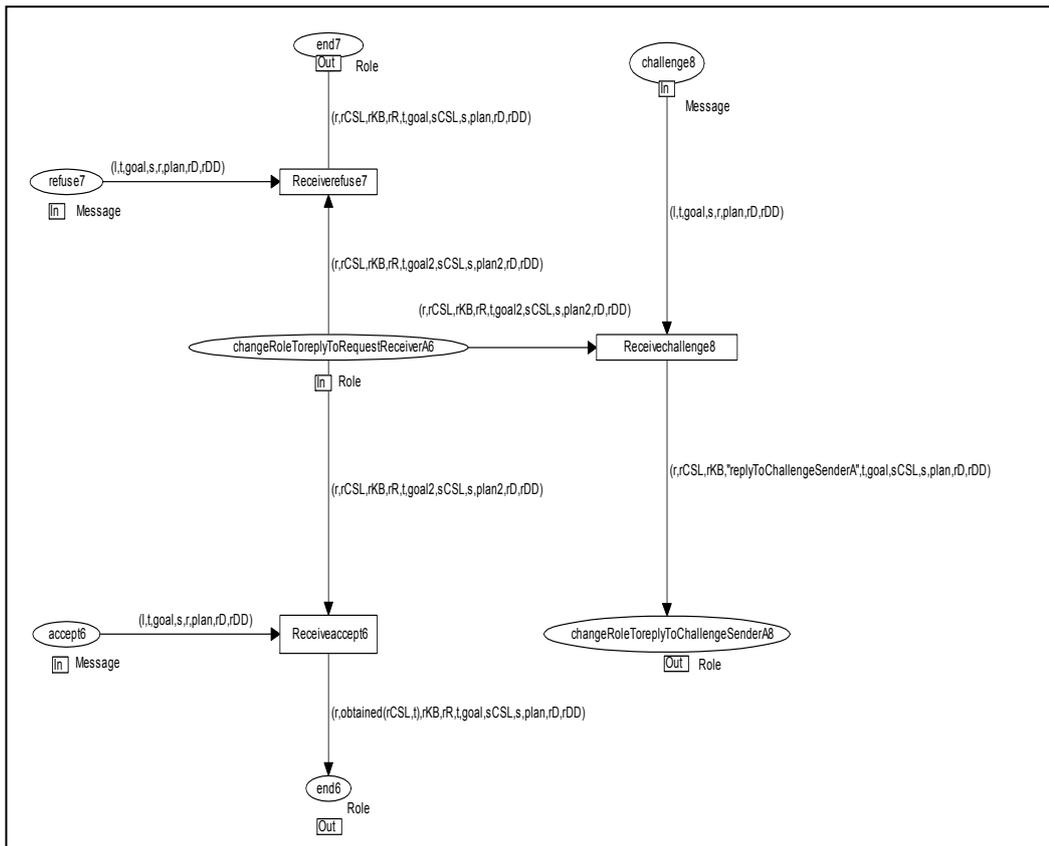


Figure A.9: The replyToRequestSenderB CPN Subpage

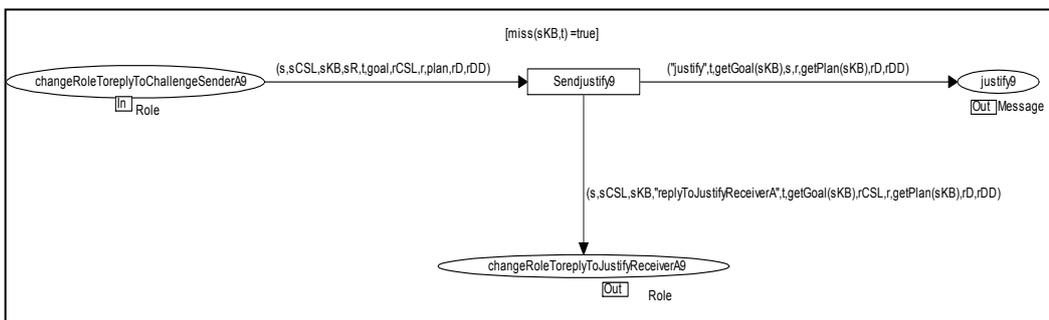


Figure A.10: The replyToChallengeSenderA CPN Subpage

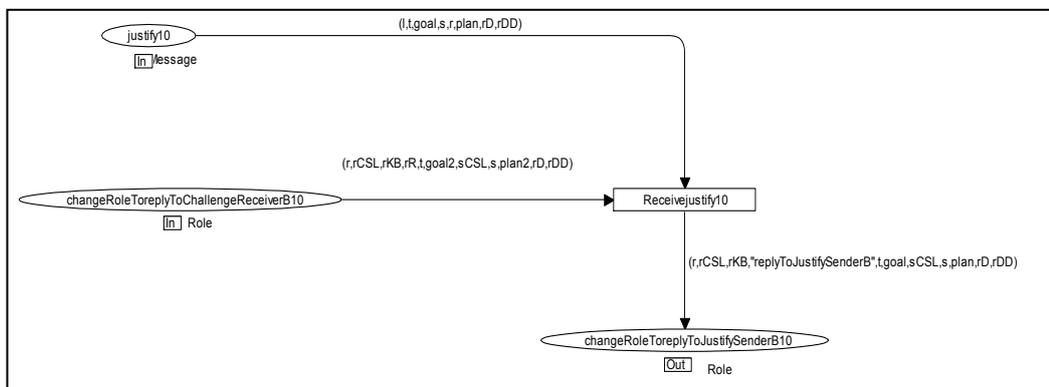


Figure A.11: The replyToChallengeReceiverB CPN Subpage

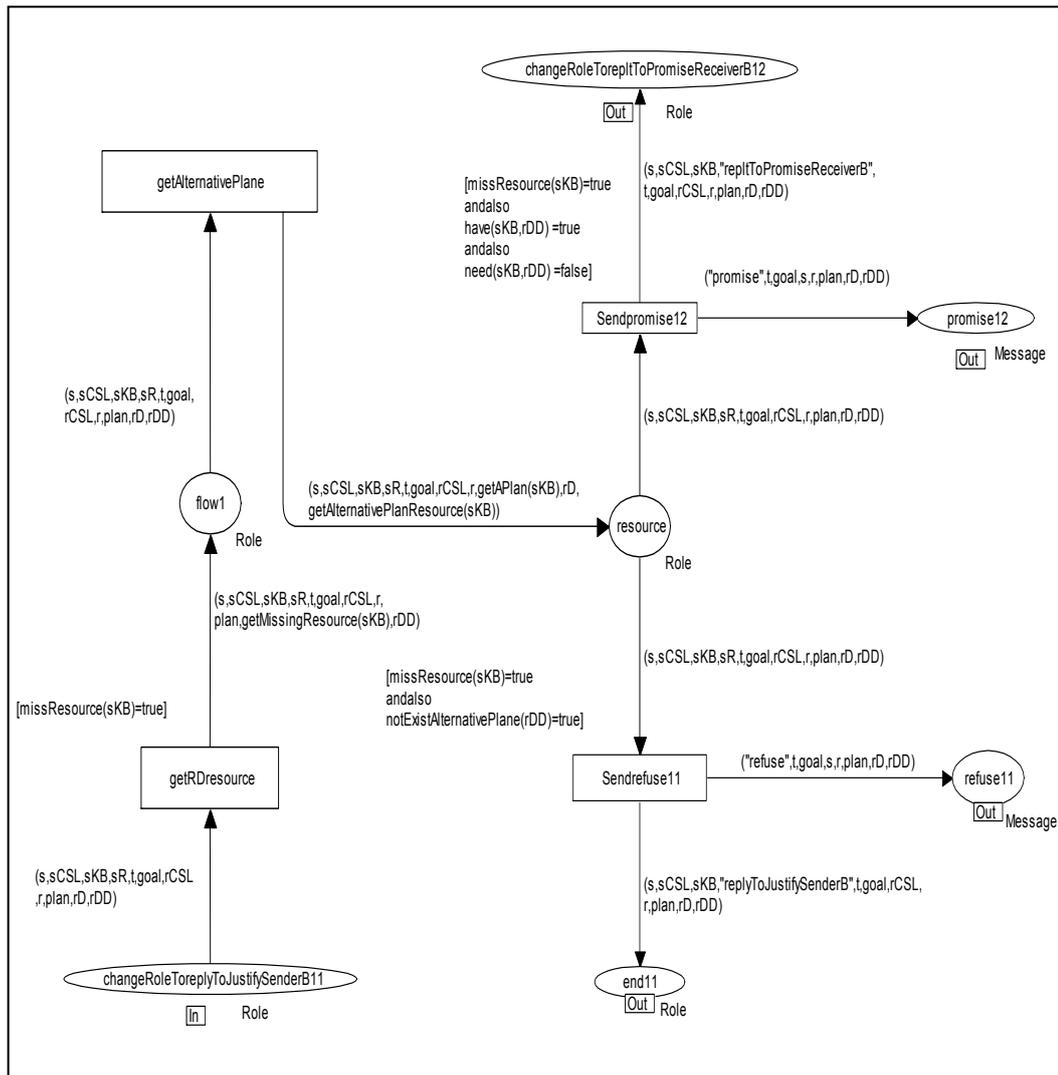


Figure A.12: The *replyToJustifySenderB* CPN Subpage

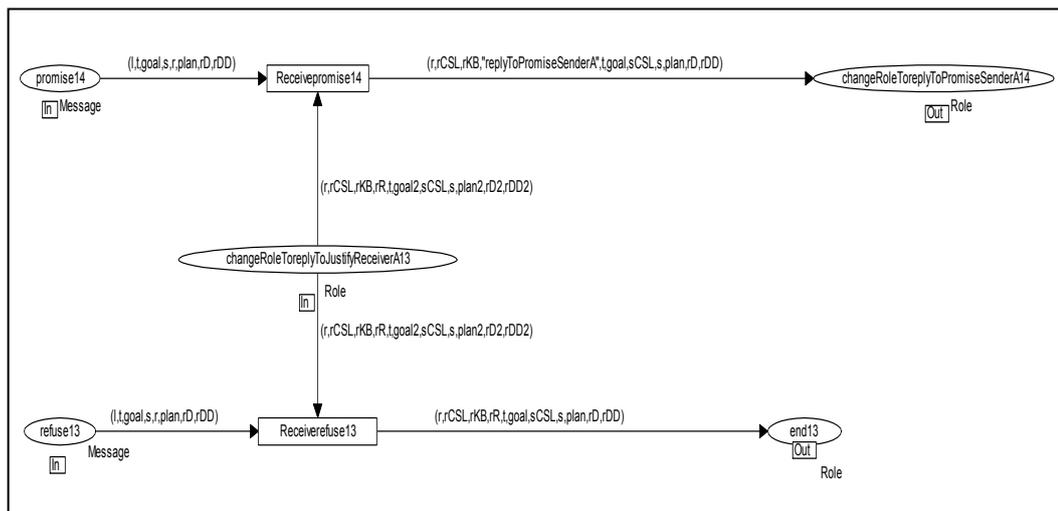


Figure A.13: The *replyToJustifyReceiverA* CPN Subpage

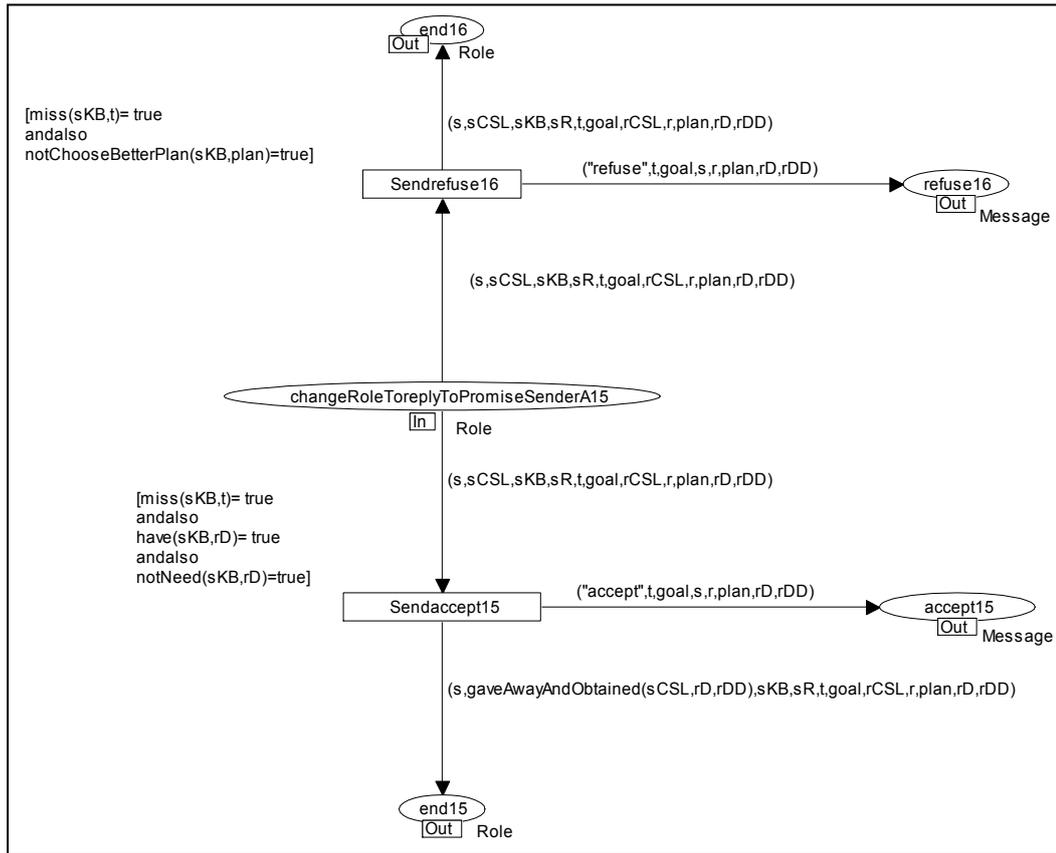


Figure A.14: The `replyToPromiseSenderA` CPN Subpage

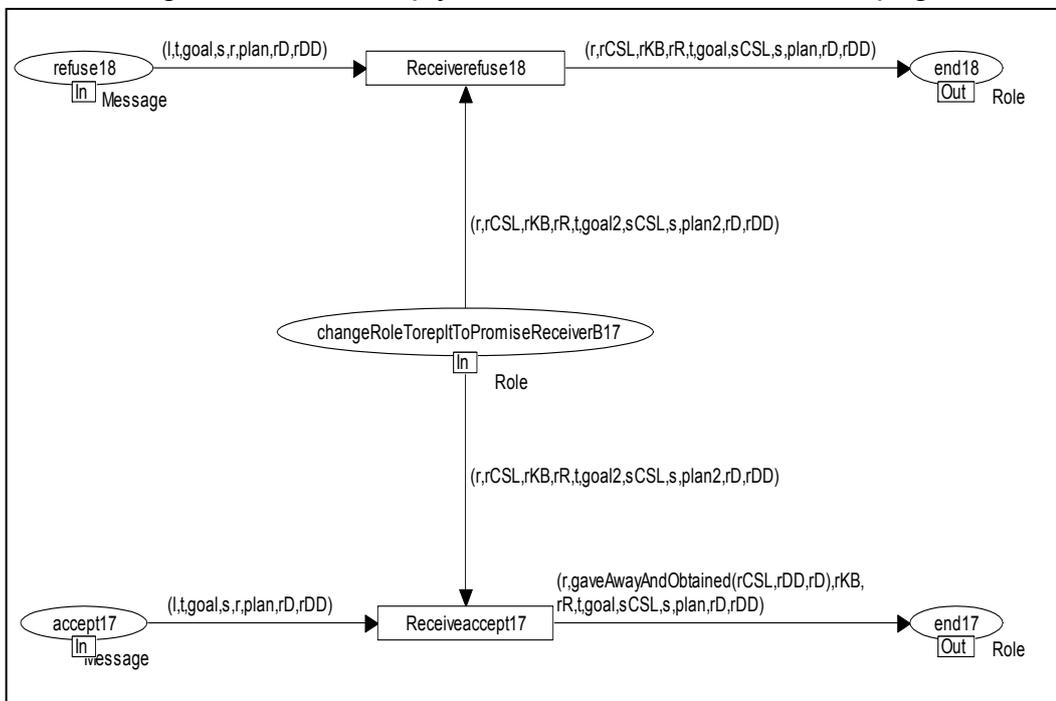


Figure A.15: The `replyToPromiseReceiverB` CPN Subpage

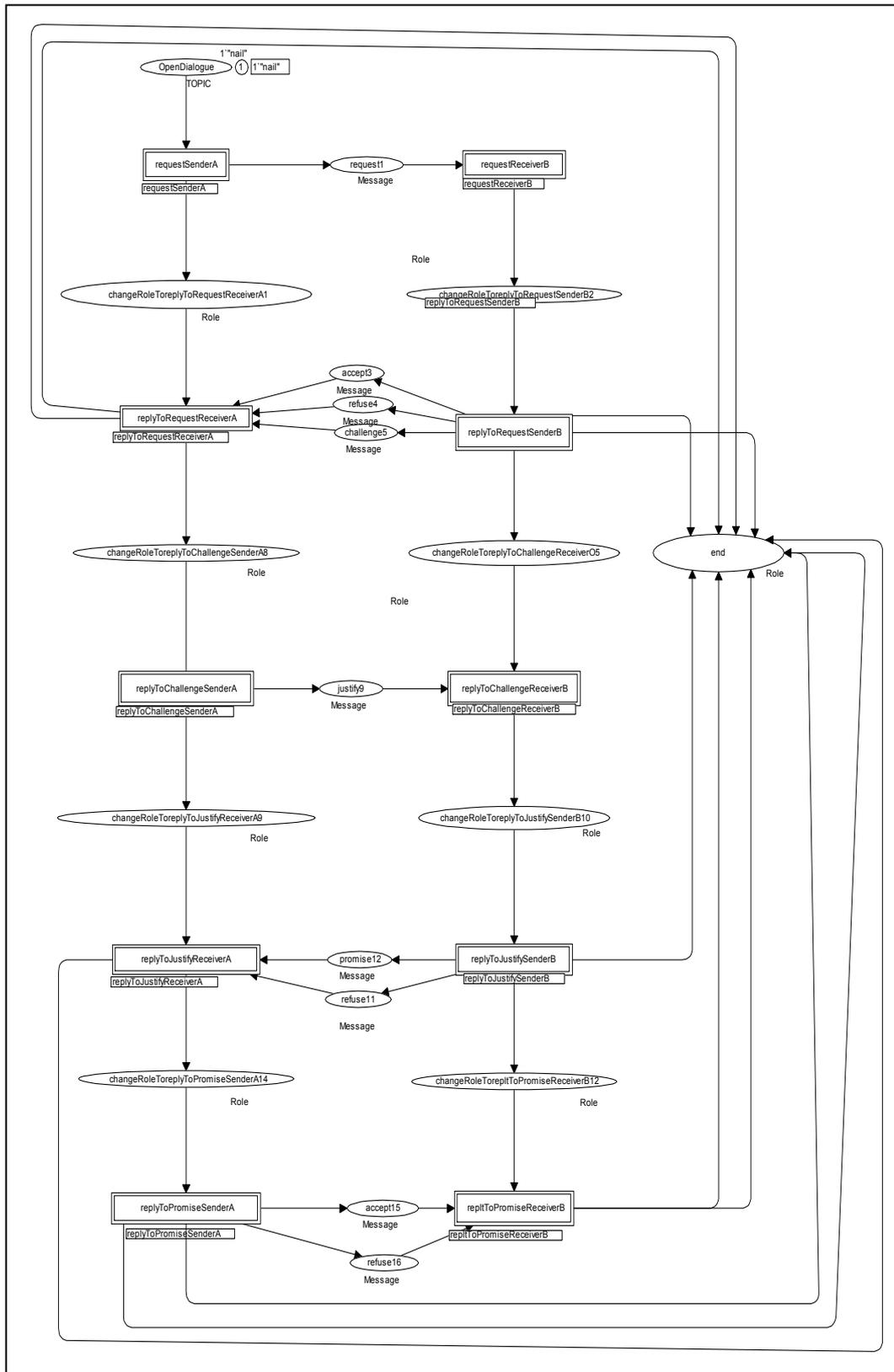


Figure A.16: The protocol CPN Superpage

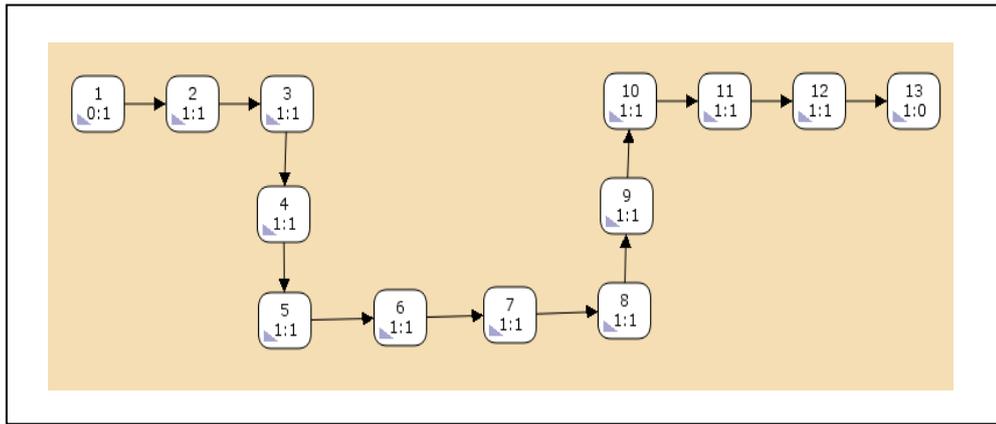


Figure A.17: The State Space Graph

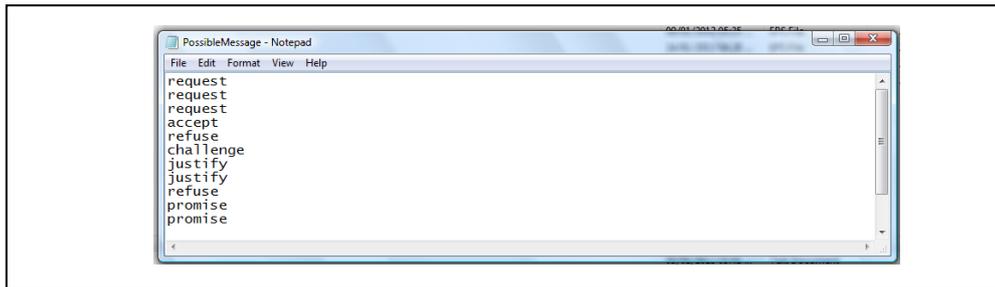


Figure A.18: Possible Locutions File

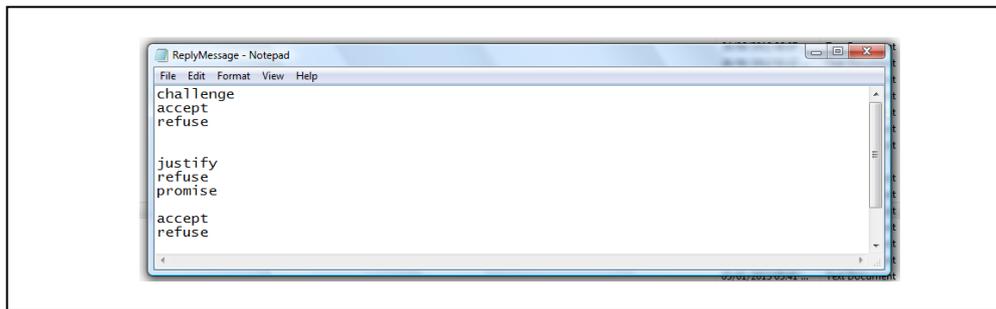


Figure A.19: Reply Locutions File

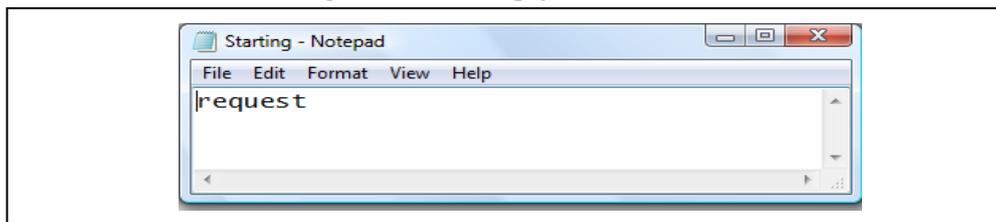


Figure A.20: Starting Locutions File

Step Three: Automated Creation of DID Properties

In this step, the verification tool succeeded in automatically creating the nine property files. See Figures A.18, A.19, A.20, A.21, A.22, A.23, A.24, A.25 and A.26.

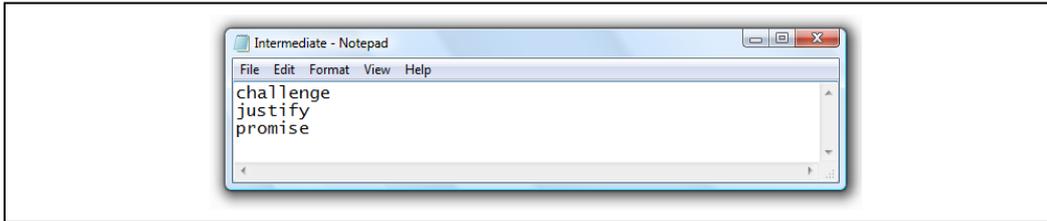


Figure A.21: Intermediate Locutions File

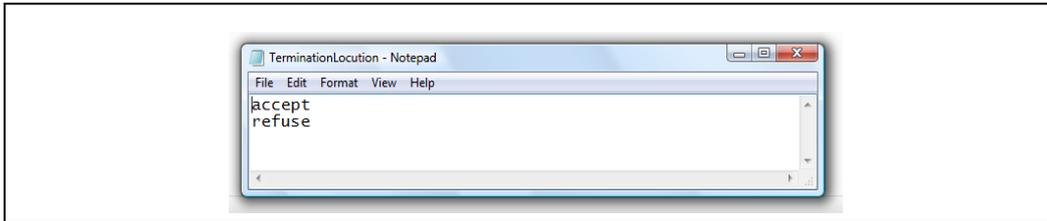


Figure A.22: Termination Locutions File

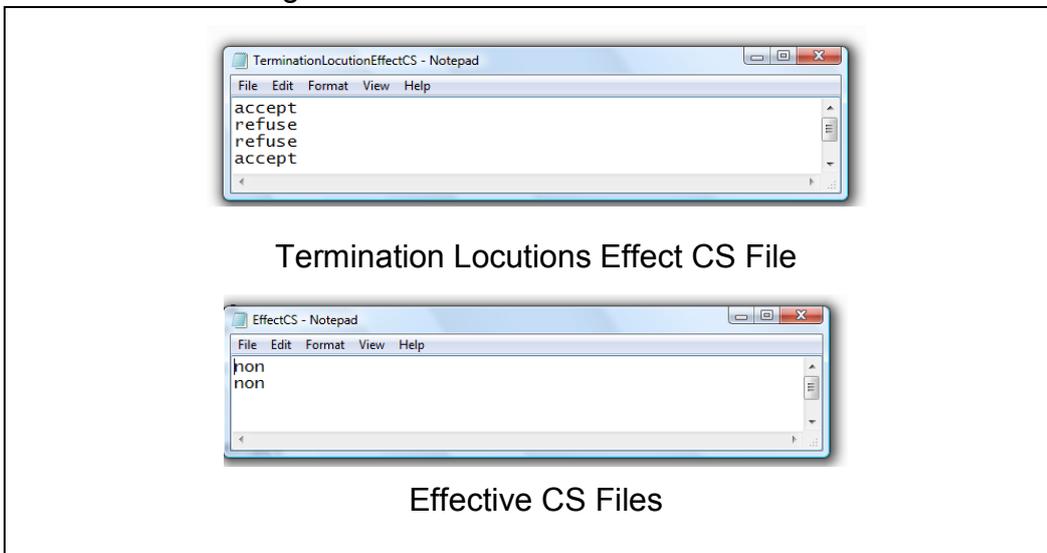


Figure A.23: Termination Locutions Effect CS and Effective CS Files

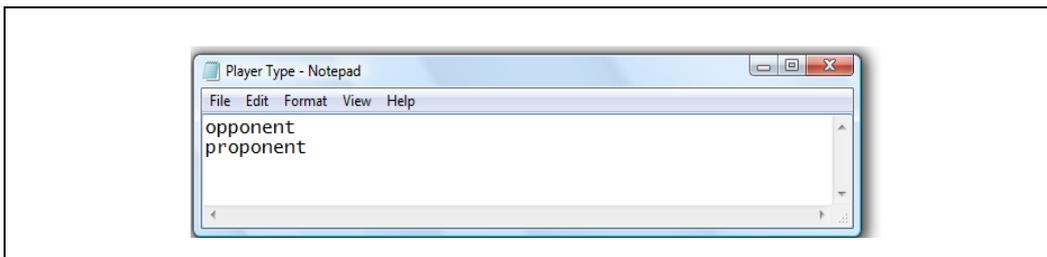


Figure A.24: Player Types File

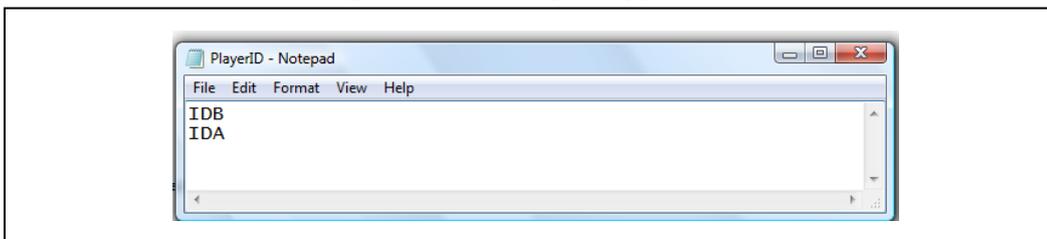


Figure A.25: Player Ids File

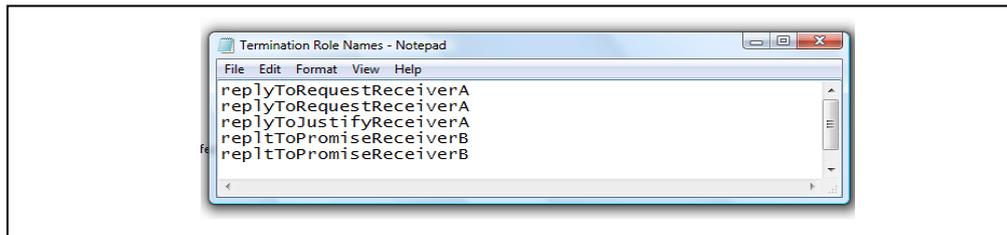


Figure A.26: Termination Role Names File

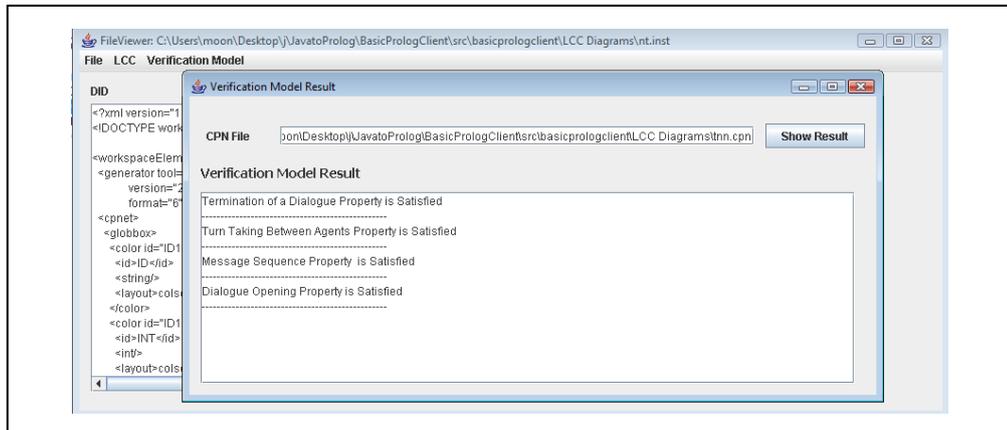


Figure A.27: The Verification Result of the Five Basic Properties

Step Four: Applying the Verification Process

The verification of the negotiation dialogue LCC protocol CPN Model (verifying of the five properties: Dialogue opening property, Termination of a dialogue property, Turn taking between agents property, Message sequencing property and Recursive message property) was done using the steps explained in chapter 7 and the results obtained were corresponding to the expected behaviour of the system (Figure A.27 shows the verification result of the five basic properties).

Step Five: Adding and Verification of New Property

Paper [Sadri et. al., 2001] explains two properties:

- (1) Successful request dialogue property: a negotiation dialogue between agents A and B is consider to be a successful if (see Figure A.28):
 - a. Agent B accepts a request of agent A ;
 - b. Agent A accepts a promise of agent B ;

- c. Agent *B* accepts a promise of agent *A*.
- (2) C-Successful request dialogue property: a negotiation dialogue between agents *A* and *B* is consider to be a c-successful if (see Figure A.29):
- a. Agent *A* accepts a promise of agent *B* and commits to give *R'* resource in exchange for *R''*;
 - b. Agent *B* accepts a promise of agent *A* and commits to give *R''* resource in exchange for *R'*.

The CPN model generated by the verification tool for the negotiation dialogue was not able to verify these two properties. It needed manual translations of the textual explanation of these properties to SML functions in the CPN model. These translations had to be done manually by creating new pages in the CPM model and then writing the SML functions in the new page. The following two subsections explain the SML functions of successful and c-successful dialogue properties.

Successful Request Dialogue Property SML Representation

Figure A.30 shows the algorithm of the CPN SML specification of this property:

- (1) Line 1: Read the state space graph Termination nodes information from the Property6 text file and save this information in *TNodes* variable.
- (2) Line 2: Call function *CheckProperty6*.
- (3) Line 3: Function inputs are *TNodes*.
- (4) Line 4: Extract the *message* information from *TNodes* (*message* represents termination message).
- (5) Lines 5: Check that the termination message in the state space is equal to the "accept" where:
 - a. *SuccessfulRequestChecking* function is used to compare the termination message in the state space with "accept" ;

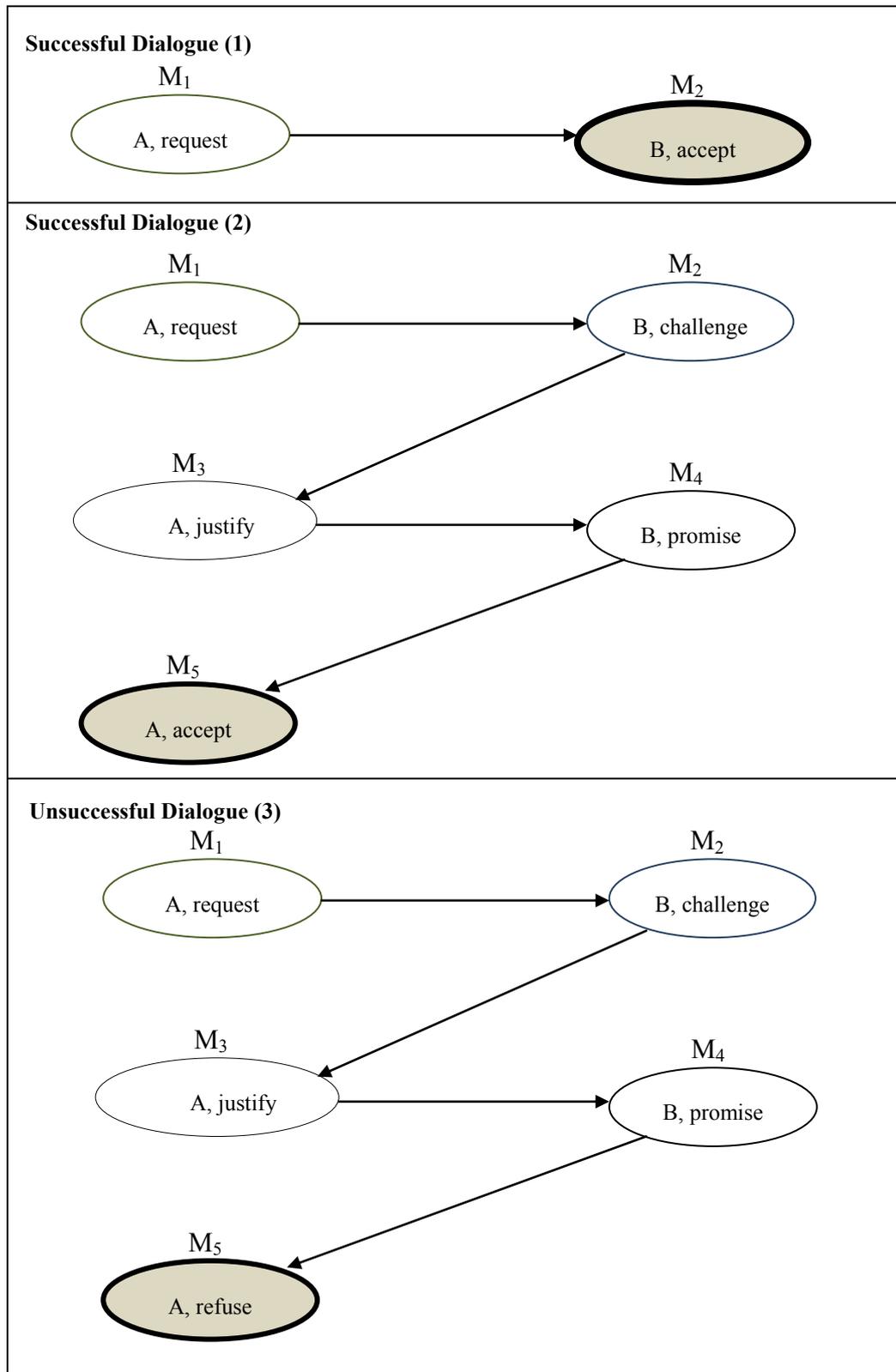


Figure 2.8: Successful and Unsuccessful Dialogue Examples

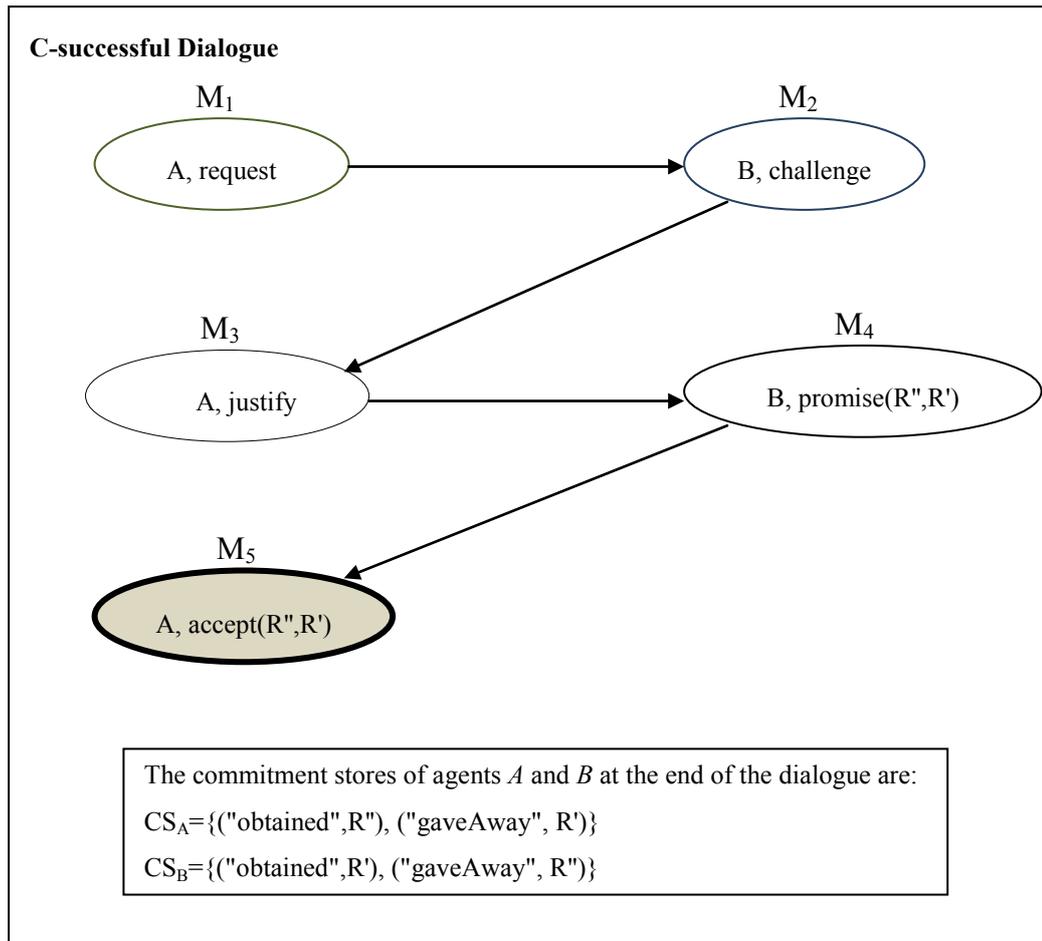


Figure 2.9: C-successful Dialogue Example

```

26. Read&Save  TNodes = state space termination nodes information
27. Call    CheckProperty6
28. Input  (TNodes)
29. Extract (message)
30. val mResult= SuccessfulRequestChecking(message)
31.   if (mResult >= 0) then
32.     "Property 6(Successful request dialogue) is Satisfied"
33.   else
34.     "Property 6(Successful request dialogue) is not Satisfied"
35. End CheckProperty6
36. Create&Save Property6 result file
  
```

Figure A.30: Property 6 (Successful Dialogue) as an Standard ML Function

- b. *mResult* represents the *SuccessfulRequestChecking* function result. It is considered true if the termination message in the state space is equal to the "accept".
- (6) Lines 6 to 9: Check the result of the comparison. A positive (negative) result indicates that Property 6 is satisfied (unsatisfied).
- (7) Line 11: Create Property6 result file and write the result of *CheckProperty6* in this file.

C-successful Request Dialogue Property SML Representation

Figure A.31 shows the algorithm of the CPN SML specification of this property:

- (1) Line 1: Read the state space graph Termination nodes information from the Property6 text file and save this information in *TNodes* variable.
- (2) Line 2: Call function *CheckProperty7*.
- (3) Line 3: Function inputs are *TNodes*.
- (4) Line 4: Extract the needed information from *TNodes* where:
 - j) *message* represents termination message;
 - k) *sender* represents termination message sender ID;
 - l) *receiver* represents termination message receiver ID;
 - m) *sCS* represents sender commitment store;
 - n) *rCS* represents receiver commitment store;
- (5) Lines 5: Check that the termination message in the state space is equal to the "accept" where:
 - a. *SuccessfulRequestChecking* function is used to compare the termination message in the state space with "accept" ;

```

1. Read&Save  TNodes = state space termination nodes information
2. Call   CheckProperty7
3. Input  (TNodes)
4. Extract (message, sender, receiver, sCS, rCS)
5. val mResult= SuccessfulRequestChecking(message)
6. val csContant = checkTheContantofCS(message, sCS,rCS)
7.   if (mResult >= 0) andalso (csContant= true) then
8.     "Property 7(C-successful request dialogue) is Satisfied"
9.   else
10.    "Property 7(C-successful request dialogue) is not Satisfied"
11. End CheckProperty7
12. Create&Save Property7 result file

```

Figure A.31: Property 7 (C-successful Dialogue) as an Standard ML Function

- b. mResult represents the *SuccessfulRequestChecking* function result. It is considered true if the termination message in the state space is equal to the "accept".
- (6) Lines 6: Check that the content of the CS in the termination message of the sender agent in the state space have ("obtained",R") and ("gaveAway", R') items. This line also checks the content of the CS in the termination message of the receiver agent in the state space have ("obtained",R') and ("gaveAway", R") items where:
- a. *checkTheContantofCS* function is used to compare the content of the CSs;
 - b. csContant represents the *checkTheContantofCS* function result.
- (7) Lines 7 to 10: Check the result of the comparison. A positive (negative) result indicates that Property 7 is satisfied (unsatisfied).
- (8) Line12: Create Property7 result file and write the result of *CheckProperty7* in this file.

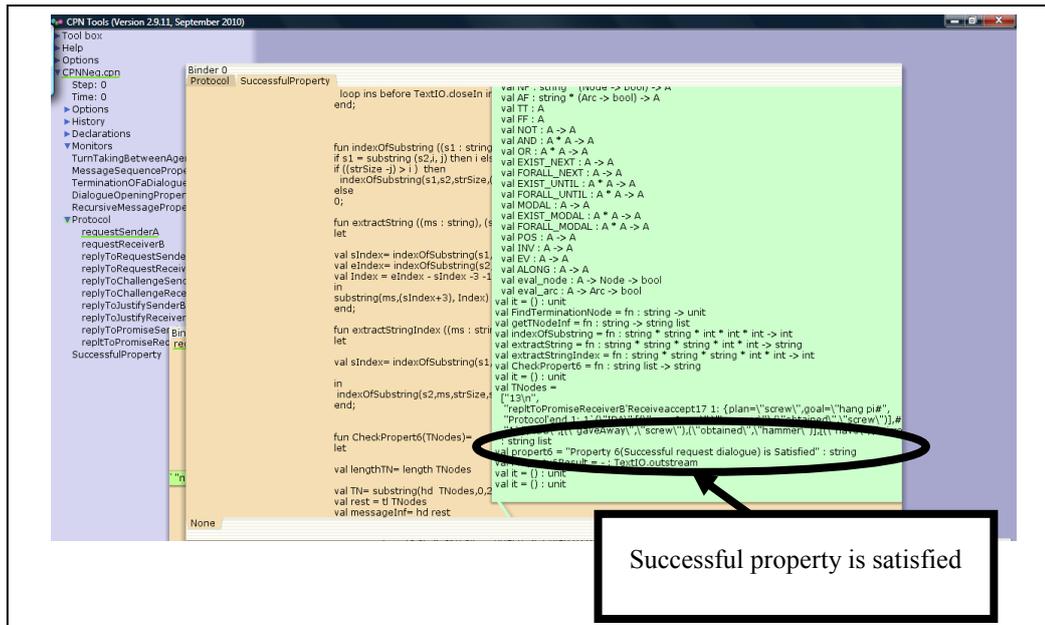


Figure A.32: Property 6 (successful Dialogue) Verification Result

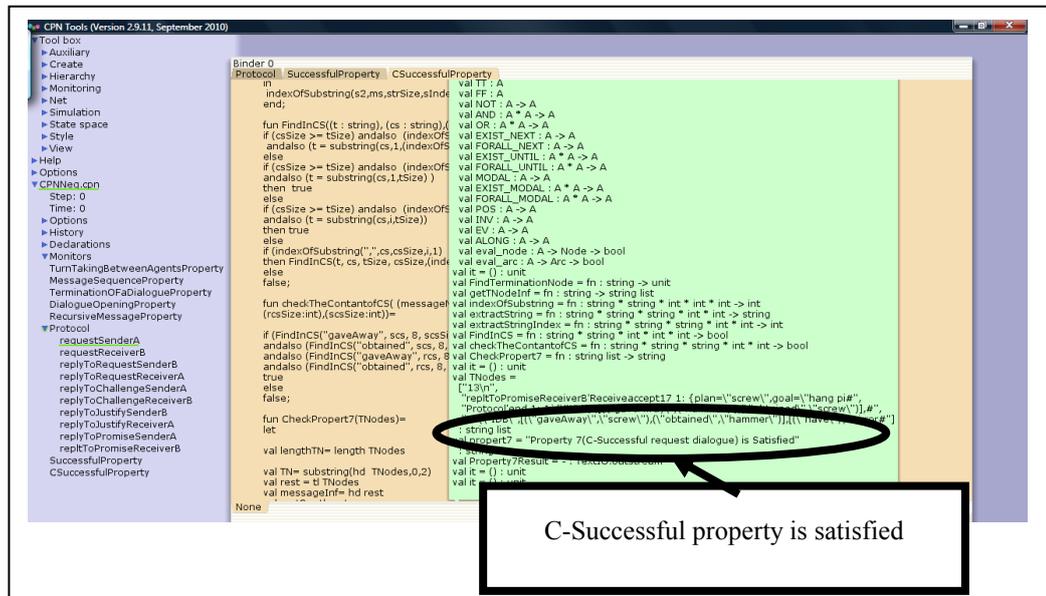


Figure A.33: Property 7 (C-successful Dialogue) Verification Result

Applying the Verification Process

After finishing manual translations of the textual explanation of these properties to SML functions in the CPN model, the verification of the negotiation dialogue LCC protocol CPN Model (verifying of the successful and c-successful properties) was done using the steps explained in chapter 7 and the results obtained were corresponding to the expected behaviour of the system (Figures A.32 and A.33 show the verification result of the these two properties).

Appendix B

N-agent Dialogue

To handle N-agent dialogue games, we extended DID diagram. This appendix presents the formal definition of DID for N-agent in Section B.1. An example of the persuasion dialogue between N-agent is presented in Section B.2. A description of LCC-Argument protocol general N-agent design patterns is presented in Section B.3.

B.1 DID for N-agent Formal Definition

In this section we extend the formal definition of DID for two agents to handle N-agent. Readers not interested in such details are encouraged to skip ahead to section "DID for a persuasion dialogue between N-agent" for an example of the DID or skip ahead to section B.3 for the general N-agent design patterns.

Definition 14: N-agent Players

A multi-agent system consists of a finite set of players (agents).

Players = $\{A_1, A_2, \dots, A_n\}$,

Where,

- $A_i \in \text{Players}$, where $i=1,2,3, \dots, n$
- Each player A_i has its own commitment store set $CS_i \subseteq \wp(\text{Args}(L))$, which contains a set of propositions to which the player is committed in the discussion.
- Each player A_i has its own knowledge base or beliefs set $KB_i \subseteq \wp(\text{Args}(L))$, which represents the propositions on which the agent believes.

Definition 15: N-agent Act Type

'ActType' is a function which determines the type of 'Act'.

$$\mathbf{ActType}: \text{Acts} \rightarrow \wp(\text{Types})$$

Where,

- Types = { RecursiveStarting, Intermediate, RecursiveTermination, Divided },
- RecursiveStarting: this type can be used to open a dialogue,
- Intermediate: this type can be used to remain in the dialogue,
- RecursiveTermination: this type can be used to terminate the dialogue,
- Divided: this type can be used to divide agents into groups and then to change the multi agent dialogue to two agents dialogue.

Definition 16: Recursive-conditions

'ReC' is a function which specifies the move recursive-conditions according to the dialogue protocol. It takes as input parameters an act and the recursive arguments and returns a Boolean and new recursive arguments.

$$\mathbf{ReC}: \text{Acts} \times \wp(\text{args(L)}) \rightarrow \text{Boolean}$$

Definition 17: Divided conditions

'DC' is a function which specifies the agent divided conditions according to the dialogue protocol. It takes as input parameters an act, players, the commitment store of all players and the knowledge based of all players and returns a Boolean.

$$\mathbf{DC}: \text{Acts} \times \wp(\text{Players}) \times (\text{args(L)})^n \times (\text{args(L)})^n \rightarrow \text{Boolean}$$

Definition 18: Next Player in N-agent dialogue

'NextPlayer' is a function which determines the next players to move at specific moment of a dialogue.

NextPlayer: Move $\rightarrow \wp(\text{Players})$

Definition 19: N-agent Dialogue Move

In the N-agent dialogue, there are three types of move:

(1) One sender and more than one agent will take the next turn (N-receiver):

A move $M_t \in \text{Moves}$, $t \geq 1$, is defined as:

$M_t = (\text{player}_t, \text{act}_t, \text{SetM}_{t-1}, \text{setPlayer}_j, \text{sender}_t, \text{rSetRole}_t)$,

Where,

- $\text{player}_t \in \text{Players}$ represents the player of the move,
- $\text{player}_t \notin \text{setPlayer}_j$
- $\text{act}_t \in \text{Acts}$ represents the speech act performed in the move,
- $\text{SetM}_{t-1} \in \wp(\text{Moves}) \cup \{\text{null}\}$ represents the previous moves (M_t is a reply to SetM_{t-1}),
- $\text{setPlayer}_j \in \wp(\text{Players})$ represents the next players in the dialogue,
- $\text{sender}_t \in \text{Roles}$ represents the role identifier of player (sender agent),
- $\text{rSetRole}_e \in \wp(\text{Roles})$ represents the role identifiers of the setPlayer_j (receiver agents),

(2) One sender agent and one receiver agent:

A move $M_t \in \text{Moves}$, $t \geq 1$, is defined as:

$M_t = (\text{player}_t, \text{act}_t, \text{SetM}_{t-1}, \text{player}_j, \text{sRole}_t, \text{rRole}_{\text{player}})$,

Where,

- $\text{player}_t \in \text{Players}$ represents the player of the move,

- $act_t \in Acts$ represents the speech act performed in the move,
- $player_t \neq player_j$,
- $player_j \in Players$ represents the next player in the dialogue,
- $SetM_{t-1} \in \wp(Moves) \cup \{null\}$ represents the previous moves (M_t is a reply to $SetM_{t-1}$),
- $sRole_e \in Roles$ represents the role identifiers of the $player_t$ (sender agent),
- $rRole_{player} \in Roles$ represents the role identifier of the $player_j$ (receiver agent).

(3) More than one sender (N-sender) and one receiver agent:

A move $M_t \in Moves$, $t \geq 1$, is defined as:

$$M_t = (setPlayer_t, act_t, SetM_{t-1}, player_j, sSetRole_t, rRole_t),$$

Where,

- $setPlayer_j \in \wp(Players)$ represents the players of the move,
- $act_t \in Acts$ represents the speech act performed in the move,
- $SetM_{t-1} \in \wp(Moves) \cup \{null\}$ represents the previous moves (M_t is a reply to $SetM_{t-1}$),
- $player_t \in Players$ represents the next player of the move,
- $player_t \notin setPlayer_j$
- $sSetRole_e \in \wp(Roles)$ represents the role identifiers of the $player_t$ sender agents,
- $rRole_t \in Roles$ represents the role identifier of the $player_j$ (receiver agent).

Definition 20: Legal move for N-agent

'legalMoveNAgent' is a function which specifies the legal moves at a particular moment in the N-agent dialogue. It takes the dialogue history at a particular moment and the commitment store of all players:

$$\text{LegalMovesNAgent: MoveSeq} \times (\wp(\text{args(L)}) \times \wp(\text{args(L)})^n \rightarrow \wp(\text{Moves})$$

Rule 4: (Start N-agent Dialogue)

This rule says that a N-agent dialogue always starts with a *RecursiveStarting* act by proposal agent:

$$\text{LegalMovesNAgent}([], CS_1, CS_2, \dots, CS_n) = \{ M_1 \}$$

Where,

- $M_1 = (\text{proposal}, \text{act}_1, \text{null}, \text{setPlayer}_j, \text{sRole}_{\text{proposal}}, \text{rSetRole}_1)$,
- $\text{proposal} \notin \text{setPlayer}_j$
- $\text{ActType}(\text{act}_1) = \{\text{RecursiveStarting}\}$,
- $\text{PreC}(\text{act}_1, \text{KB}_{\text{proposal}}, \text{CS}_{\text{proposal}}) = \text{true}$, where $\text{KB}_{\text{proposal}}$ represents proposal agent's knowledge base and $\text{CS}_{\text{proposal}}$ represents proposal agent's commitment store.
- $\text{PostC}(\text{act}_1, \text{KB}_j, \text{CS}_j) = \text{true}$ (for each $\text{player} \in \text{setPlayer}_j$), where KB_j represents agent knowledge base and CS_j represents agent commitment store.

Rule 5: (Reply to a Proposal Agent's Move)

This rule says that more than one move will reply to a proposal agents' move:

$$\text{LegalMovesNAgent}([M_1, M_2, \dots, M_t], CS_1, CS_2, \dots, CS_n) = \text{SetM}_{t+1}$$

if

- $M_t = (\text{proposal}, \text{act}_t, \text{SetM}_{t-1}, \text{setPlayer}_j, \text{sRole}_{\text{proposal}}, \text{rSetRole}_t)$,

- $\text{proposal} \notin \text{setPlayer}_j$
- $\text{PreC}(\text{act}_t, \text{KB}_{\text{proposal}}, \text{CS}_{\text{proposal}}) = \text{true}$, where $\text{KB}_{\text{proposal}}$ represents proposal agent's knowledge base and $\text{CS}_{\text{proposal}}$ represents proposal agent's commitment store.
- $\text{PostC}(\text{act}_t, \text{KB}_j, \text{CS}_j) = \text{true}$ (for each $\text{player} \in \text{setPlayer}_j$), where KB_j represents agent knowledge base and CS_j represents agent commitment store.
- $M_{t+1} = (\text{setPlayer}_j, \text{act}_{t+1}, M_t, \text{proposal}, \text{sSetRole}_{t+1}, \text{rRole}_{\text{proposal}})$,
- $M_{t+1} \in \text{SetM}_{t+1}$
- $\text{ActType}(\text{act}_{t+1}) = \{\text{Intermediate}\}$,
- $\text{act}_{t+1} \in \text{Replies}(\text{act}_t)$ (M_{t+1} replies to M_t),
- $\text{PreC}(\text{act}_{t+1}, \text{KB}_j, \text{CS}_j) = \text{true}$ (for each $\text{player} \in \text{setPlayer}_j$), where KB_j represents agent knowledge base and CS_j represents agent commitment store.
- $\text{PostC}(\text{act}_{t+1}, \text{KB}_{\text{proposal}}, \text{CS}_{\text{proposal}}) = \text{true}$, where $\text{KB}_{\text{proposal}}$ represents proposal agent's knowledge base and $\text{CS}_{\text{proposal}}$ represents proposal agent's commitment store.

With this rule we are specifying also the turn-taking restriction. The sender of move M_t is the receiver of all the move from the SetM_{t+1} and the receiver of move M_t is the sender of all the move from the SetM_{t+1} .

Rule 6: (N-agent Dialogue Termination)

This rule says that a N-agent dialogue always terminates with a RecursiveTermination act by the proposal agent:

$$\text{LegalMovesNAgent} ([M_1, M_2, \dots, M_t], \text{CS}_1, \text{CS}_2, \dots, \text{CS}_n) = \emptyset$$

if

- $M_t = (\text{proposal}, \text{act}_t, M_{t-1}, \text{null}, \text{sRole}_{\text{proposal}}, \text{rSetRole}_t)$,
- $\text{ActType}(\text{act}_t) = \{\text{RecursiveTermination}\}$,
- $\text{PreC}(\text{act}_t, \text{KB}_{\text{proposal}}, \text{CS}_{\text{proposal}}) = \text{true}$, where $\text{KB}_{\text{proposal}}$ represents proposal agent's knowledge base and $\text{CS}_{\text{proposal}}$ represents proposal agent's commitment store.
- $\text{PostC}(\text{act}_t, \text{KB}_j, \text{CS}_j) = \text{true}$ (for each $\text{player} \in \text{setPlayer}_j$, setPlayer_j represents the previous players and $\text{proposal} \notin \text{setPlayer}_j$), where KB_j represents agent knowledge base and CS_j represents agent commitment store.

Rule 7: (Divide Agents in to Groups)

This rule says that proposal agent is responsible of dividing agents into groups composed of two agents and sending *Divided act* to all other agents to inform them about the groups. Once agents are divided in the group, dialogues take place between two agents (the next move is a move in dialogue between two agents):

$\text{LegalMovesNAgent} ([M_1, M_2, \dots, M_t], \text{CS}_1, \text{CS}_2, \dots, \text{CS}_n) = \{M_{t+1}\}$

- $M_t = (\text{proposal}, \text{act}_t, M_{t-1}, \text{setPlayer}_j, \text{sRole}_{\text{proposal}}, \text{rSetRole}_t)$,
- $\text{ActTypes}(\text{act}_t) = \{\text{Divided}\}$,
- $\text{proposal} \notin \text{setPlayer}_j$,
- M_{t+1} is a move in dialogue between two agents (Note that M_{t+1} must be a legal move in the two agents dialogue. See *Definition 14*),
- $\text{PreC}(\text{act}_t, \text{KB}_{\text{proposal}}, \text{CS}_{\text{proposal}}) = \text{true}$, where $\text{KB}_{\text{proposal}}$ represents proposal agent's knowledge base and $\text{CS}_{\text{proposal}}$ represents proposal agent's commitment store,

- $\text{PostC}(\text{act}_i, \text{KB}_j, \text{CS}_j) = \text{true}$ (for each $\text{player} \in \text{setPlayer}_j$), where KB_j represents agent knowledge base and CS_j represents agent commitment store,
- $\text{DC}(\text{act}_i, \text{Players}, \text{SetKB}, \text{SetCS}) = \text{true}$, where
 - each $\text{player}_i \in \text{Players}$ has $\text{KB}_i \in \text{SetKB}$ and has $\text{CS}_i \in \text{SetCS}$
 - KB_i represents agent knowledge base
 - and CS_i represents agent commitment store
 - $i = 1, 2, \dots, n$

Rule 8: (Return Back to Dialogue Between N-agent)

This rule says that :

$\text{LegalMovesNAgent} ([M_1, M_2, \dots, M_{t+1}], \text{CS}_1, \text{CS}_2, \dots, \text{CS}_n) = \{M_{t+2}\}$

If

- M_{t+1} is a move in dialogue between two agents
- $M_{t+1} = (\text{player}_i, \text{act}_{t+1}, M_{t-1}, \text{null}, \text{sRole}_{t+1}, \text{rRole}_{t+1})$,
- $\text{ActType}(\text{act}_{t+1}) = \{\text{Termination}\}$,
- $\text{PreC}(\text{act}_{t+1}, \text{KB}_i, \text{CS}_i) = \text{true}$, where KB_i represents agent I 's knowledge base and CS_i represents agent I 's commitment store.
- $\text{PostC}(\text{act}_{t+1}, \text{KB}_k, \text{CS}_k) = \text{true}$, where KB_k represents agent K 's knowledge base and CS_k represents agent K 's commitment store.
- $M_{t+2} = (\text{proposal}, \text{act}_{t+2}, M_{t+1}, \text{setPlayer}_j, \text{sRole}_{\text{proposal}}, \text{rSetRole}_{t+2})$,
- M_{t+2} is a move in dialogue between N-agent
- $\text{ActTypes}(\text{act}_{t+2}) = \{\text{RecursiveStarting}\}$

- $\text{proposal} \notin \text{setPlayer}_j$
- $\text{PreC}(\text{act}_{t+2}, \text{KB}_{\text{proposal}}, \text{CS}_{\text{proposal}}) = \text{true}$, where $\text{KB}_{\text{proposal}}$ represents proposal agent's knowledge base and $\text{CS}_{\text{proposal}}$ represents proposal agent's commitment store.
- $\text{PostC}(\text{act}_{t+2}, \text{KB}_j, \text{CS}_j) = \text{true}$ (for each $\text{player} \in \text{setPlayer}_j$), where KB_j represents agent knowledge base and CS_j represents agent commitment store.

B.2 DID for N-agent Example

Figure B.1, which was adapted from [Ito and Shintani, 1997], illustrates an example of a persuasion dialogue between N-agent:

- The system will randomly select a proposal agent
- A proposal agent sends (broadcasting) a *proposal(Topic)* locution to all other agents.
- Each agent who receives the *proposal(Topic)* reports acceptance of the *proposal(Topic)* by sending an *accept(Topic)* locution or rejection of the *proposal(Topic)* by sending a *reject(Topic)* locution.
- If the agents reach an agreement (if Acceptance number \geq The number of supporter agents), the proposal sends *reachAgreement(Topic)* to all other agents.
- If the agents could not reach an agreement on the *proposal(Topic)*, the proposal divides agents into groups composed of two agents and sends *argueWith* locution to all other agents to inform them about the groups.

DID formal definition for a persuasion dialogue between N-agent

(1) **Players:** $\text{Players} = \{\text{Agent}_1, \text{Agent}_2, \dots, \text{Agent}_n\}$

Each player has its own KB and CS such that:

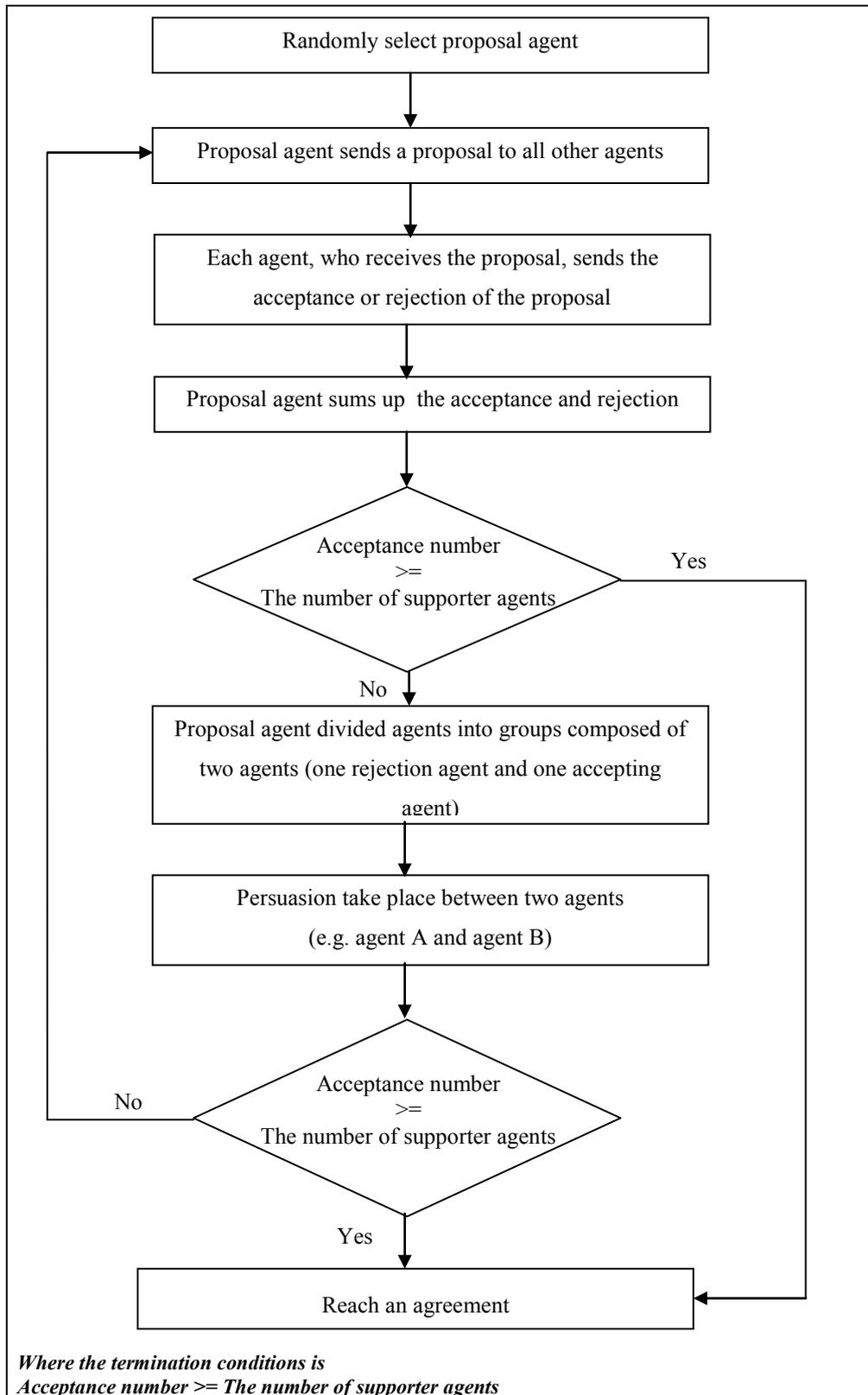


Figure B.1: Persuasion Dialogue Between N-agent

Bridging the Specification Protocol Gap in Argumentation

Agent₁ argumentation system AS_{Agent1} ($AS_{Agent1} = \{KB_{Agent1}, CS_{Agent1}\}$)

(2) There are five locutions (Acts):

Acts = {proposal(Topic), accept(Topic), reject(Topic), reachAgreement(Topic), argueWith(Topic, Agent_p, Agent_o)}

(3) ActType(Act):

Act	ActType (Act)
proposal	{RecursiveStarting}
accept	{ Intermediate }
reject	{Intermediate}
reachAgreement	{RecursiveTermination}
argueWith	{Divided}

(4) Replies(Act):

Act	Replies(Act)	Note
proposal(Topic)	{ accept(Topic), reject(Topic)}	
accept(Topic)	{ reachAgreement(Topic), argueWith(Topic)}	
reject(Topic)	{ reachAgreement(Topic), argueWith(Topic)}	
reachAgreement(Topic)	∅	
argueWith(Topic, Agent _p , Agent _o)	{claim(Topic)}	Replies(Act) for argueWith locution represents the Starting Locution icon in the DID for two agents (e.g. Replies(Act)= claim(Topic) which represents the Starting Locution icon in the persuasion dialogue between two agents in section 4.2.1). In other words, we need to connect <i>argueWith</i> with the Starting Locution icon in the DID for two agents.

(5) PreC(Act,KB,CS):

Lets Player = Proposal

Act	PreC(Act,KB,CS)	Note
proposal(Topic)	addTopicToCS(Topic,CS _{Proposal}) = true	
accept(Topic)	findTopicInKB(Topic, KB _{ID}) = true and notFindTopicInCS (Topic,CS _{ID}) = true and notFindOppTopicInCS (not(Topic),CS _{ID}) = true and addTopicToCS(Topic,CS _{ID}) = true	
reject(Topic)	notFindTopicInKB(Topic,KB _{Proposal}) = true and notFindTopicInCS(Topic,CS _{Proposal}) = true	
reachAgreement(Topic)	greaterThanOrEequal(NAccepting, NSupporters) = true	<i>greaterThanOrEequal</i> function returns true if the number of accepting agents <i>NAccepting</i> is greater than or equal to the number of supporter agents <i>NSupporters</i> . (<i>NAccepting</i> >= <i>NSupporters</i>)
argueWith(Topic, Agent _p ,Agent _o)	lessThan(NAccepting,NSupporters) = true and isNotEmpty(RejectionList) = true and isNotEmpty(AcceptingList) = true	<ul style="list-style-type: none"> • <i>lessThan</i> function returns true if the number of accepting agents <i>NAccepting</i> is less than the number of supporter agents <i>NSupporters</i>. (<i>NAccepting</i> < <i>NSupporters</i>) • <i>isNotEmpty</i> function returns true if the list is not empty.

(6) PostC(Act,KB,CS):

 let Player(M_i)= Proposal

Act	PostC(Act,KB,CS)	Note
proposal(Topic)	true	
accept(Topic)	addToAcceptingList (AcceptingList, AccList ,ID) = true and increaseAccepting (NAccepting,NAcc) = true and addIDToList(AgentList, SendingList, ID) = true	<ul style="list-style-type: none"> • <i>addToAcceptingList</i> function always returns true and results in proposal agent adding the accepting agent <i>ID</i> to the <i>AcceptingList</i> (<i>AccList</i> =<i>AcceptingList</i> \cup {<i>ID</i>}). • <i>increaseAccepting</i> function increases the number of accepting agents (<i>NAcc</i> = <i>NAccepting</i> +1) • <i>addIDToList</i> function always returns true and results in proposal agent adding the agent <i>ID</i> to the <i>SendingList</i>
reject(Topic)	addToRejectingList (RejectingList,RejList,ID) = true and increaseRejecting (NRejecting,NRej) = true and addIDToList(AgentList, SendingList, ID) = true	<ul style="list-style-type: none"> • <i>addToRejectingList</i> function always returns true and results in proposal agent adding the rejecting agent <i>ID</i> to the <i>RejectingList</i> (<i>RejList</i>=<i>RejectingList</i> \cup {<i>ID</i>}). • <i>increaseRejecting</i> function increases the number of rejecting agents (<i>NRej</i> =<i>NRejecting</i>+1)
reachAgreement (Topic)	true	
argueWith(Topic, Agent _p ,Agent _o)	true	

(7) ReC(Act,KB,CS):

 let Player(M_i)= Proposal

Bridging the Specification Protocol Gap in Argumentation

Act	ReC(Act,KB,CS)	Note
proposal(Topic)	getAgentIDFromList (AgentList, otherAgents,ID) = true	<i>getAgentIDFromList</i> function gets agent <i>ID</i> from the <i>AgentsList</i> and puts the remaining agents in the <i>otherAgents</i> list (<i>OtherAgents</i> = <i>AgentsList</i> – { <i>ID</i> })
accept(Topic)	notEqual(AgentList, SendingList)	<i>notEqual</i> function compare the <i>AgentList</i> with the <i>Sending</i> and returns true if these two lists are equal
reject(Topic)	notEqual(AgentList, SendingList)	
reachAgreement(Topic)	getAgentIDFromList (AgentList, otherAgents)=true	
argueWith(Topic, Agent _p ,Agent _o)	creatOneAgentGroups (RejectingList,Re, AcceptingList,Ac, AgentGroup, AGroup,P,O) = true	<i>creatOneAgentGroups</i> function: (4) creates one agent group by getting one agent <i>O</i> from the <i>Rejectinglist</i> and one agent <i>P</i> from the <i>Acceptinglist</i> . (5) adds the new agents groups to <i>AGroup</i> list (<i>AGroup</i> = <i>AgentGroup</i> + {(<i>P</i> , <i>O</i>)}. (6) Saves the remained rejection agent in <i>Re</i> list and saves the remained accepting agents in <i>Ac</i> .

(8) LegalMovesNAgent(M_t , CS_{Agent1} , $CS_{Agent2}, \dots, CS_{AgentN}$)

From Figure B.2, we can see that:

- Dialogues open by making a *proposal* move
- In this dialogue, the argument terminates once one agent sends *reachAgreement*.
- Both *accept* and *reject* \in {Intermediate}. There are several moves to these moves. (there are arrows coming out from these moves).
- After *argueWith* \in {Divided}, the dialogue between two agents begins.

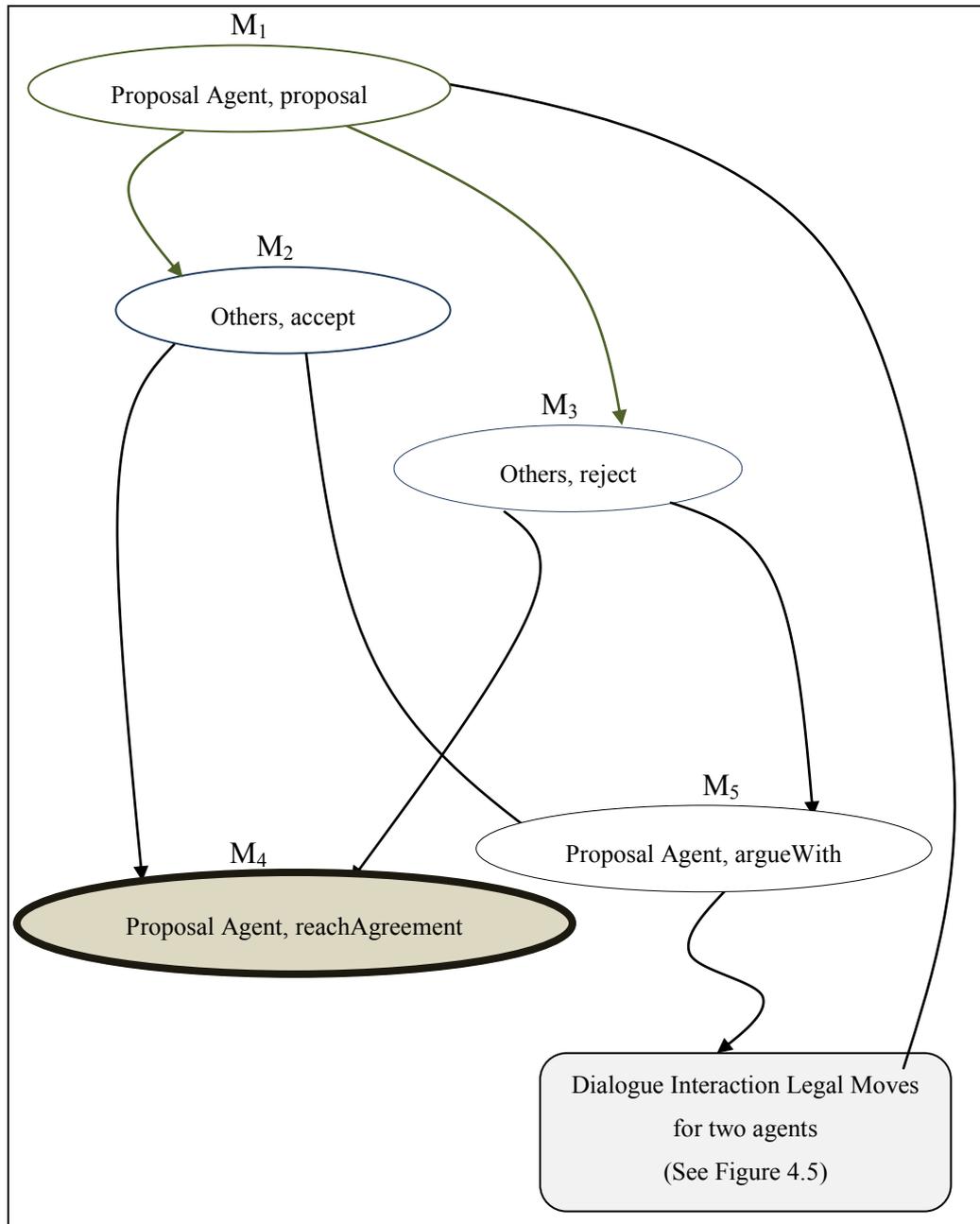


Figure B.2: The Persuasion Dialogue Between N-agent Legal Moves

- The turn-taking between participants switches after each move (the agents take it in turns to make moves):
 - if M_1 then Player = Proposal,
 - else NextPlayer = All other agents iff Player = Proposal
and NextPlayer = Proposal iff Player = All other agents

DID for a persuasion dialogue between N-agent

The DID of this example is shown in Figure B.3 (Note that pre-conditions and post-conditions for locutions are not shown in this figure since they are shown in Figures B.4(a), B.4(b), and B.4(c).) In Figure B.3, a dialogue always starts with a *proposal* and ends with a *reachAgreement* locution. *Proposal Agent* can open the discussion by sending a *proposal(Topic)* locution, if it is able to satisfy both the pre-condition and the recursive condition that are connected to the sender role of this locution: 1) *getAgentIDFromList(AgentList,otherAgents,ID)* that returns true if *AgentList* is not empty, gets *agent ID* from the *AgentsList* and puts the remaining agents in the *otherAgents* list; 2) *addTopicToCS(Topic,CS_{proposal})* that returns true if *Proposal Agent* is able to add *Topic* to its commitment store *CS_{proposal}* (if *Topic* is not already in the *CS_{proposal}*), which is always returned true. Then, turn-taking switches to *All other agents*. Each of them has to choose between two different possible reply locutions: *accept(Topic)* or *reject(Topic)*. Each agent will make its choice using the pre-conditions which appear in the rhombus shape. An agent sends *accept(Topic)*, if it is able to satisfy: 1) *findTopicInKB(Topic, KB_{ID})* that returns true if the agent is able to find *Topic* in its knowledge base *KB_{ID}*; 2) *notFindTopicInCS(Topic,CS_{ID})* that returns true if the agent is not able to find *Topic* in its commitment store *CS_{ID}*; 3) *notFindOppTopicInCS(not(Topic),CS_{ID})* that returns true if the agent is not able to find the opposite of *Topic* in its commitment store *CS_{ID}*; 4) *addTopicToCS(Topic,CS_{ID})* that returns true if the agent is able to add *Topic* to its commitment store *CS_{ID}* which always returns true. An agent sends *reject(Topic)*, if it is able to satisfy: 1) *notFindTopicInKB(Topic,KB_{ID})* that returns true if the agent is not able to find *Topic* in its knowledge base *KB_{ID}*; 2) *notFindTopicInCS(Topic,CS_{ID})* that returns true if the agent is not able to find *Topic* in its commitment store *CS_{ID}*.

After that, the turn switches to *Proposal Agent*, and so forth. The argument terminates when *Proposal Agent* sends *reachAgreement* locution to all other agents.

Note that in this example, each dialogue game between two agents has four input parameters: 1) *Topic* (which represents the main topic of the dialogue between N-agent); 2) *ID_{Proposal}* (which represents the proposal agent ID);

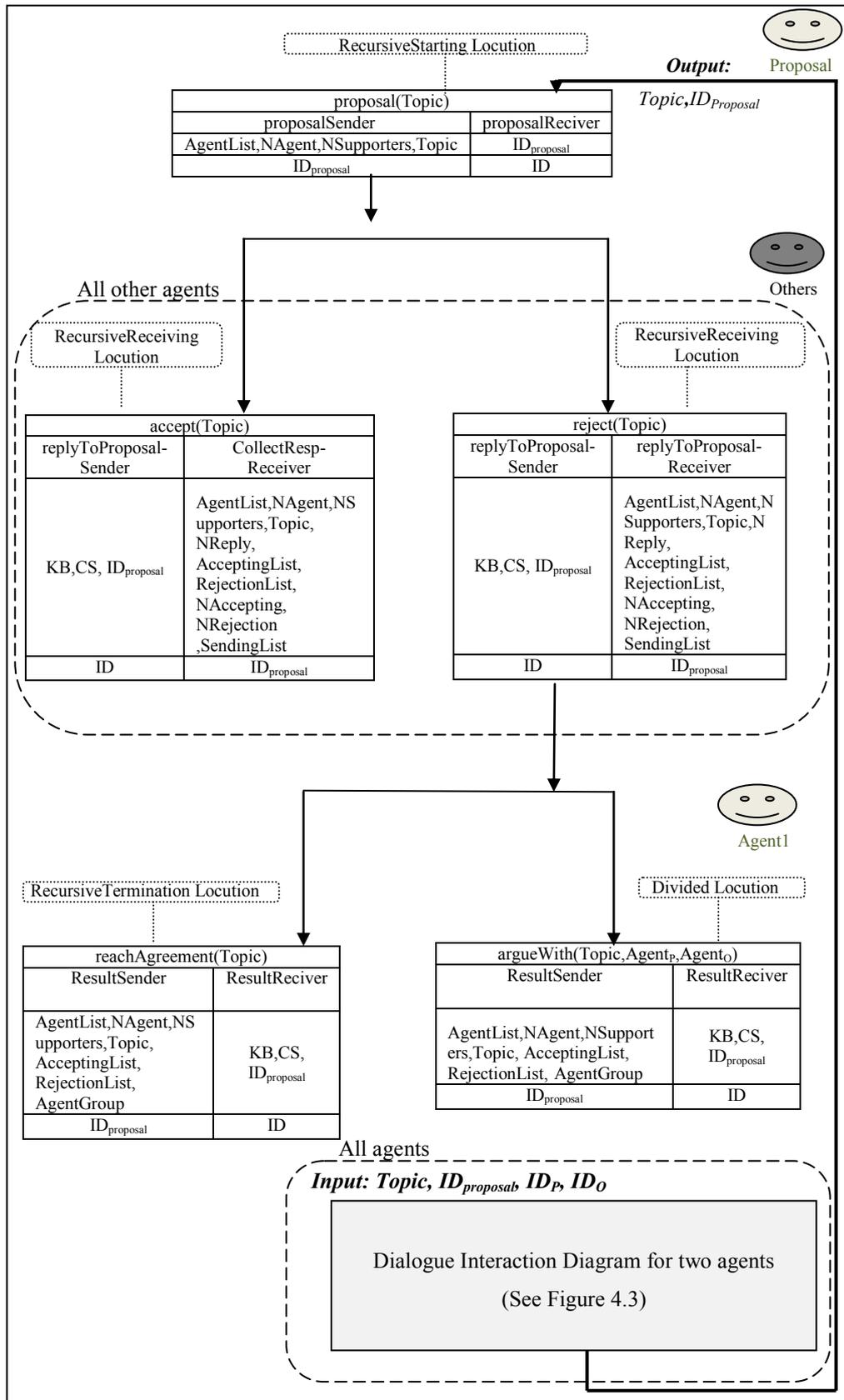


Figure B.3: Dialogue Interaction Diagram for N-agent (DIDN)

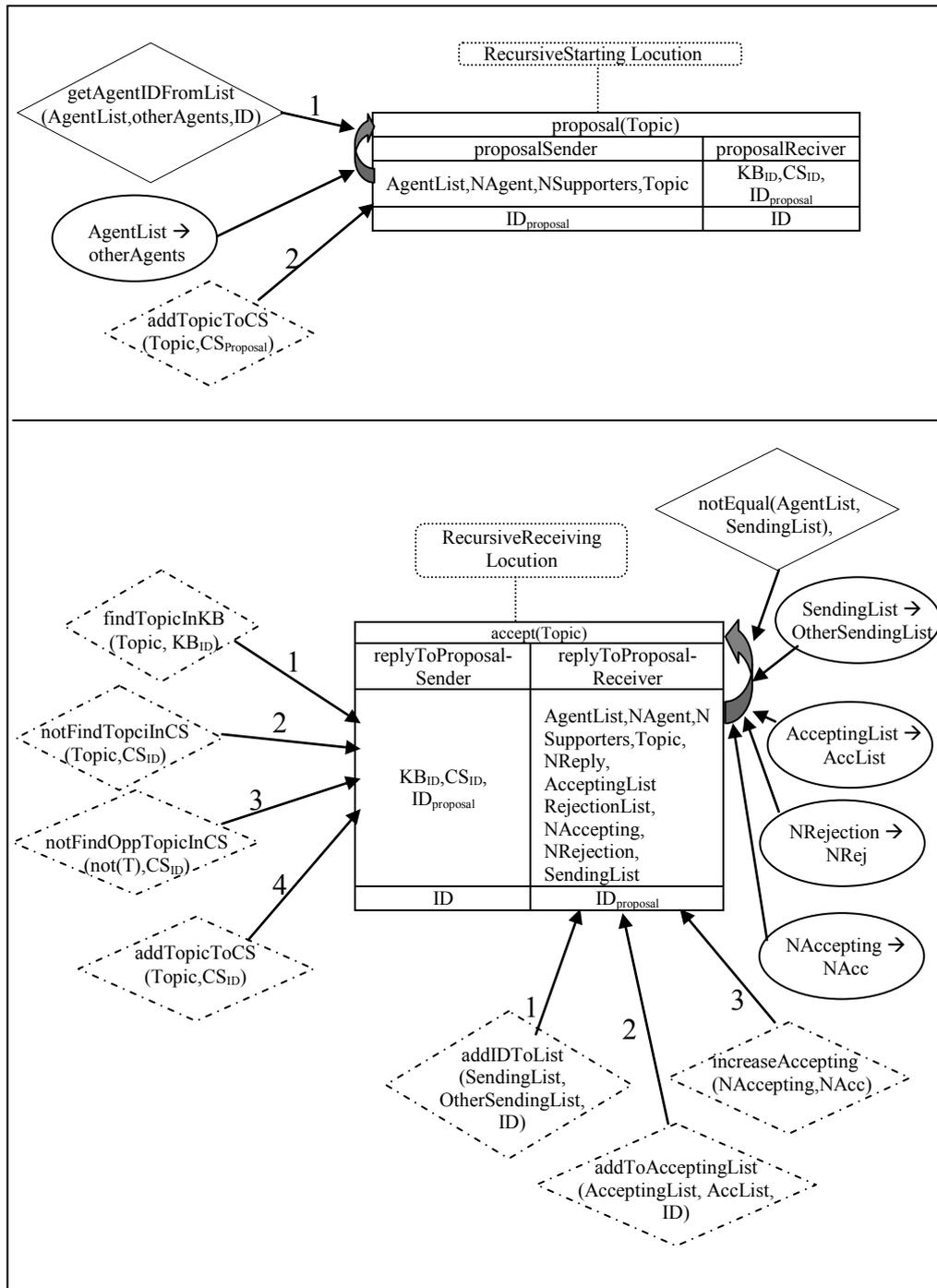


Figure B.4 (a): DIDN Locutions Pre-conditions and Post-conditions

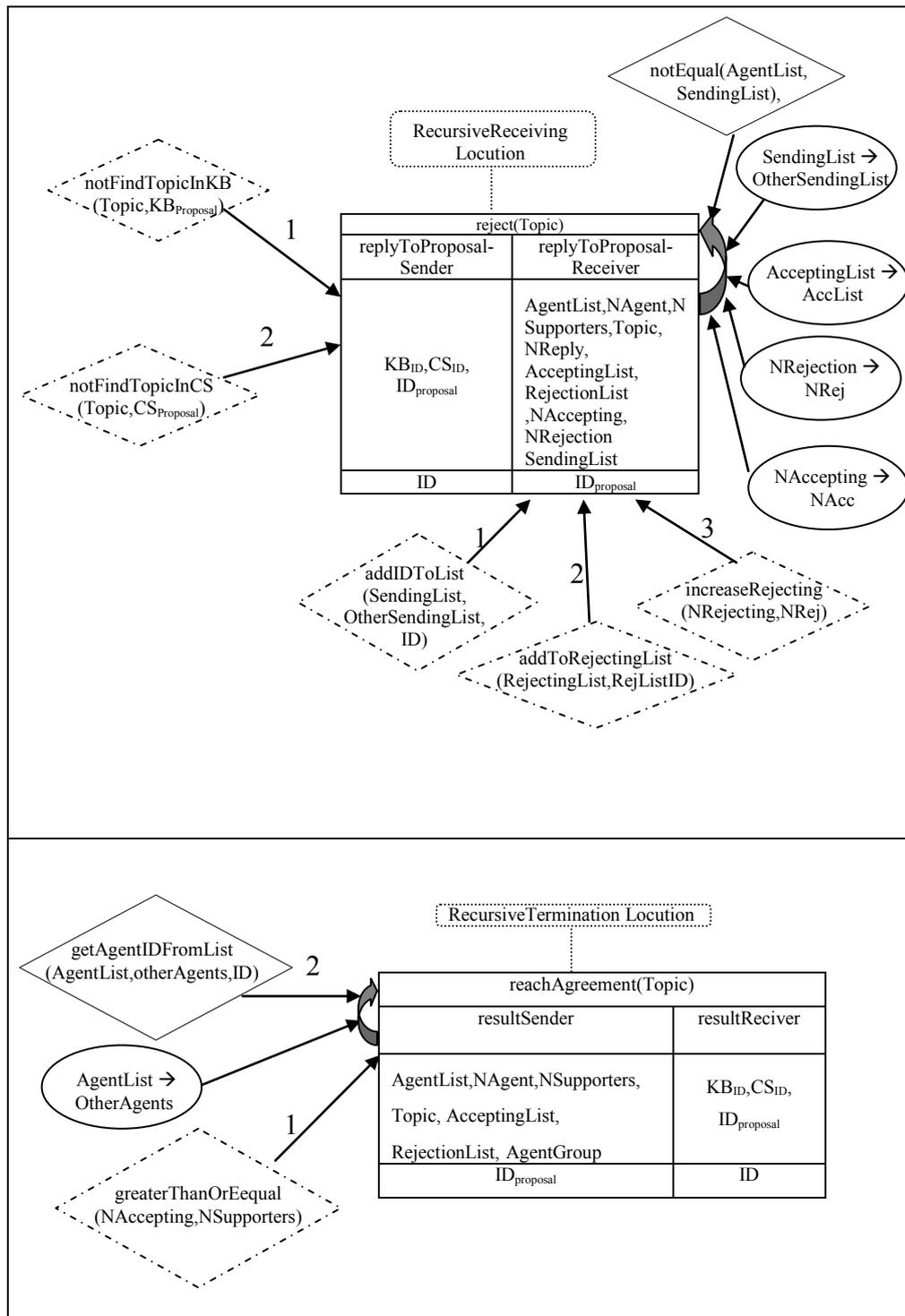


Figure B.4(b): DIDN Locutions Pre-conditions and Post-conditions

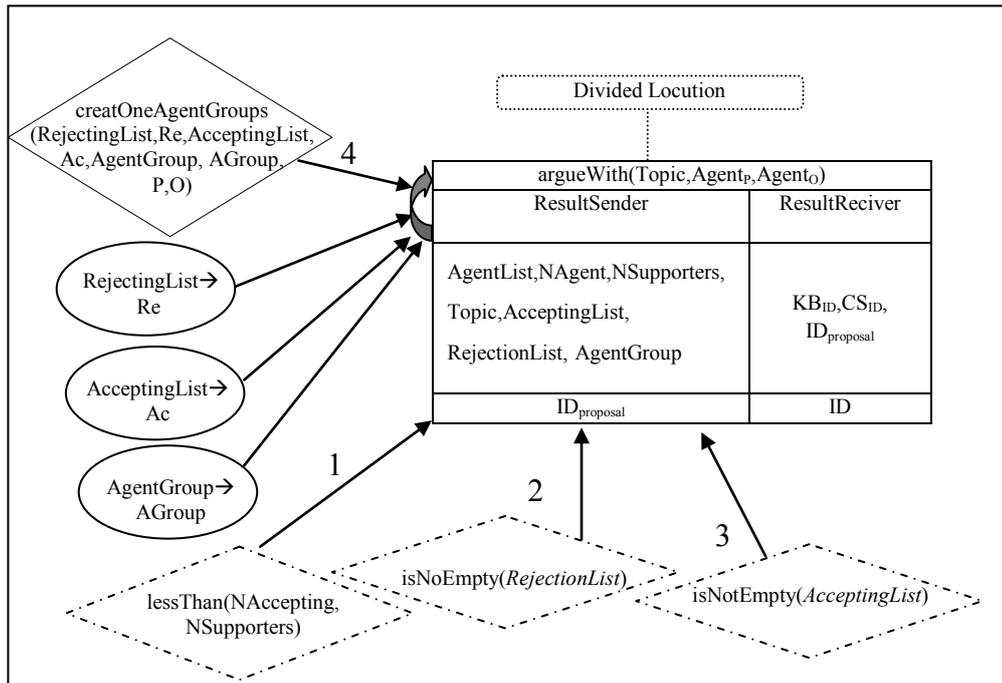


Figure B.4(c): DIDN Locutions Pre-conditions and Post-conditions

3) ID_P (which represents the first agent ID in the current group); 4) ID_O (which represents the second agent ID in the current group). Each of the dialogue games between two agents has two output parameters: 1) *Topic* (which represents the main topic of the dialogue between N-agent); 2) $ID_{Proposal}$ (which represents the sender agent ID).

The basic Scenario of Interaction Protocol of Persuasion Dialogue between N-agent

An example (see Figure 4.14) of the persuasion dialogue among seven agents is shown in Figure B.5 (note that the DID between two agents is not shown in this diagram). The goal of the dialogue is to persuade all agents that *A's car is safe*. In this example:

- (1) *A* opens a discussion by sending a *proposal* ("My car is safe") to all other agents (*B, C, D, E, F* and *G*).
- (2) Each agent checks with its argumentation system *AS* ($AS = \{KB, CS\}$) whether "*A's car is safe*" is acceptable:

- If an agent finds that "*A's car is safe*", it sends *accept("My car is safe")* to *A*,
- If an agent does not find "*A's car is safe*", it sends *reject("My car is safe")* to *A*,

In this example, *C* accepts the proposal and *B*, *C*, *D*, *E*, *F* and *G* reject the proposal.

(3) *A* sums up the acceptance and rejection locutions.

- If the acceptance number is equal to the number of agents (termination condition), the agents have reached an agreement and *A* sends a *reachAgreement("My car is safe")* locution to all other agents.
- If the number of rejections is equal or greater than one (*Divided condition*), *A* divides agents into groups of two under the condition that it cannot put two accepting agents or two rejection agents together in one group (note that if the number of agents is even, every agent has a partner. If the number of agents is odd, the last agent lacks a partner). Then, *A* sends an *argueWith* locution to all other agents to inform them about the groups.

In this example, group one consists of *A* and *B* and group two consists of *C* and *D* (note that *E*, *F* and *G* have rejected the proposal so we cannot put them together in one group.)

- (4) Within each group, dialogues take place between two agents. In this example, each group will use the DID between two agents given in Figure 4.3.
- (5) Each agent in the group sends either an *accept("My car is safe")* or *reject("My car is safe")* locution to *A*.
- (6) Then, *A* repeats step 3. The following are the new groups: group one consists of *A* and *E*, group two consists of *B* and *F*. Within each group, dialogues take place between two agents.

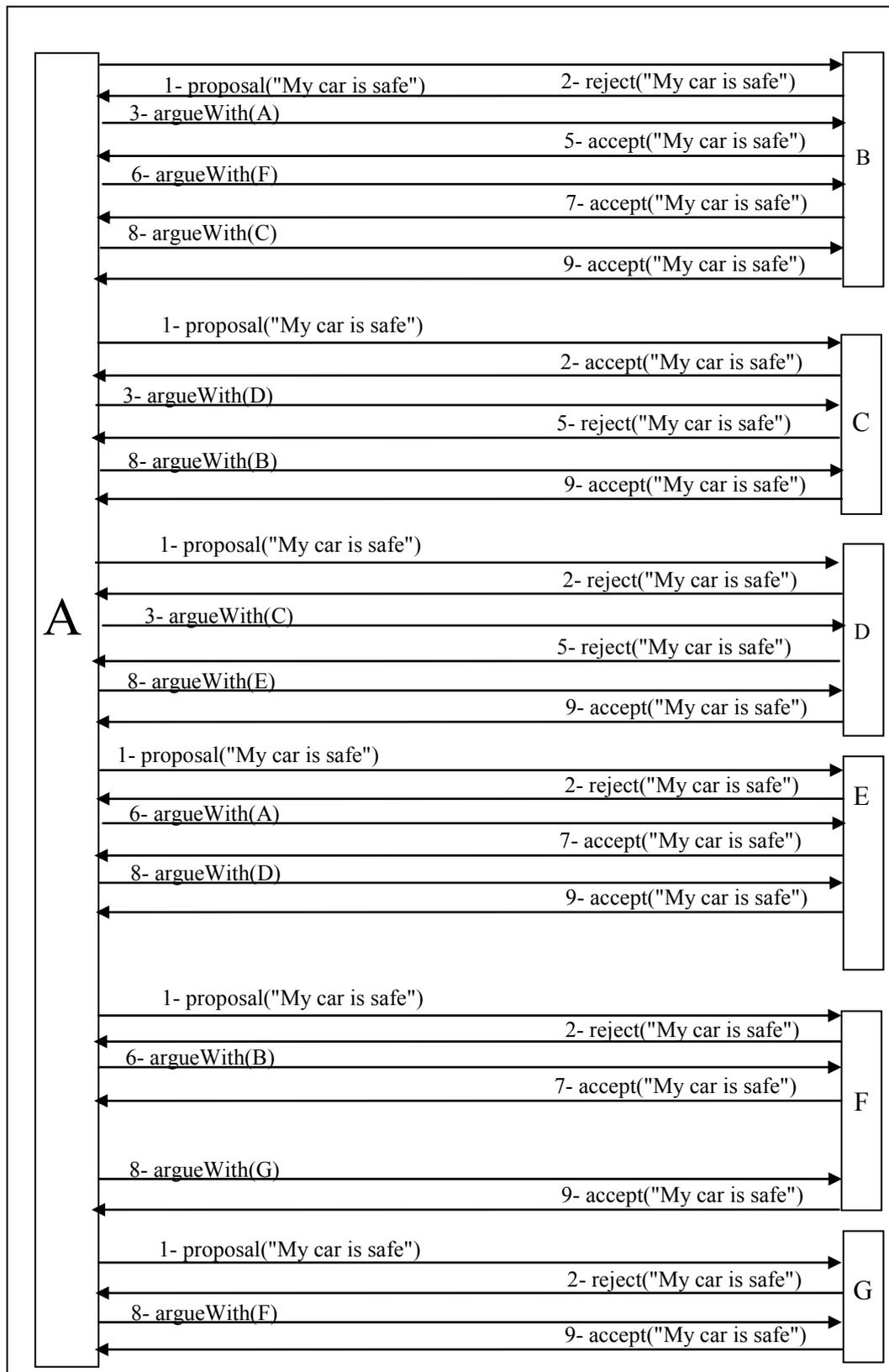


Figure B.5: The Complex Car Safety Example Among N-agent

- (7) Each agent in the group sends either an *accept("My car is safe")* or *reject("My car is safe")* locution to *A*.
- (8) Then, *A* repeats step 3. The following are the new groups: group one consists of *B* and *C*, group two consists of *E* and *D*, and group three consists of *F* and *G*. Within each group, dialogues take place between two agents.
- (9) Each agent in the group sends either an *accept("My car is safe")* or *reject("My car is safe")* locution to *A*. Finally, *A* sums up the acceptance and rejection locutions and finds that the acceptance number is equal to the number of agents, which means that the agents have reached an agreement. *A* sends *reachAgreement("My car is safe")* to all other agents.

B.3 General N-agent Patterns

As mentioned in chapter 4 and 5, we have focused on those involving more than two agents where synthesized LCC protocols specify broadcasting methods to divide agents into groups composed of two agents (with these two-agent dialogues then being specified using DID). That means our tool limited the LCC argumentation protocol for N-agent to a broadcasting notation. However, we believe that we are able to extend it to work with different types of N-agent protocols by adding more general patterns to the library. These new patterns must be able to work with recursive concepts of DID for N-agent (since recursive concept is considered the most important concepts of N-agent protocols).

B.3.1 General LCC-Argument N-agent Patterns

This section describes three general LCC recursive patterns:

Pattern6:

Name: Recursive Starting (Sending) pattern (RSP)

Problem: How to start an argument (dialogue) for $N \geq 3$? or how to send a message to more than one agents.

Solution: Both agents send/receive a message (locution) and then change their roles so as to remain in the dialogue (Figure B.6).

- (1) Sender (speaker) agent proposes an action (start dialogue) by sending a Recursive Starting locution to all agents and then changes its role.
- (2) Other agents (all agents except the sender agent) receive a Recursive Starting locution and then change their role

Context (Pre-conditions):

- Use this pattern when a sender agent has not already started a dialogue for $N \geq 3$ agents;
- Or, use this pattern when one agent wants to send a message to more than one agents.

Consequence (Post-conditions):

- Sender and other agents engaged in a dialogue.
- Sender and all other agents (receivers) change their roles to remain in the dialogue.

Structure:

a(R1_{Sender}(AgentList, NAgent, Topic), ID_{Sender})::=

R_{Sender} $\xrightarrow{\text{RSL}}$ R_{Receiver}

then

(
 a(R2_{sender} (OtherAgents, NAgent, Topic), ID_{sender}) \leftarrow FailureRecursiveC
 or
 a(R1_{sender} (OtherAgents, NAgent, Topic), ID_{sender})
).

a(R1_{Receiver}(KB_{ID}, CS_{ID}, ID_{Sender}), ID)::=

R_{Receiver} $\xleftarrow{\text{RSL}}$ R_{Sender}

then

a(R2_{Receiver}(KB_{ID}, CS_{ID}, ID_{Sender}), ID).

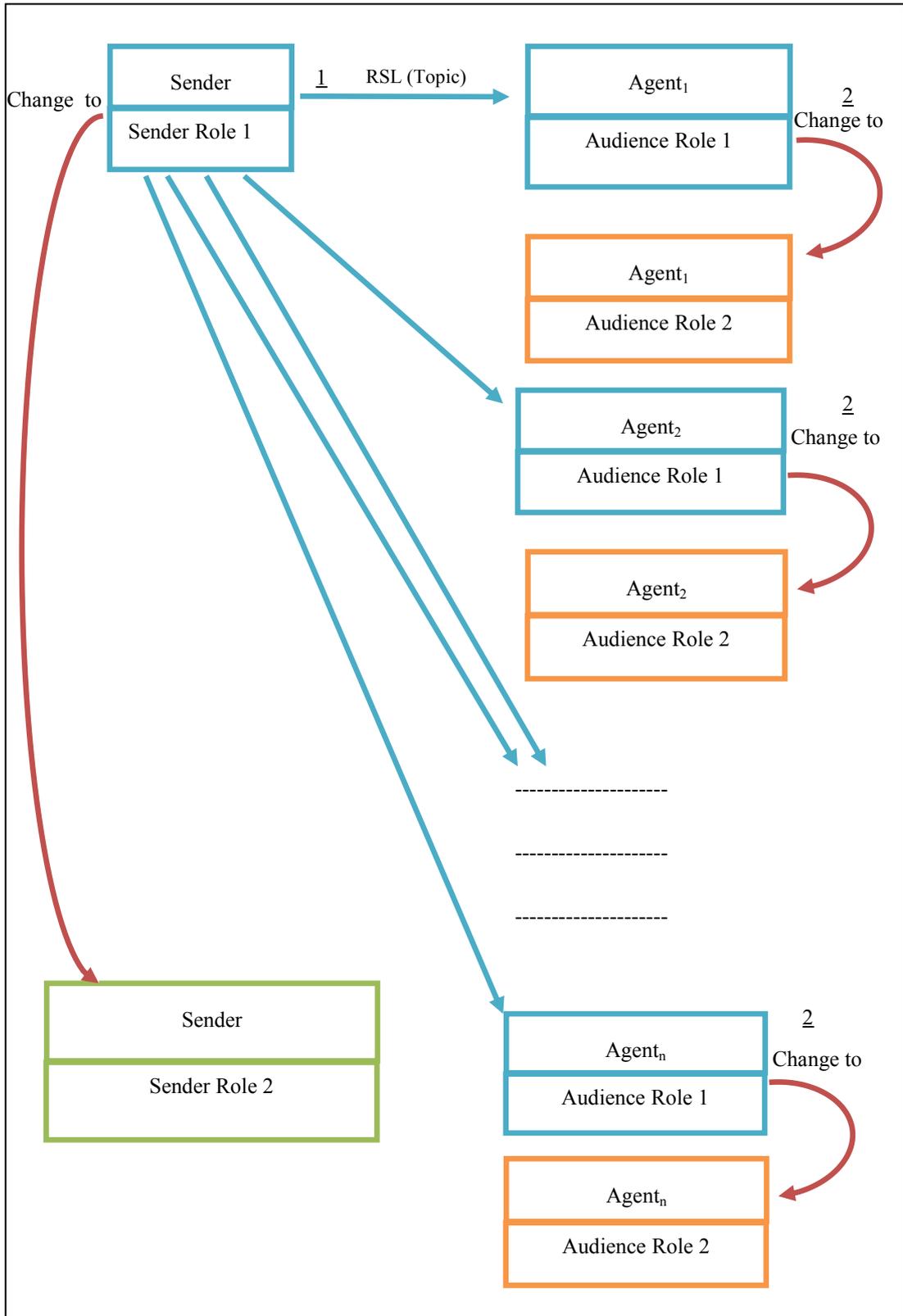


Figure B.6: Recursive Starting(Sending) Pattern Solution

This pattern represents a generic recursive clause. In this pattern, and in the rest of the patterns, RRL represents Recursive Receiving Locution, ' $\approx>$ ' represents outgoing messages from a role, and ' $\approx<$ ' represents incoming messages. $FailureRecursiveC$ represent a condition when it is true the recursive end (usually, $FailureRecursiveC$ is a condition over $AgentList$).

In this LCC code, there are two roles: RI_{Sender} and $RI_{Receiver}$. The RI_{Sender} role of the sender agent ID_{sender} has three input parameters: (1) $AgentList$ which represents the agents ID list; (2) N_{Agent} which represents the number of agents (note that the number of agents is ≥ 3). (3) $Topic$ to open dialogue. The RI_{Sender} role begins by sending a Recursive Starting locution RSL to the $RI_{Receiver}$ role (the ' $\approx>$ ' symbol indicates that the RI_{Sender} role may send one or more different RSLs to the $RI_{Receiver}$ role.). Then, the RI_{Sender} role check $FailureRecursiveC$. If this condition is true, the RI_{Sender} changes its role to the $R2_{Sender}$, otherwise, it recurse.

The $RI_{Receiver}$ role of receiver agent $ID_{Receiver}$ has three input parameters: (1) $KB_{Receiver}$ which represents the agent knowledge base list; (2) $CS_{Receiver}$ which represents the agent commitment store list. Note that CS_A is empty, since $RI_{Receiver}$ represents the first role of the audience agent in the LCC protocol; (3) ID_{Sender} which represents the sender agent identifier. The $RI_{Receiver}$ role begins by receiving a Recursive Starting locution RSL from RI_{Sender} . Then, it changes its role to the $R2_{Receiver}$.

Rewriting methods:

First (Sending method): Rewriting of the " $R_{Sender} \xrightarrow{RSL} R_{Receiver}$ "

The main function of rewriting is to allow generic relations between the R_{Sender} and the $R_{Receiver}$ to be rewritten in a specific way. There might be a direct, complex or indirect relation between them. If there is a general relation " $R_{Sender} \xrightarrow{RSL} R_{Receiver}$ ", then it is possible to specialise it within two different statements:

Rewrite 1: (one locution)

We might specialise " $R_{Sender} \xrightarrow{RSL} R_{Receiver}$ " to an interaction statement that sends a $RRL(Topic)$ message to agent $ID_{Receiver}$, which is achieved by the $RecursiveC$ and CI

constraints. In practice, *RecursiveC* represents a recursive condition (usually, *RecursiveC* is a condition over *AgentList*), *C1* represents a condition (*C1* may represent more than one condition that is connected by *or* and *and* operators) that must be satisfied in order for a sender agent to send the Recursive Starting location (usually, *C1* is a condition over *Topic*).

$$\text{RSL}(\text{Topic}) \Rightarrow a(\text{R}_{\text{receiver}}(\text{KB}_{\text{ID}}, \text{CS}_{\text{ID}}, \text{ID}_{\text{Sender}}), \text{ID}) \leftarrow \text{RecursiveC and C1}$$

Rewrite 2:(multiple locution)

We might specialise " $\text{R}_{\text{Sender}} \xrightarrow{\text{RSL}} \text{R}_{\text{Receiver}}$ " to an interaction statement that sends a *RSL(Topic)* message to agent $\text{ID}_{\text{Receiver}}$ which is achieved by the constraints *RecursiveC* and *C1*. Then, there is another relation between $\text{R}_{\text{Sender}1}$ and $\text{R}_{\text{Receiver}1}$.

$$\begin{array}{l} \text{RSL}(\text{Topic}) \Rightarrow a(\text{R}_{\text{receiver}}(\text{KB}_{\text{ID}}, \text{CS}_{\text{ID}}, \text{ID}_{\text{Sender}}), \text{ID}) \leftarrow \text{RecursiveC and C1} \\ \text{OR} \\ \text{R}_{\text{Sender}} \xrightarrow{\text{RSL}} \text{R}_{\text{Receiver}} \end{array}$$

Second (Receiving method): Rewriting of the " $\text{R}_{\text{Receiver}} \xleftarrow{\text{RSL}} \text{R}_{\text{Sender}}$ "

If there is a general relation " $\text{R}_{\text{Receiver}} \xleftarrow{\text{RSL}} \text{R}_{\text{Sender}}$ ", then it is possible to specialise it within two different statements:

Rewrite 1: (one locution)

We might specialise " $\text{R}_{\text{Receiver}} \xleftarrow{\text{RSL}} \text{R}_{\text{Sender}}$ " to an interaction statement that receive a *RSL(Topic)* message from agent $\text{ID}_{\text{Sender}}$. *C2* represents a condition that must be satisfied after receiver agent receives the Recursive Sending location. In practice, *C2* may represent more than one condition that is connected by *or* and *and* operators. Usually, *C2* is a condition over the role arguments (e.g. *KB* and *CS*).

$$\text{C2} \leftarrow \text{RSL}(\text{Topic}) \Leftarrow a(\text{R}_{\text{1sender}}(\text{KB}_{\text{Sender}}, \text{CS}_{\text{Sender}}, \text{ID}_{\text{Receiver}}), \text{ID}_{\text{sender}})$$

Rewrite 2:(multiple locution)

We might specialise " $\text{R}_{\text{Receiver}} \xleftarrow{\text{RSL}} \text{R}_{\text{Sender}}$ " to an interaction statement that receive a *RSL(Topic)* message from agent $\text{ID}_{\text{Sender}}$. Then, there is another relation between $\text{R}_{\text{Sender}1}$ and $\text{R}_{\text{Receiver}1}$.

$C2 \leftarrow RSL(\text{Topic}) \leq a(R1_{\text{sender}}(KB_{\text{Sender}}, CS_{\text{Sender}}, ID_{\text{Receiver}}), ID_{\text{sender}})$
 or
 $R_{\text{Receiver}} \overset{RSL}{\approx} R_{\text{Sender}}$

Pattern7:

Name: Recursive Receiving Pattern (RRP)

Problem: How to receive a message from more than one agents

Solution :

- (1) One or more agents send(s) the same RRL to the receiver agent and then change(s) their role(s).
- (2) Receiver receive RRL from all other agents (senders) and then change its role to remain in the dialogue.

Context (Pre-conditions): Use this pattern when more than one agents want to send a message to one agent.

Consequence (Post-conditions): Receiver and all other agents (senders) change their roles to remain in the dialogue.

Structure:

$a(R1_{\text{Sender}}(KB, CS, \text{Topic}, ID_{\text{Receiver}}), ID_{\text{Sender}}) ::=$

$R_{\text{Sender}} \overset{RRL}{\approx} R_{\text{Receiver}}$

then

$a(R2_{\text{sender}}(KB, CS, \text{Topic}, ID_{\text{Receiver}}), ID_{\text{sender}})$.

$a(R1_{\text{Receiver}}(\text{AgentList}, \text{SendingList}, N\text{Agent}, \text{Topic}), ID_{\text{Receiver}}) ::=$

$R_{\text{Receiver}} \overset{RRL}{\approx} R_{\text{Sender}}$

then

(

$a(R1_{\text{Receiver}}(\text{AgentList}, \text{OtherSendingLists}, N\text{Agent}, \text{Topic}), ID_{\text{Receiver}})$

$\leftarrow \text{RecursiveC}$

or

$a(R2_{\text{Receiver}}(\text{AgentList}, \text{OtherASendingLists}, N\text{Agent}, \text{Topic}), ID_{\text{Receiver}})$

).

RecursiveC represents a recursive condition (usually, *RecursiveC* is a condition over *AgentList* and *SendingList* e.g. $RecursiveC = notEqual(AgentList, SendingList)$).

In this LCC code, there are two roles: RI_{Sender} and $RI_{Receiver}$. The RI_{Sender} role of the sender agent ID_{sender} has three input parameters: (1) *KB* which represents the agent knowledge base list; (2) *CS* which represents the agent commitment store list; (3) $ID_{Receiver}$ which represents the receiver agent identifier.

The $RI_{Receiver}$ role of audience agent ID_A has four input parameters: (1) *AgentList* which represents the agents ID list.; (2) *SendingList* which represents the sender agents ID list. Initially, *SendingList* is empty; (3) *NAgent* which represents the number of agents (note that the number of agents is ≥ 3). (4) *Topic* which represents the dialogue game topic. The $RI_{Receiver}$ role begins by receiving RRL from RI_{Sender} .

The $RI_{Receiver}$ role begins by receiving a *RRL* message from the RI_{Sender} role (the ' \approx ' symbol indicates that the $RI_{Receiver}$ role may receive one or more different RRLs from the RI_{Sender} role). Then, the $RI_{Receiver}$ role check *RecursiveC*. If this condition is true, the $RI_{Receiver}$ recurse, otherwise, it changes its role to the $R2_{Receiver}$.

Rewriting methods:

First (Sending method): Rewriting of the " $R_{Sender} \xrightarrow{RRL} R_{Receiver}$ "

If there is a general relation " $R_{Sender} \xrightarrow{RRL} R_{Receiver}$ ", then it is possible to specialise it within two different statements:

Rewrite 1: (one locution)

We might specialise " $R_{Sender} \xrightarrow{RRL} R_{Receiver}$ " to an interaction statement that sends a *RRL(Topic)* message to agent $ID_{Receiver}$, which is achieved by the constraint *CI*. In practice, *CI* may represent more than one condition that is connected by *or* and *and* operators. Usually, *CI* is a condition over the role arguments (e.g. *KB* and *CS*).

$$\boxed{RRL(Topic) \Rightarrow a(R_{Receiver}(AgentList, NAgent, N, Topic), ID_{Receiver}) \leftarrow CI}$$

Rewrite 2:(multiple locution)

We might specialise " $R_{Sender} \overset{RRL}{\rightsquigarrow} R_{Receiver}$ " to an interaction statement that sends a $RRL(Topic)$ message to agent $ID_{Receiver}$ which is achieved by the constraint $C1$. Then, there is another relation between $R_{Sender1}$ and $R_{Receiver1}$.

$$\begin{array}{l} RRL(Topic) \Rightarrow a(R_{Receiver}(AgentList, NAgent, N, Topic), ID_{Receiver}) \leftarrow C1 \\ \text{or} \\ R_{Sender} \overset{RRL}{\rightsquigarrow} R_{Receiver} \end{array}$$

Second(Receiving method): Rewriting of the " $R_{Receiver} \overset{RRL}{\leftarrow} R_{Sender}$ "

If there is a general relation " $R_{Receiver} \overset{RRL}{\leftarrow} R_{Sender}$ ", then it is possible to specialise it within two different statements:

Rewrite 1: (one locution)

We might specialise " $R_{Receiver} \overset{RRL}{\leftarrow} R_{Sender}$ " to an interaction statement that receive a $RRL(Topic)$ message from agent ID_{Sender} . $C2$ represents a condition that must be satisfied after receiver agent receives the Recursive Receiving locution. In practice, $C2$ may represent more than one condition that is connected by *or* and *and* operators. Usually, $C2$ is a condition over the recursive arguments. (Note that if $C2$ does not work with all recursive arguments, the tool will write the recursive argument as the $C2$ condition automatically. See section B.2.2 for more a detailed example).

$$C2 \leftarrow RRL(Topic) \leftarrow a(R_{Sender1}(KB_{ID}, CS_{ID}, ID_{Receiver}), ID_{Sender1})$$

Rewrite 2:(multiple locution)

We might specialise " $R_{Receiver} \overset{RRL}{\leftarrow} R_{Sender}$ " to an interaction statement that receive a $RRL(Topic)$ message from agent ID_{Sender} . Then, there is another relation between $R_{Sender1}$ and $R_{Receiver1}$.

$$\begin{array}{l} C2 \leftarrow RRL(Topic) \leftarrow a(R_{Sender1}(KB_{ID}, CS_{ID}, ID_{Receiver}), ID_{Sender1}) \\ \text{or} \\ R_{Receiver1} \overset{RRL}{\leftarrow} R_{Sender1} \end{array}$$

Pattern8:

Name: Recursive Termination-Sending Pattern (RTSP)

Problem: How to send and change roles or terminate an argument (dialogue) for $N \geq 3$ agents.

Solution :

(1) Dialogue Termination (Recursive Termination locution) (Figure B.7):

- The sender agent sends *Recursive Termination* locution to all other agents and then terminates its role.
- All other agents receive *Recursive Termination* locution and then terminate their roles.

(2) Sending and Changing roles (Figure B.6):

- Sender agent sends a Recursive Starting locution to all agents and then changes its role .
- All receiver agents receive a Recursive Starting and then change their roles.

Context (Pre-conditions): Use Recursive Termination-Sending pattern to send a message and change roles, or to terminate a dialogue between 3 or more agents (when agents reach an agreement).

Consequence (Post-conditions):

(1) Dialogue Termination :

- The dialogue between N-agent is terminated

(2) Sending and Changing roles:

- The sender agent and all receiver agents change their roles to remain in the dialogue.

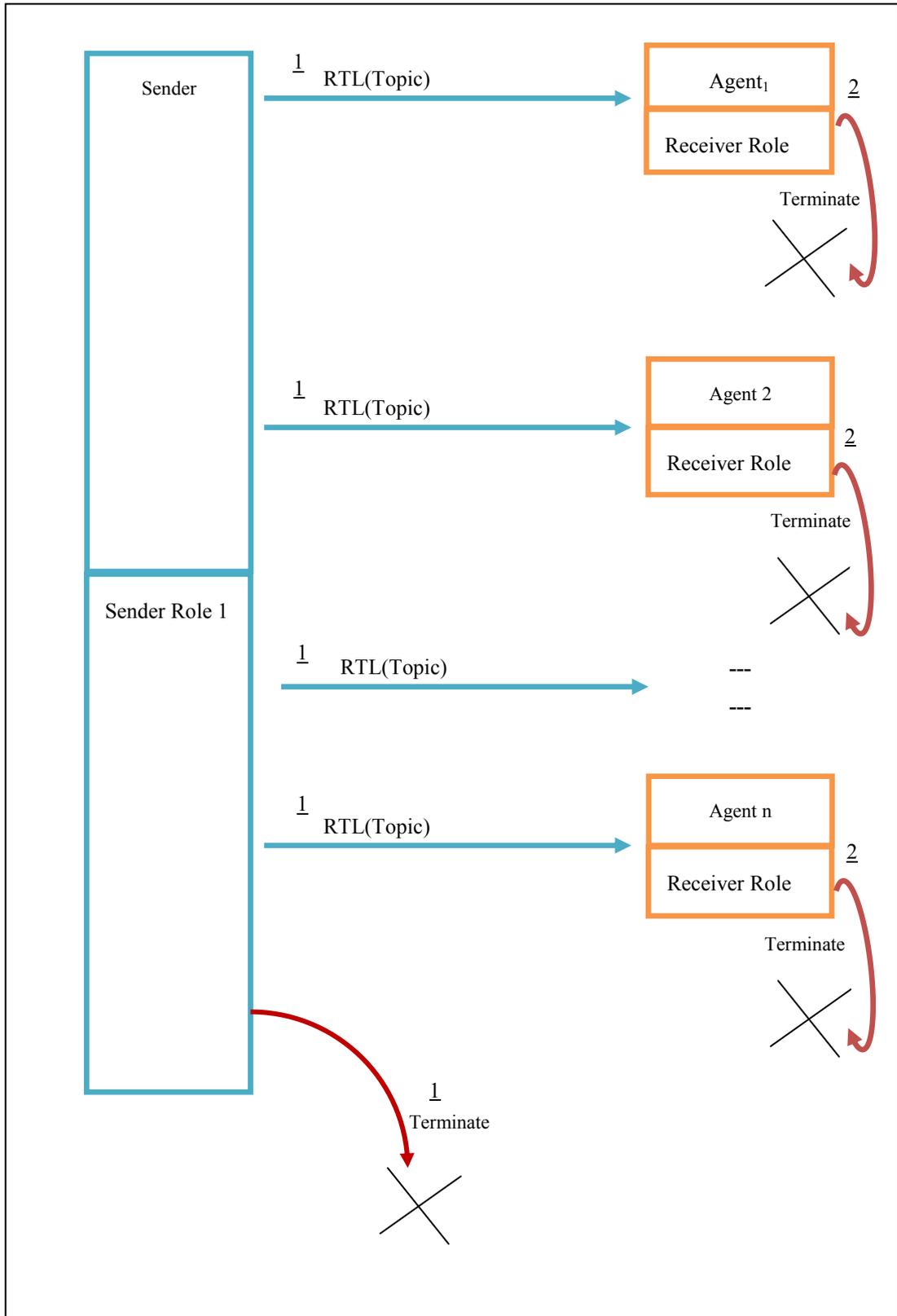


Figure B.7: Recursive Termination-Sending Pattern (Termination) Solution

Structure:

```

a(Rsender( AgentList, Topic),IDsender) ::=
  (
    RSender       $\overset{RTL}{\approx\Rightarrow}$       RReceiver
    then
    (
      null  $\leftarrow$  FailureRecursiveC1
      or
      a(Rsender (OtherAgents, Topic),IDsender)
    )
  )
or
  (
    RSender       $\overset{RSL}{\approx\Rightarrow}$       RReceiver
    then
    (
      a(R2sender (OtherAgents, Topic),IDsender)
       $\leftarrow$  FailureRecursiveC2
      or
      a(Rsender (OtherAgents, Topic),IDsender)
    )
  ).
a(RReceiver(KBID,CSID,IDSender), ID) ::=
  RReceiver       $\overset{RTL}{\approx\Leftarrow}$       RSender
  or
  (
    RReceiver       $\overset{RSL}{\approx\Leftarrow}$       RSender
    then
    a(R2Receiver(KBID,CSID, IDSender), ID)
  ).
  
```

This pattern represents a generic recursive clause. *RTL* represents the Recursive Termination locution and *FailureRecursiveC* represents a condition that when it is true forces the recursion to end (usually, *FailureRecursiveC* is a condition over *AgentList*).

In this LCC code, there are two roles: *R_{Sender}* and *R_{Receiver}*. The *R_{Sender}* role of the sender agent *ID_{sender}* has two input parameters: *AgentList* and *Topic*. It begins by

either: (1) sending a Recursive Termination locution. Then, the R_{Sender} role checks $FailureRecursiveC1$. If this condition is true, the R_{Sender} terminates, otherwise, it recurse; (2) sending a Recursive Starting locution RSL to the $R_{Receiver}$ role (the ' $\approx>$ ' symbol indicates that the R_{Sender} role may send one or more different RSLs to the $R_{Receiver}$ role.). Then, the R_{Sender} role check $FailureRecursiveC2$. If this condition is true, the R_{Sender} changes its role to the $R2_{Sender}$, otherwise, it recurse.

The $R_{Receiver}$ role of audience agent ID has three input parameters: (1) $KB_{Receiver}$ which represents the agent knowledge base list; (2) $CS_{Receiver}$ which represents the agent commitment store list.; (3) ID_{Sender} which represents the sender agent identifier. The $R_{Receiver}$ role begins by either receiving: (1) a Recursive Termination locution from R_{Sender} (the ' $\approx<$ ' symbol indicates that the $R_{Receiver}$ role may receive one or more different RTLs from the R_{Sender} role); or (2) a Recursive Starting locution RSL from R_{Sender} . Then, it changes its role to the $R2_{Receiver}$.

Rewriting methods:

First (Sending Termination method): Rewriting of the " $R_{Sender}^{RTL \approx>} R_{Receiver}$ "

If there is a general relation " $R_{Sender}^{RTL \approx>} R_{Receiver}$ ", then it is possible to specialise it within two different statements:

Rewrite 1: (one termination locution)

We might specialise " $R_{Sender}^{RTL \approx>} R_{Receiver}$ " to an interaction statement that sends a $RTL(Topic)$ Recursive Termination message to agent $ID_{Receiver}$, which is achieved by the $RecursiveC$ and $C1$ constraints. In practice, $RecursiveC$ represents a recursive condition (usually, $RecursiveC$ is a condition over $AgentList$), $C1$ represents a condition ($C1$ may represent more than one condition that is connected by *or* and *and* operators) that must be satisfied in order for a sender agent to send the Recursive Termination locution (usually, $C1$ is a condition over $Topic$).

$$RTL(Topic) \Rightarrow a(R_{receiver}(KB_{ID}, CS_{ID}, ID_{sender}), ID) \leftarrow RecursiveC \text{ and } C1$$

Rewrite 2:(multiple termination locution)

We might specialise " $R_{Sender} \xrightarrow{RTL} R_{Receiver}$ " to an interaction statement that sends a $RTL(Topic)$ Recursive Termination message to agent $ID_{Receiver}$ which is achieved by the *RecursiveC* and *C1* constraints. Then, there is another termination relation between R_{Sender} and $R_{Receiver}$.

$$\begin{array}{l} \text{RTL(Topic)} \Rightarrow a(R_{Receiver}(KB_{ID}, CS_{ID}, ID_{sender}), ID) \leftarrow \text{RecursiveC and C1} \\ \text{Or} \\ R_{Sender} \xrightarrow{RTL} R_{Receiver} \end{array}$$

Second (Receiving Termination method): Rewriting of the " $R_{Receiver} \xleftarrow{RTL} R_{Sender}$ "

If there is a general relation " $R_{Receiver} \xleftarrow{RTL} R_{Sender}$ ", then it is possible to specialise it within two different statements:

Rewrite 1: (one locution)

We might specialise " $R_{Receiver} \xleftarrow{RTL} R_{Sender}$ " to an interaction statement that receive a $RTL(Topic)$ message from agent ID_{Sender} . *C2* represents a condition that must be satisfied after receiver agent receives the Recursive Termination locution. In practice, *C2* may represent more than one condition that is connected by *or* and *and* operators. Usually, *C2* is a condition over the role arguments (e.g. *KB* and *CS*).

$$C2 \leftarrow RTL(Topic) \Leftarrow a(R1_{sender}(KB_{Sender}, CS_{Sender}, ID_{Receiver}), ID_{sender})$$

Rewrite 2:(multiple locution)

We might specialise " $R_{Receiver} \xleftarrow{RTL} R_{Sender}$ " to an interaction statement that receive a $RTL(Topic)$ message from agent ID_{Sender} . Then, there is another relation between $R_{Sender1}$ and $R_{Receiver1}$.

$$\begin{array}{l} C2 \leftarrow RTL(Topic) \Leftarrow a(R1_{sender}(KB_{Sender}, CS_{Sender}, ID_{Receiver}), ID_{sender}) \\ \text{or} \\ R_{Receiver} \xleftarrow{RSL2} R_{Sender} \end{array}$$

Third (Sending method): Rewriting of the " $R_{Sender} \xrightarrow{RSL} R_{Receiver}$ "

See rewriting method of Recursive Sending Pattern (*Rewriting of the " $R_{Sender} \xrightarrow{RSL} R_{Receiver}$ "*).

Fourth(Receiving method): Rewriting of the " $R_{Receiver} \approx^{RSL} R_{Sender}$ "

See rewriting method of Recursive Sending Pattern (*Rewriting of the " $R_{Receiver} \approx^{RSL} R_{Sender}$ "*).

Pattern9:

Name: Recursive Termination-Divided Pattern (RTDP)

Problem: How to divide agents into groups of two or terminate an argument (dialogue) for $N \geq 3$ agents.

Solution :

(1) Dialogue Termination (Recursive Termination locution) (Figure B.7):

- The sender agent sends *Recursive Termination* locution to all other agents and then terminates its role.
- All other agents receive *Recursive Termination* locution and then terminate their roles.

(2) Divide agents (chapter 5, Figure 5.3):

- The sender agent sends *argueWith(Topic,Agent_P,Agent_O)* locution for a pair of agents: *Agent_P* and *Agent_O* (telling them to interact together) and then recurses or changes its role.
- Both *Agent_P* and *Agent_O* receive *argueWith(Topic,Agent_P,Agent_O)* locution and then change their roles to *startDID* role.

Context (Pre-conditions): Use Recursive Termination-Divided pattern to divide agents into groups or to terminate a dialogue between 3 or more agents (when agents reach an agreement).

Consequence (Post-conditions):

(1) Dialogue Termination :

- The dialogue between N-agent is terminated

(2) Divide agents:

- Divide agents into groups of two and start dialogues between two agents.

Structure:

Figure B.8 illustrates the structure of this pattern. This pattern represents a generic recursive clause. *FailureRecursiveC* represents a condition when it is true the recursive end (usually, *FailureRecursiveC* is a condition over *AgentList*).

In this LCC code, there are four roles: R_{Sender} , *TerminationR_{Sender}*, *divideGroup_{Sender}* and R_{Receiver} . The R_{Sender} role of the sender agent ID_{sender} has nine input parameters: *AgentList*, *NAgent*, *NSupporters*, *Topic*, *NReply*, *AcceptingList*, *RejectionList*, *NAccepting* and *NRejection*. The R_{Sender} role begins by checking *TerminationC* condition. If this condition is true, then the proposal agent changes its role to the *TerminationR_{sender}* role. Otherwise, the R_{Sender} role checks *DivideC* condition. If this condition is true, then the sender agent changes its role to the *divideGroup_{proposal}* role.

The *TerminationR_{Sender}* role of the sender agent ID_{sender} has two input parameters: *AgentList* and *Topic*. It begins by sending a Recursive Termination locution (the '≈>' symbol indicates that the *TerminationR_{Sender}* role may send one or more different RTLs to the R_{Receiver} role). Then, the *TerminationR_{Sender}* role check *FailureRecursiveC1*. If this condition is true, the *TerminationR_{Sender}* terminates, otherwise, it recurse;

The *divideGroup_{Sender}* role of the sender agent ID_{sender} has six input parameters: *AgentList*, *NAgent*, *NSupporters*, *Topic*, *AcceptingList* and *RejectionList*. It also has one output parameter: *AgentGroup*. This role is responsible for dividing the agents in

```

a(Rsender(AgentList,NAgent,NSupporters,Topic,AcceptingList,
RejectionList, AgentGroup),IDsender)::=
a(TerminationRsender (AgentAgents, Topic),IDsender) ← TerminationC

or

a(divideGroupsender (AgentList , NAgent,NSupporters,Topic,
AcceptingList,RejectionList, [ ]),IDproposal) ← DivideC.

a(TerminaitonRsender( AgentList, Topic),IDsender)::=
    TerminaitonRSender RTL≈> RReceiver
    then
    ( null ← FailureRecursiveC1
    or
    a(Rsender (OtherAgents, Topic),IDsender) ).

a(divideGroupSender (AgentList, NAgent,NSupporters,Topic,
AcceptingList,RejectionList,AgentGroup), IDSender ):=
    (
    argueWith (Topic,P,O) => a(RRecevier (KBp,CSp,IDSender), P)
    ← RecursiveC
    then
    argueWith (Topic,O,P) => a(RRecevier (KBo,CSo,IDSender), O)
    )

    then
    (
    a(recursSender(AgentList, NAgent,NSupporters ,0 ,Topic),IDSender)
    ← FailureRecursiveC2
    or
    a(divideGroupSender(AgentList ,NAgent,NSupporters,Topic,Ac,Re,AGroup),
    IDSender)
    ).

a(RRecevier(KBID,CSID,IDSender), ID)::=
    RReceiver <≈RTL TerminationRSender

    or

    (
    argueWith(Topic,ID,ID2) <= a(divideGroupSender( _ , _ , _ , _ , _ , _ , _ ),IDSender)
    then
    a(startDID(KBID,CSID ,Topic, IDSender, ID2),ID)
    ).
    
```

Figure B.8: Recursive Termination-Divided Pattern Structure

the *AgentList* list into a group composed of two agents. It begins by checking *RecursiveC*. If this condition is true, then this role creates the first agent group by taking one agent from the head of the *AcceptingList* and one agent from the head of the *RejectionList*. It then sends the *argueWith* message to the first group (agent *P* and agent *O*) and asks them to start arguing together about the dialogue *Topic*. Then, if the *FailureRecursiveC2* condition is true, the sender agent changes its role to the *recursProposal* role (see chapter 5, Recurs-To-N-Dialogue Pattern), otherwise, it recurses.

The $R_{Receiver}$ role of audience agent ID has three input parameters: (1) $KB_{Receiver}$ which represents the agent knowledge base list; (2) $CS_{Receiver}$ which represents the agent commitment store list.; (3) ID_{Sender} which represents the sender agent identifier. The $R_{Receiver}$ role begins by either: (1) receiving a Recursive Termination locution from $TerminaitonR_{Sender}$ (the ' \approx ' symbol indicates that the $R_{Receiver}$ role may receive one or more different RTLs from the $TerminaitonR_{Sender}$ role); (2) receiving an *argueWith* message from $divideGroup_{Sender}$. Then, it changes its role to the *startDID* role(see chapter 5, Move-To-Dialogue Pattern).

Rewriting methods:

First (Sending Termination method): Rewriting of the " $TerminaitonR_{Sender} \xrightarrow{RTL} R_{Receiver}$ "

See rewriting method of Recursive Termination-Sending Pattern (*Rewriting of the " $R_{Sender} \xrightarrow{RTL} R_{Receiver}$ "*).

Second (Receiving Termination method): Rewriting of the " $R_{Receiver} \xleftarrow{RTL} TerminaitonR_{Sender}$ "

See rewriting method of Recursive Termination-Sending Pattern (*Rewriting of the " $R_{Receiver} \xleftarrow{RTL} R_{Sender}$ "*).

Pattern10:

Name: Receiving/Sending Recursive Pattern (RSRP)

Problem: How to send and receive more than one message?

Solution :

- (1) Sender agent sends a RSL to more than one agent and then changes its role.
- (2) Receiver agent receive RRL from more than one agent (senders) and then change its role.

Context (Pre-conditions):

- Use this pattern when one agent wants to send a message to more than one agent and more than one agent want to send a message to one agent.

Consequence (Post-conditions):

- All other agents (senders and receivers) change their roles to remain in the dialogue.

Structure: This pattern is a combination of Pattern 6 and 7 (see pattern 6 and pattern 7).

a(R1_{Sender}(AgentList, NAgent, Topic), ID_{Sender}) ::=

R_{Sender} RSL \approx > R_{Receiver}

then

(
 a(R2_{sender} (OtherAgents, NAgent, Topic), ID_{sender}) \leftarrow FailureRecursiveC
 or
 a(R1_{sender} (OtherAgents, NAgent, Topic), ID_{sender})
).

a(R1_{Receiver} (AgentList, SendingList, NAgent, Topic), ID_{Receiver}) ::=

R_{Receiver} \leftarrow \approx RRL R_{Sender}

then

(
 a(R1_{Receiver} (AgentList, OtherSendingLists, NAgent, Topic), ID_{Receiver})
 \leftarrow RecursiveC
 or
 a(R2_{Receiver} (AgentList, OtherASendingLists, NAgent, Topic), ID_{Receiver})
).

Rewriting methods:

First (Sending method): Rewriting of the " $R_{Sender} \xrightarrow{RSL} R_{Receiver}$ "

See rewriting method of Recursive Sending Pattern (*Rewriting of the " $R_{Sender} \xrightarrow{RSL} R_{Receiver}$ "*).

Second(Receiving method): Rewriting of the " $R_{Receiver} \xleftarrow{RRL} R_{Sender}$ "

See rewriting method of Recursive Receiving Pattern (*Rewriting of the " $R_{Receiver} \xleftarrow{RRL} R_{Sender}$ "*).

B.3.2 Automated Synthesis Steps for Generating Agent Protocol for General N-agent Automatically

The N-agents' general protocol automated synthesis algorithm is illustrated in Figure B.9:

- (1) The tool begins with the locution icon at the top of the DID. Note that if more than one locution icon appears in one level, then the tool begins with the locution to the left (since it works from left to right).
- (2) Following this, the tool selects one pattern from the LCC-Argument patterns for general N-agent protocol library. This pattern depends on the locution type. Note that each locution type is connected to only one LCC-Argument pattern. See Table B.1.
- (3) After that, if the selected pattern has rewriting methods, the tool selects one or more of the rewriting methods. The number of rewriting methods selected is dependent on the number of locution icons in this level. If this level has one locution icon, the tool selects the rewriting method **Rewrite 1** (rewriting method with one locution). If this level has more than one locution icon, the tool selects the rewriting method **Rewrite 2** (rewriting method with multiple locutions).

1. **Input** (DID, LCC-Argument patterns)
2. **Select&Save** Icon= one DID locution icon (**Step1**)
3. **Select&Save** Pattern= one pattern from the LCC-Argument patterns for general N-agent protocol library (**Step2**)
4. **If** (Pattern has rewriting methods) then (**Step3**)
5. **If** (level has one locution icon) then
6. **Select&Save** RewriteMethod=**Rewrite 1**
7. **If** (level has more than one locution icon) then
8. **Select&Save** RewriteMethod=**Rewrite 2**
9. **Match** (Icon,Pattern,RewriteMethod) (**Step4**)
10. **If** (Pattern =*Recursive Termination-Divided*) then (**Step5**)
11. **Use** Recurs-To-N-Dialogue Pattern
12. recursNumber = number of Termination locution icon in the DID for two agents -1
13. **If** (reurseNumber = 0) then *//one Termination Locution*
14. **Select&Save** RewriteMethod2=**Rewrite 1**
15. **Match** (Termination Icon, Recurs-To-N-Dialogue Pattern, RewriteMethod2)
16. **Else** *//more than one Termination Locution*
17. **Loop begin** (if i=1)
18. **Select&Save** RewriteMethod2=**Rewrite 2**
19. **Match** (Termination Icon, Recurs-To-N-Dialogue Pattern, RewriteMethod2)
20. i= i+1
21. **Loop end** (if i = reurseNumber)
22. **Go To** two agents algorithm
23. **Add** lines to connect N-agents' protocol with two agents' protocol
24. **Go To** line 2 (**Step6**)
25. **Output** LCC protocol

Figure B.9: N- Agents Protocol Automated Synthesis Algorithm

Locution Type	Pattern Name
Recursive Starting Locution	Recursive Starting (Sending) Pattern
Recursive Receiving Locution	Recursive Receiving Pattern
Recursive Termination Locution and Divided Locution	Recursive Termination-Divided
Recursive Termination Locution and Recursive Starting Locution	Recursive Termination-Sending Pattern

Table B.1 Relationship Between Locution Type and Patterns

- (4) Then, the tool applies the selected pattern by matching formal parameters (variables) with its corresponding values in the locution icon to generate pairs of LCC clauses or roles (sender and receiver roles). If the selected pattern has rewriting methods, the tool matches the formal parameters in the selected rewriting methods with its corresponding values in the locution icon to generate pairs of LCC clauses or roles. The matching process matches one parameter at a time. It begins with the locution icon and occurs from the top-down and left to right. It then moves to the left side conditions and then to the right side conditions. Finally, if the selected pattern has recursive (changing) roles, the tool moves to the next level and matches the recursive roles in the pattern with the recursive roles in the locution icon on the next level.
- (5) After that, the selected pattern is the *Recursive Termination-Divided* pattern. The tool uses the *Rekurs-To-N-Dialogue Pattern* to generate the LCC role which is used to inform the proposal agent about the ending of the dialogue between two agents:
 - a) The tool selects one or more rewriting methods. The number of selected rewriting methods is the number of the *Termination Locution* icons in the DID for two agents, minus one. For example, if the number of *Termination Locution* icons is equal to five, then the number of *end messages* is equal to $5 \times 2 = 10$ and the number of rewriting methods is equal $5-1= 4$ (each rewriting methods has two end messages and by default *Rekurs-To-N-Dialogue* pattern receives two *end messages*, one from the first *Termination Locution* sender role and one from the first *Termination Locution* receiver role).
 - b) The tool applies this pattern by matching the formal parameters with their corresponding values in the Termination locution icons in the DID for two agents, to generate one of the LCC clauses or roles for the proposal agent.
 - c) Finally, the tool follows the steps of the automated synthesis process of two agents' protocol to generate the LCC protocol for DID for two agents. Note that the tool adds two lines after each *Termination Locution* (message) in the

LCC protocol for two agents to connect N-agents' protocol with two agents' protocol:

- Line one: Sending end message to proposal.
- Line two: Changing agents' role to the receiver role of the locution icon at the top of the DID of the dialogue between N-agent.

```
(
  TL (Topic) => a(R, ID)
  then
  end(Topic)=>
  a(recursProposal(AgentList,NAgent,NSupporters,NReply,Topic),IDProposal)
  then
  a(FirstReceiverRoleID (KBID,CSID, IDproposal), ID)
)
```

- (6) Moves to the next level in the DID for N-agent and repeats steps 2, 3,4 and 5. Note that the automated synthesis process finishes when the tool matches the last level in the DID with one of the LCC-Argument patterns. If the selected pattern has recursive (changing) roles, the tool moves to the locution icon reply level, which represents the reply rules of the selected locution icon, and matches the recursive roles in the pattern with the recursive roles in the locution icon on this level.

B.3.3 An Example of an LCC Protocol begin generated for General N-agent Dialogue

This section represents the generated LCC protocol from the automated agent protocol synthesis tool "***GenerateLCCProtocol***". In this example, the tool receives as input the DID of a persuasion dialogue between N-agent, which is shown in Figure B.3. Then the tool generates the LCC protocol by using LCC-Argument patterns (N-agent general patterns). The final LCC protocol is illustrated in Figures B.10(a), B.10(b), B.10(c), and B.10(d). Please see appendix C for a detailed description of how to transfer a DID to an LCC protocol by using LCC-Argument patterns:

Prposal	Other Agents
<pre> a(proposalSender_{proposal}(AgentList,NAgent,NSupporters,Topic),ID_{proposal})::= proposal(Topic) => a(proposalReceiver_{ID}(KB_{ID},CS_{ID},ID_{proposal}), ID) ←getAgentIDFromList(AgentList,otherAgents,ID) and addTopicToCS(Topic,CS_{proposal}) then (a(replyToProposalReceiver_{proposal} (AgentList, NAgent,NSupporters,Topic,[],[],0,0, []), ID_{proposal}) ← agentListEmpty(AgentList) or a(proposalSender_{proposal} (OtherAgents, NAgent,NSupporters,Topic), ID_{proposal})).</pre>	<pre> a(proposalReceiver_{ID}(KB_{ID},CS_{ID},ID_{proposal}), ID)::= proposal(Topic)<= a(proposalSender_{proposal}(AgentList,NAgent,NSupporters,Topic), ID_{proposal}) then a(replyToProposalSender(KB_{ID},CS_{ID},Topic,ID_{proposal}), ID).</pre>

Figure B.10(a): Generated LCC Protocol for N-agent Dialogue

- (1) The tool begins with the locution icon at the top of the DID (See Figure B.2) of the persuasion dialogue between N-agent, which is *proposal(Topic)*.
- (2) The tool then selects the *Recursive Starting (Sending) Pattern* (since the locution type is the *Recursive Starting Locution*).
- (3) The tool applies the *Recursive Starting (Sending) Pattern* by matching formal parameters in the *Recursive Starting (Sending) Pattern* with its corresponding values in the *proposal(Topic)* icon, starting from the top-down and moving left to right.
- (4) The tool moves to the next level (level two of the DID of the persuasion dialogue).
- (5) Following this, the tool selects the *Recursive Receiving Pattern* (since the locution type is the *Recursive Receiving Locution*).
- (6) The tool applies the *Recursive Receiving Pattern*.
- (7) Moves to the next level (level three of the DID of the persuasion dialogue).

Prposal	Other agents
<pre> a(replyToPrposalReceiver_{proposal}(AgentList,NAgent,NSupporters,Topic,AcceptingList,RejectionList,NAccepting,NRejection,SendingList), ID_{proposal}) ::= (addIDToList(SendingList,OtherSedingList,ID) and addToAcceptingList(AcceptingList,AccList,ID) and increaseAccepting(NAccepting,NAcc) and RejList= RejectionList and NRej is NRejection← accept(Topic) <=a(replyToProposalSender_{ID}(KB_{ID},CS_{ID},Topic,ID_{proposal}), ID) or addToRejectingList(RejectingList,RejList,ID) and increaseRejecting(NRejecting,NRej) and increaseReply (NReply,NRep) and AccList=AcceptingList and NAcc is NAccepting ← reject(Topic) <=a(replyToProposalSender_{ID}(KB_{ID},CS_{ID},Topic,ID_{proposal}), ID)) then (a(replyToPrposalReceiver_{proposal}(AgentList,NAgent,NSupporters,Topic,AccList,RejList,NAcc,NRej,OtherSendingList), ID_{proposal}) ← notEqual(AgentList,SendingList) or a(resultSender_{proposal} (AgentList,NAgent,NSupporters,Topic,NReply,AcceptingList,RejectionList,NAccepting,NRejection), ID_{proposal})). </pre>	<pre> a(replyToPrposalSender_{ID}(KB_{ID},CS_{ID},Topic,ID_{proposal}), ID) ::= (accept(Topic) => a(replyToPrposalReceiver_{proposal} (_ , _ , _ , _ , _ , _ , _ , _ , _ , _ , ID_{proposal}) ← findTopicInKB(Topic, KB_{ID}) and notFindTopicInCS (Topic,CS_{ID}) and notFindOppTopicInCS (not(Topic),CS_{ID}) and addTopicToCS(Topic,CS_{ID}) or reject(Topic) => a(replyToPrposalReceiver_{proposal} (_ , _ , _ , _ , _ , _ , _ , _ , _ , _ , ID_{proposal}) ← notFindTopicInKB(Topic,KB_{Proposal}) and notFindTopicInCS(Topic,CS_{Proposal}) then a(resultReceiver_{ID}(KB_{ID},CS_{ID},Topic,ID_{proposal}), ID) . </pre>

Figure B.10(b): Generated LCC Protocol for N-agent Dialogue

Prposal	Other agents
<pre> a(resultSender_{proposal}(AgentList,NAgent, NSupporters,Topic,AcceptingList,RejectionList, AgentGroup), ID_{proposal}) ::= a(sendReachAgreement_{proposal} (AgentList,Topic),ID_{proposal}) ← greaterThanOrEqual(NAccepting,NSupporters) or a(divideGroup_{proposal} (AgentList , NAgent,NSupporters ,Topic,AcceptingList,RejectionList, []),ID_{proposal}) ← lessThan(NAccepting,NSupporters) and isEmpty(RejectionList) and isEmpty(AcceptingList) a(sendReachAgreement_{proposal} (AgentList, Topic),ID_{proposal}) ::= reachAgreement(Topic) => a(resultReceiver_{ID}(KB_{ID},CS_{ID},ID_{proposal}), ID) ← getAgentIDFromList(AgentList,otherAgents,ID) then (null ← isEmpty(AgentList) or a(sendReachAgreement_{proposal} (OtherAgents, Topic), ID_{proposal})). a(divideGroup_{proposal}(AgentList,NAgent, NSupporters,Topic,AcceptingList,RejectionList, AgentGroup), ID_{proposal})::= (argueWith (Topic,P,O) => a(resultReceiver_p(KB_p,CS_p,Topic,ID_{proposal}), P) ← creatOneAgentGroup(Rejecting,Re,Accepting,Ac, AgentGroup, AGroup,P,O) then argueWith (Topic,O,P) => a(resultReceiver_o(KB_o,CS_o,Topic,ID_{proposal}), O)) then (a(recurs_{proposal} (AgentList, NAgent,NSupporters ,0 ,Topic),ID_{proposal}) ← isEmpty(Re) or isEmpty(Ac) or a(divideGroup_{proposal}(AgentList ,NAgent, NSupporters,Topic,Ac,Re,AGroup),ID_{proposal})). </pre>	<pre> a(resultReceiver_p(KB_p,CS_p,Topic,ID_{Sender}),P) ::= reachAgreement(Topic) <= a(sendReachAgreement_{proposal} (AgentList, Topic),ID_{proposal}) or (argueWith(Topic,P,O) <= a(divideGroup_{Sender}(AgentList, NAgent,NSupporters,Topic,AcceptingList, RejectionList,AgentGroup),ID_{Sender}) then a(startDID(KB_p,CS_p,Topic,ID_{Sender}, O),P)). a(resultReceiver_o(KB_o,CS_o,Topic,ID_{Sender}), O) ::= reachAgreement(Topic) <= a(sendReachAgreement_{proposal} (AgentList, Topic),ID_{proposal}) or (argueWith(Topic,O,P) <= a(divideGroup_{Sender}(AgentList, NAgent,NSupporters,Topic,AcceptingList, RejectionList,AgentGroup),ID_{Sender}) then a(startDID(KB_o,CS_o,Topic, ID_{Sender}, P),O)). </pre>

Figure D.10(c): Generated LCC Protocol for N-agent Dialogue

Prposal	Other agents
<pre> a(recurs_{Proposal} (AgentList, NAgent,NSupporters,replyN,Topic),ID_{Proposal}) ::= (N = replyN +1 ← end(Topic) <= a(replyToClaimSender_O (KB_O,CS_O,CS_P,T,ID_{Proposal},ID_P),ID_O) or N = replyN +1 ← end(Topic)<= a(replyToClaimReceiver_P (KB_P,CS_P,CS_O,T,ID_{Proposal},ID_O),ID_P) or N = replyN +1 ← end(Topic) <=a(replyToWhySender_P(KB_P,CS_P,CS_O,T,ID_{Propo} _{sal},ID_O),ID_P) or N = replyN +1 <-- end(Topic) <= a(replyToWhyReceiver_O (KB_O,CS_O,CS_P,T,ID_{Proposal},ID_P),ID_O) or N = replyN +1 ← end(Topic) <= a(replyToArgueSender_O (KB_P,CS_P,CS_O,T,Pre,ID_{Proposal},ID_O),ID_O) or N = replyN +1 <-- end(Topic) <= a(replyToArgueReceiver_P (KB_O,CS_O,CS_P,T,Pre,ID_{Proposal},ID_P),ID_P)) then (a(proposalSender_{proposal} (AgentList,NAgent,NSupporters,Topic), ID_{proposal}) ← isEqual(N, NAgent) or a(recurs_{Proposal} (AgentList, NAgent,NSupporters, N,Topic),ID_{Proposal}). </pre>	<pre> a(startDID_P(KB_P,CS_P,CS_O,T,ID_{Proposal},ID_O),ID_P) ::= a(claimSender (KB_P,CS_P,CS_O,T,ID_{Proposal},ID_O), ID_P) ← addTopicToCS(T,CS_P) or a(claimReceiver (KB_P,CS_P,CS_O,T,ID_{Proposal},ID_O), ID_P). a(startDID_O(KB_O,CS_O,CS_P,T,ID_{Proposal},ID_P),ID_O) ::= = a(claimSender (KB_O,CS_O,CS_P,T,ID_{Proposal},ID_P), ID_O) ← addTopicToCS(T,CS_P) or a(claimReceiver (KB_O,CS_O,CS_P,T,ID_{Proposal},ID_P), ID_O). </pre>

Figure B.10(d): Generated LCC Protocol for N-agent Dialogue

- (8) Following this, the tool selects the *Recursive Termination-Divided Pattern* (since this level has two locution types : one locution type is the *Recursive Termination* and one locution type is *Divided Locution*).
- (9) Applies the *Recursive Termination-Divided Pattern*.

Bridging the Specification Protocol Gap in Argumentation

- (10) Selects and Applies the *Rekurs-To-N-Dialogue Pattern* to connect N-agents' dialogue with two agents' protocol.
- (11) Finally, the tool follows the steps of the automated synthesis process of two agents' protocol to generate the LCC protocol for DID for two agents. Note that the tool adds two lines after each *Termination Locution* (message) in the LCC protocol for two agents to connect N-agents' protocol with two agents' protocol (See Figure C.8(d), Figure C.8(e) and Figure C.8(f) in appendix C).

Appendix C

Persuasion Dialogue

This appendix presents a detailed description of how to transform a DID of a persuasion dialogue [Parkken, 2000] to an LCC protocol by using LCC-Argument patterns. It also presents a detailed example of the CPN model, the State Space and the Verification Model Properties of a CPN persuasion dialogue model. We open this appendix with a detail example which illustrates how the agent protocol automated synthesis tool "*GenerateLCCProtocol*" works to build a persuasion dialogue protocol between two agents in Section C.1. Section C.2 represents a detail example which illustrates how the agent protocol automated synthesis tool "*GenerateLCCProtocol*" works to build a persuasion dialogue protocol between N agents. Finally, Section C.3 represents the CPN model and the verification model properties of the persuasion dialogue.

C.1 An Example of an LCC Protocol begin generated for Two Agents

This section represents a detailed description of how to transform a DID of a persuasion dialogue, which is shown in Figure 4.3, to an LCC protocol by using LCC-Argument patterns. The final LCC protocol is illustrated in Figures C.1(a) and C.1(b). Below we explain the algorithm followed by the tool:

- (1) Begins with the locution icon at the top of the DID of the persuasion dialogue, which is *claim(T)*.
- (2) Selects the *Starting Pattern* (since the locution type is the *Starting Locution*).
- (3) Applies the *Starting Pattern* by matching formal parameters in the *Starting Pattern* with its corresponding values in the *claim(T)* icon, starting from the top-down and moving left to right (See Figure C.2(a)):

- a) Starting from the top of the locution icon, the tool matches SL with $claim(T)$.
 - b) Moving to the left side of the locution icon, the tool matches RP_1 with $claimSender_{P_1}$, role parameters with $(KB_P, CS_P, CS_O, T, ID_O)$, and role id with ID_P .
 - c) Moving to the right side of the locution icon, the tool matches RO_1 with $claimReceiver_{O_1}$, role parameters with (KB_O, CS_O, CS_P, ID_P) , and role id with ID_O .
 - d) Moving to the left side conditions, the tool matches CI with $addTopicToCS(T, CS_P)$.
 - e) Moving to the next level (See Figure C.2(b)), because the *Starting Pattern* has recursive roles, the sender agent will become the receiver and vice versa in the next level. The tool matches agent P recursive role with the right side of the locution icon. It matches RP_2 with $replyToClaimReceiver_P$, role parameters with $(KB_P, CS_P, CS_O, T, ID_O)$, and role id with ID_P . Then, the tool matches agent O recursive role with the left side of the locution icon. It matches RO_2 with $replyToClaimSender_O$, role parameters with $(KB_O, CS_O, CS_P, T, ID_P)$, and role id with ID_O .
- (4) Note that the next level in this example (level two of the DID of the persuasion dialogue) contains two locution icons: $why(T)$, which is located in the left of the DID, and $concede(T)$, which is located in the right. The tool starts from the locution in the left of the persuasion dialogue, which is $why(T)$.
- (5) Following this, the tool selects the *Termination-Intermediate Pattern* (since locution type is *Intermediate Locution*).
- (6) Since the selected *Termination-Intermediate Pattern* has rewriting methods, the tool selects two rewriting methods (one for $why(T)$ and one for $concede(T)$). It is important to note in this example that level two has: (1) one *Intermediate Locution* ($why(T)$) and the tool selects the *rewrite method 1* of

one intermediate locution; (2) one *Termination Locution* ($concede(T)$) and the tool selects the *rewrite method 1* of one termination locution. See Figure C.3(a).

(7) Applies the *Termination-Intermediate Pattern* by matching formal parameters in the selected rewriting methods of the *Termination-Intermediate Pattern* with its corresponding values in the $why(T)$ icon (on the left side of the DID), starting from the top-down and moving left to right (See Figure C.3(b)):

- a) Starting from the top of the locution icon, the tool matches IL with $why(T)$.
- b) Moving to the left side of the locution icon, the tool matches $R_{Sender1}$ with $replyToClaimSender_O$, role parameters with $(KB_O, CS_O, CS_P, T, ID_P)$, and role id with ID_O .
- c) Moving to the right side of the locution icon, the tool matches $R_{Receiver1}$ with $replyToClaimReceiver_P$, role parameters with $(KB_P, CS_P, CS_O, T, ID_O)$, and role id with ID_P .
- d) Moving to the left side conditions, the tool matches $C2$ with $(notFindTopicInKB(T, KB_O) \text{ and } notFindTopicInCS(T, CS_O))$. Note that in this example $C4$ equals null because no condition is connected to the right side of the locution.
- e) Moving to the next level, because the *Termination-Intermediate Pattern* has recursive roles, the sender agent will become the receiver and vice versa in the next level. The tool matches agent P recursive role with the left side of the locution icon. It matches $R_{Sender2}$ with $replyToWhySender_P$, role parameters with $(KB_P, CS_P, CS_O, T, ID_O)$, and role id with ID_P . The tool then matches agent O recursive role with the right side of the locution icon. It matches $R_{Receiver2}$ with $replyToWhyReceiver_O$, role parameters with $(KB_O, CS_O, CS_P, T, ID_P)$, and role id with ID_O . (See Figure C.3(c))

(8) Moves right to the $concede(T)$ locution. It applies the *Termination-Intermediate Pattern* by matching formal parameters in the selected rewriting methods of the

Termination-Intermediate Pattern with its corresponding values in the *concede(T)* icon (on the right side of the DID), starting from the top-down and moving left to right (See Figure C.3(d)):

- a) Starting from the top of the locution icon, the tool matches TL with *concede(T)*.
 - b) Moving to the left side of the locution icon, the tool matches $R_{SenderI}$ with *replyToClaimSender_O*, role parameters with $(KB_O, CS_O, CS_P, T, ID_P)$, and role id with ID_O .
 - c) Moving to the right side of the locution icon, the tool matches $R_{ReceiverI}$ with *replyToClaimReceiver_P*, role parameters with $(KB_P, CS_P, CS_O, T, KID_O)$, and role id with ID_P .
 - d) Moving to the left side conditions, the tool matches CI with *(findTopicInKB(T, KB_O) and notFindTopicInCS(T, CS_O) and notFindOppTopicInCS(not(T), CS_O) and addTopicToCS (T, CS_O))*. Note that in this example C3 equals null because no condition is connected to the right side of the locution.
- (9) Moves to the next level in the DID and repeats steps 4 and 8. Note that the automated synthesis process finishes when the tool matches level four in the DID (in Figure 4.3) with one of the LCC-Argument patterns.

Bridging the Specification Protocol Gap in Argumentation

Agent P	Agent O
<p>a(claimSender_P(KB_P,CS_P,CS_O,T,ID_O),ID_P) ::=</p> <p>claim(T) => a(claimReceiver_O(KB_O,CS_O, CS_P,ID_P),ID_O) ← addTopicToCS(T,CS_P)</p> <p>then</p> <p>a(replyToClaimReceiver_P(KB_P,CS_P,CS_O,T,ID_O), ID_P).</p>	<p>a(claimReceiver_O(KB_O,CS_O,CS_P,ID_P),ID_O) ::=</p> <p>claim(T) <= a(claimSender_P(KB_P,CS_P,CS_O,T,ID_P),ID_P)</p> <p>then</p> <p>a(replyToClaimSender_O(KB_O,CS_O,CS_P,T,ID_P),ID_O).</p>
<p>a(replyToClaimReceiver_P(KB_P,CS_P,CS_O, T,ID_O), ID_P) ::=</p> <p>concede(T) <= a(replyToClaimSender_O(KB_O,CS_O, CS_P,T,ID_P),ID_O)</p> <p>or</p> <p>why(T) <= a(replyToClaimSender_O (KB_O,CS_O, CS_P,T,ID_P),ID_O)</p> <p>then</p> <p>a(replyToWhySender_P (KB_P,CS_P,CS_O, T,ID_O),ID_P).</p>	<p>a(replyToClaimSender_O(KB_O,CS_O, ,CS_P T,ID_P),ID_O) ::=</p> <p>concede(T) => a(replyToClaimReceiver_P (KB_P,CS_P,CS_O, T,ID_O),ID_P) ←(findTopicInKB(T,KB_O) and notFindTopicInCS(T,CS_O) and notFindOppTopicInCS(not(T),CS_O) and addTopicToCS (T,CS_O))</p> <p>or</p> <p>why(T) => a(replyToClaimReceiver_P (KB_P,CS_P, CS_O, T,ID_O),ID_P) ← (notFindTopicInKB(T,KB_O) and notFindTopicInCS(T,CS_O))</p> <p>then</p> <p>a(replyToWhyReceiver_O(KB_O,CS_O,CS_P,T,ID_P),ID_O) .</p>
<p>a(replyToWhySender_P (KB_P,CS_P, CS_O,T,ID_O), ID_P) ::=</p> <p>retract(T) => a(replyToWhyReceiver_O (KB_O,CS_O, CS_P, T,ID_P),ID_O) ← (notFindPreInKB(T, KB_P) and findTopicInCS (T, CS_P) and subtractFromCS(T, CS_P))</p> <p>or</p> <p>(argue(Pre,T) => a(replyToWhyReceiver_O (KB_O,CS_O, CS_P, T,ID_P),ID_O) ←(Pre= findPremise (T,KB_P, CS_P) and addPreToCS(T,Pre,CS_P)) then</p> <p>a(replyToArgueReceiver_P (KB_P,CS_P, CS_O, T,Pre,ID_O), ID_P)).</p>	<p>a(replyToWhyReceiver_O (KB_O,CS_O, CS_P,T,ID_P),ID_O) ::=</p> <p>retract(T) <= a(replyToWhySender_P (KB_P,CS_P, CS_O, T,ID_O),ID_P)</p> <p>or</p> <p>(argue(Pre,T) <= a(replyToWhySender_P(KB_P,CS_P, CS_O, T,ID_O),ID_P) then</p> <p>a(replyToArgueSender_O (KB_O,CS_O, CS_P, T,Pre,ID_P),ID_O)).</p>

Figure C.1(a): Generated LCC Protocol for Persuasion Dialogue (Part 1)

Agent P	Agent O
<pre> a(replyToArgueReceiver_P(KB_P,CS_P, CS_O,T,Pre,ID_O),ID_P) ::= concede(T) <= a(ReplyToArgueSender_O (KB_O,CS_O, CS_P, T,Pre,ID_P), ID_O) or (argue(Def,T') <= a(replyToArgueSender_O (KB_O,CS_O, CS_P, T,Pre,ID_P), ID_O)) then a(replyToArgueSender_P (KB_P,CS_P, CS_O, T,Pre,Def,ID_O), ID_P)) or (why(Pre) <= a(replyToArgueSender_O (KB_O,CS_O, CS_P, T,Pre,ID_P), ID_O)) then a(replyToWhySender_P (KB_P,CS_P, CS_O, T,Pre,ID_O),ID_P)). </pre>	<pre> a(replyToArgueSender_O(KB_O,CS_O, CS_P, T,Pre,ID_P), ID_O) ::= concede(T) => a(replyToArgueReceiver_P (KB_P,CS_P, CS_O, T,Pre,ID_O),ID_P) ←(findPreInKB(Pre, KB_O) and notFindPreInCS(Pre, CS_O) and notFindOppPreInCS(not(Pre), CS_O) and addPreToCS(T,Pre, CS_O)) or (argue(Def,T') => a(replyToArgueReceiver_P (KB_P,CS_P, CS_O,T,Pre,ID_O),ID_P) ← (Def=findDefeats(T,Pre,KB_O, CS_O) and addDefeatToCS(Def, CS_O))) then a(replyToArgueReceiver_O (KB_O,CS_O, CS_P, T,Pre,Def,ID_P), ID_O)) or (why(Pre) => a(replyToArgueReceiver_P (KB_P,CS_P, CS_O,T,Pre,ID_O),ID_P) ←(notFindPreInKB(Pre,KB_O) and notFindPreInCS(Pre,CS_O))) then a(replyToWhyReceiver_O (KB_O,CS_O, CS_P, T,Pre,ID_P),ID_O)). </pre>

Figure C.1(b): Generated LCC Protocol for Persuasion Dialogue (Part 2)

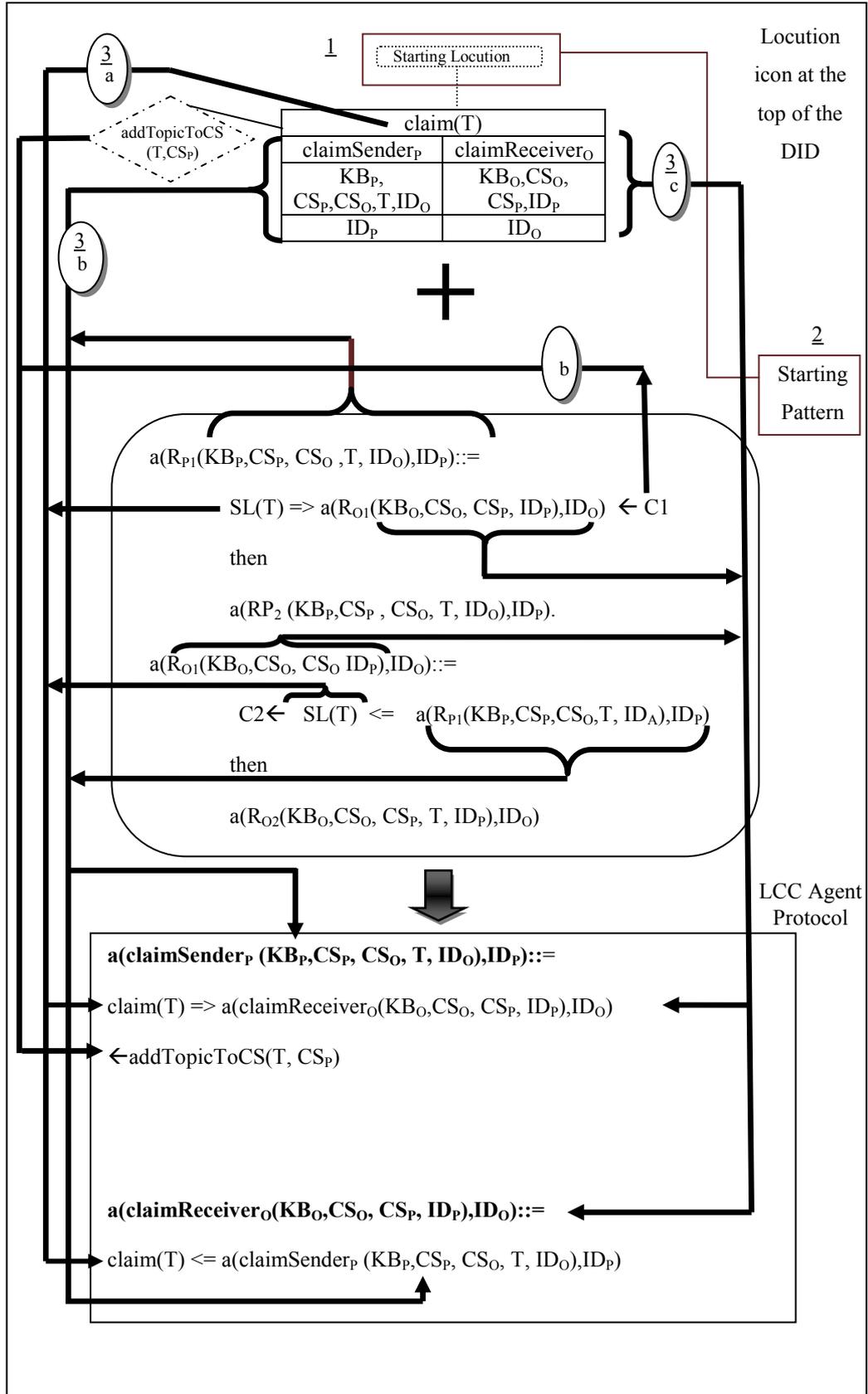


Figure C.2 (a): Step 3 of Protocol Generation (Matching the Starting Pattern)

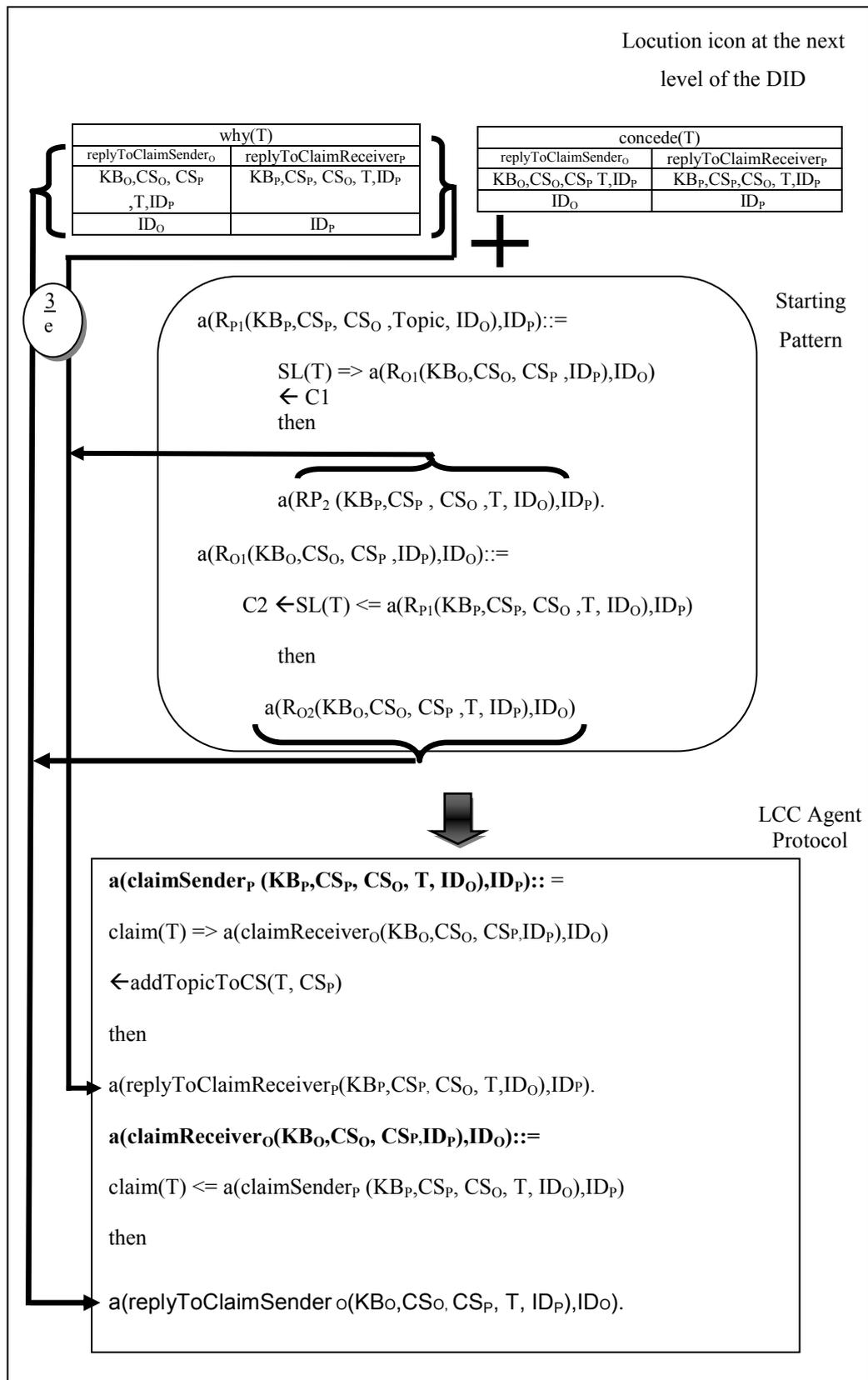


Figure C.2 (b): Step 3 of Protocol Generation (Completing the Recursive Roles)

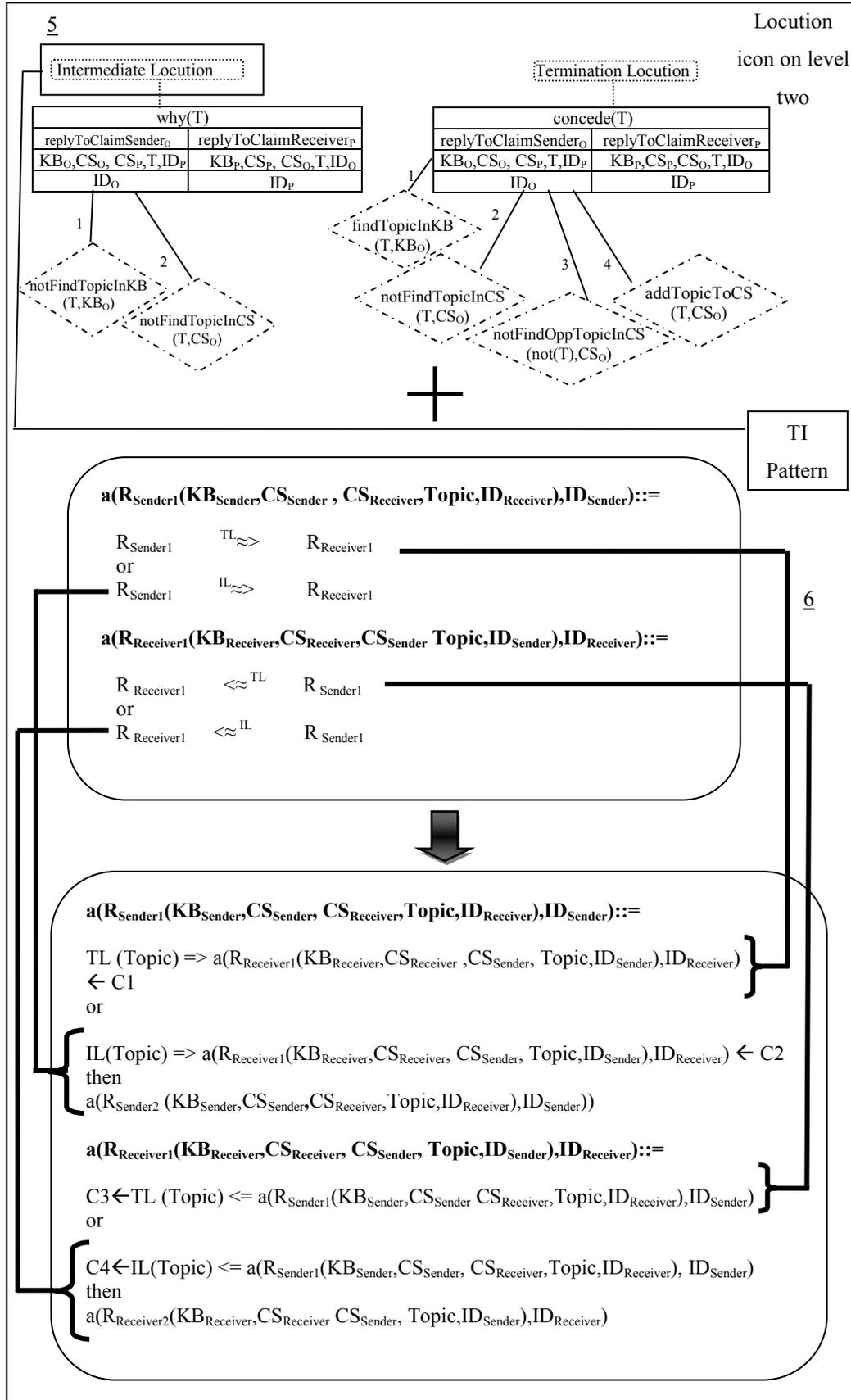


Figure C.3 (a): Step 5 and 6 of Protocol Generation

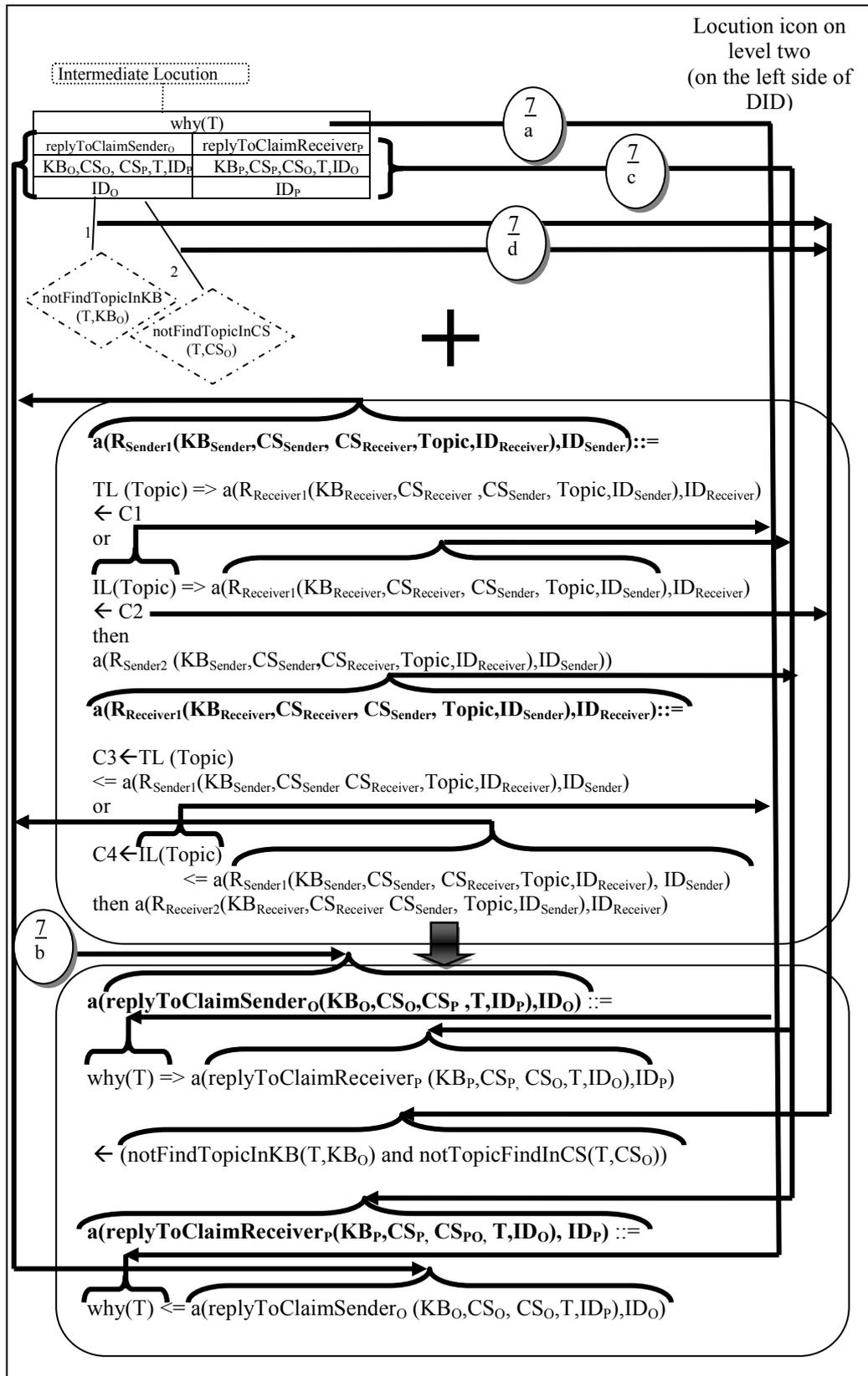


Figure C.3 (b): Step 7 of Protocol Generation (Matching the Termination-Intermediate Pattern)

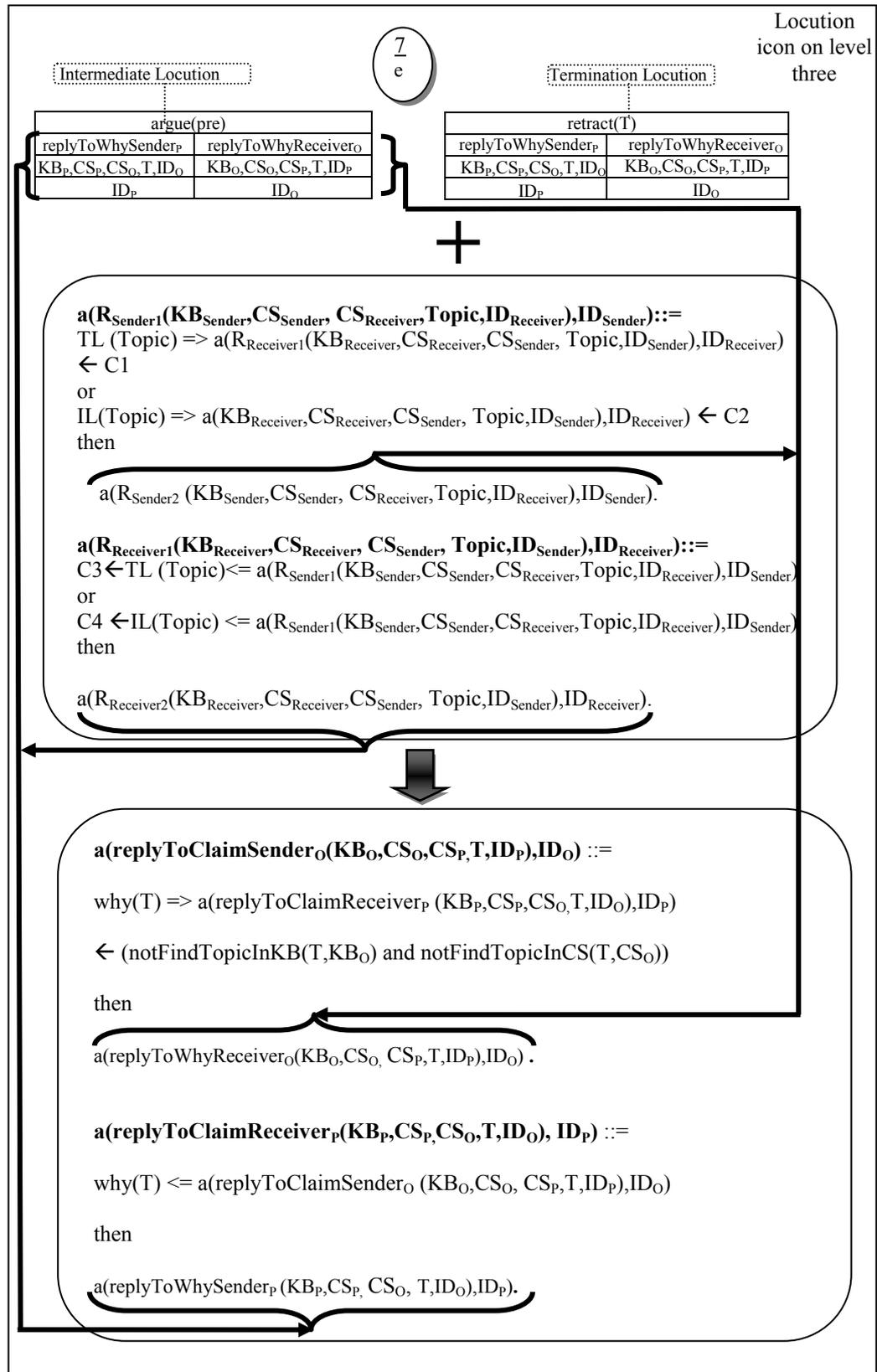


Figure C.3 (c): Step 7 of Protocol Generation (Complete the Recursive Roles)

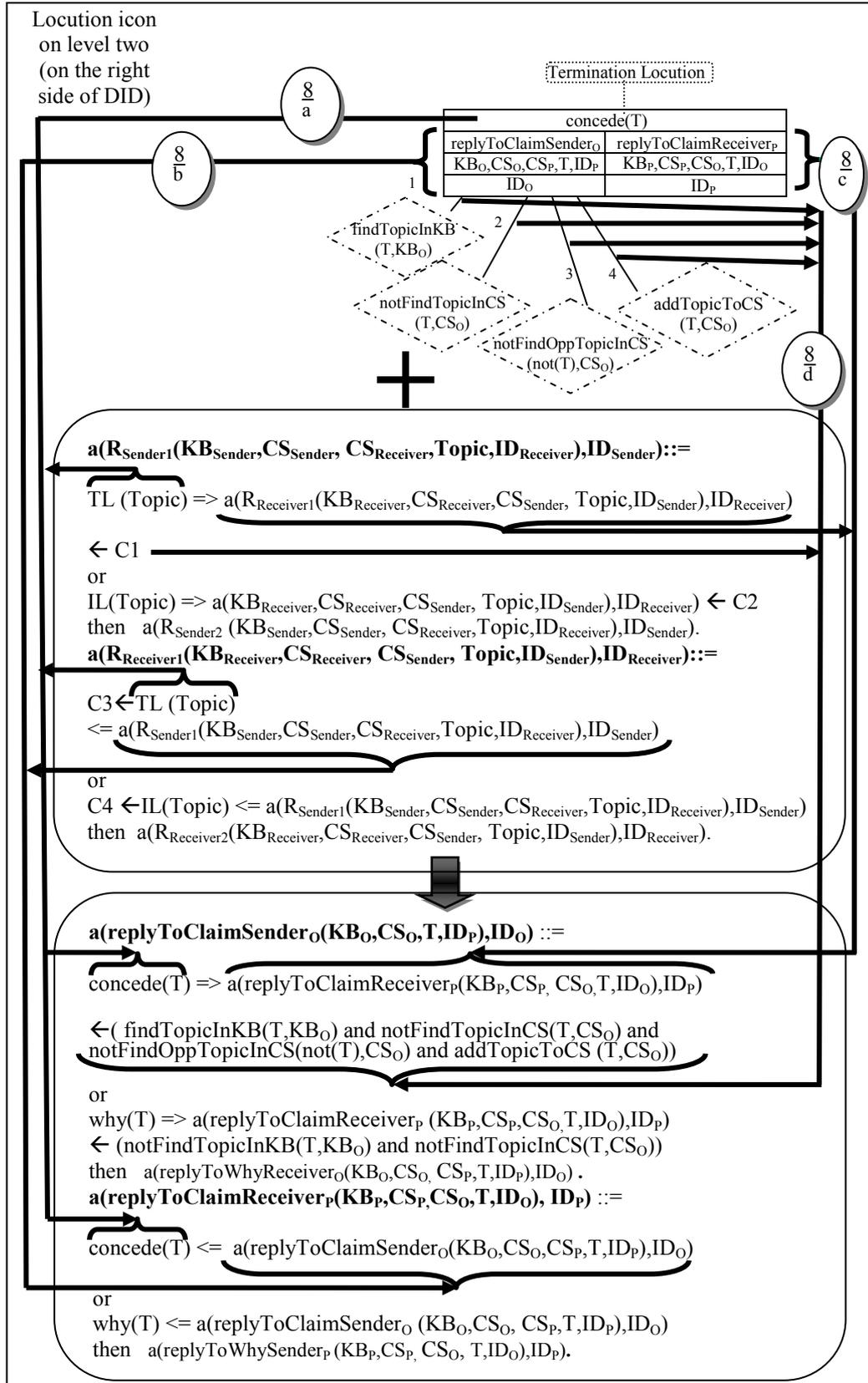


Figure C.3 (d): Step 8 of Protocol Generation (Matching the Rewriting Methods of the Termination-Intermediate Pattern)

C.2 An Example of LCC Protocol begin Generated for N-agent

In this section, we will give a detailed description of how to generate the LCC protocol of the persuasion dialogue between N-agent by using the black box of DID for N-agent (see chapter 4, section 4.4.5), LCC-Argument patterns and DID for two agents (the DID for two agents is shown in Figure 4.3). The final LCC protocol is illustrated in Figures C.8(a), C.8(b), C.8(c), C.8(d), C.8(e) and C.8(f):

(1) Begins with the Broadcasting Pattern. The tool uses the default functions of the *TerminationC1*, *DivivdeC2*, *AgentGroupC3*, and *RekursC4* conditions (See chapter 5 for more detail).

- *TerminationC1* = *greaterThanOrEequal(NAccepting, NSupporters)*
- *DivideC2* = *lessThan(NAccepting, NSupporters)* and *isNotEmpty(RejectionList)* and *isNotEmpty(AcceptingList)*
- *AgentGroupC3* = *creatOneAgentGroup*
(*RejectingList, Re, AcceptinList, Ac, AgentGroup, AGroup, P, O*)
- *RekursC4* = *isListEmpty(Re)* or *isListEmpty(Ac)*

(2) The tool then selects the *Move-To-Dialogue* Pattern and applies this pattern twice (to generate one role for *P* agent and one role for *O* agent) by matching formal parameters in the *Move-To-Dialogue Pattern* with their corresponding values in the *claim(T)* icon (the Starting locution icon in the DID of the persuasion dialogue for two agents):

- Agent *P* role:
 - a) Starting from the left side of the locution icon, the tool matches $R_{\text{Sender}1}$ with *claimSender_{P1}*.
 - b) Moving to the right side of the locution icon, the tool matches $R_{\text{Receiver}1}$ with *claimReceiver_{P1}*.

- c) The tool matches CI with its default functions ($addTopicToCS(T, CS_P)$). Note that in this example $C2$ equal null because no condition is connected to the right side of the locution.
- d) The tool matches roles parameters with $(KB_P, CS_P, CS_O, T, ID_{Proposal}, ID_O)$, and role id with ID_P . Note that the tool add $ID_{Proposal}$ and T to the role parameters (See Figure C.9(a) and Figure C.8(c)).

- Agent O role:

- a) Starting from the left side of the locution icon, the tool matches $R_{Sender1}$ with $claimSender_{O1}$.
- b) Moving to the right side of the locution icon, the tool matches $R_{Receiver1}$ with $claimReceiver_{O1}$.
- c) The tool matches CI with its default functions ($addTopicToCS(T, CS_O)$). Note that in this example $C2$ equals null because no condition is connected to the right side of the locution.
- d) The tool matches roles parameters with $(KB_O, CS_O, CS_P, T, ID_{Proposal}, ID_P)$, ID_O), and role id with ID_O . Note that the tool adds $ID_{Proposal}$ and T to the role parameters (See Figure C.9(b) and Figure C.8(c)).

(3) After that, the tool selects the Recurs-To-N-Dialogue Pattern:

- a) Since the selected Recurs-To-N-Dialogue Pattern has rewriting methods, the tool selects the Rewrite 2 (multiple end locution) rewriting methods and repeats this method twice because the Termination locution icons occurs three times in the DID of persuasion dialogue for two agents.
- b) The tool applies this pattern by matching formal parameters (variables) with their corresponding values in the Termination locution icons in the DID for two agents. As a result it generates one LCC role for the proposal agent (See in Figure C.8(c) the LCC role in the left side):

- i. Starting from the first Termination locution icon in the DID (See chapter 4, Figure 4.3) *concede(T)* on level two (See Figure C.9(c)):
 - Starting from the left side of the locution icon, the tool matches $R_{Sender1}$ with *claimSender_{P1}*, role parameters with $(KB_P, CS_P, CS_O, T, ID_{Proposal}, ID_O)$, and role id with ID_P . Note that the tool adds $ID_{Proposal}$ to the role parameters.
 - Moving to the right side of the locution icon, the tool matches $R_{Receiver1}$ with *claimReceiver_{O1}*, role parameters with $(KB_O, CS_O, CS_P, T, ID_{Proposal}, ID_P)$, and role id with ID_O . Note that the tool adds $ID_{Proposal}$ to the role parameters.

- ii. Starting from the second Termination locution icon in the DID(See chapter 4, Figure 4.3) *retract(T)* on level three (See Figure C.9(d)):
 - Moving to the left side of the locution icon, the tool matches $R_{Sender2}$ with *replyToWhySender_P*, role parameters with $(KB_P, CS_P, CS_O, T, ID_{Proposal}, ID_O)$, and role id with ID_P . Note that the tool adds $ID_{Proposal}$ to the role parameters.
 - Moving to the right side of the locution icon, the tool matches $R_{Receiver2}$ with *replyToWhyReceiver_O*, role parameters with $(KB_O, CS_O, CS_P, T, ID_{Proposal}, ID_P)$, and role id with ID_O . Note that the tool adds $ID_{Proposal}$ to the role parameters.

- iii. Starting from the third Termination locution icon in the DID(See chapter 4, Figure 4.3) *concede(T)* on level four (See Figure C.9(e)):
 - Moving to the left side of the locution icon, the tool matches $R_{Sender3}$ with *replyToArgueSende_O*, role parameters with $(KB_O, KB_O, CS_O, CS_P, T, Pre, ID_{Proposal}, ID_P)$, and role id with ID_O . Note that the tool adds $ID_{Proposal}$ to the role parameters.

- Moving to the right side of the locution icon, the tool matches $R_{Receiver3}$ with $replyToArgueReceiver_P$, role parameters with $(KB_P, CS_P, CS_O, T, Pre, ID_{Proposal}, ID_O)$, and role id with ID_P . Note that the tool adds $ID_{Proposal}$ to the role parameters.

- (4) The tool applies the automated synthesis process of the two agents' protocol to the generate persuasion dialogue LCC protocol for two agents (see section C.1).
- (5) The tool adds the "sending end message line" and "changing agents' role line" after each Termination message (locution) in the LCC protocol for two agents to connect the N-agents' protocol with the two agents' protocol. The final LCC protocol between two agents is illustrated in Figures C.8(d), C.8(e) and C.8(f).

C.3 Verification Model of the Persuasion Dialogue

In this section, we will give a detailed description of how to verify the semantics of the DID of a persuasion dialogue (shown in Figure 4.3) against the semantics of the synthesised LCC protocol (shown in Figures C.1(a) and C.1(b)). In this example, the *initial marking* is defined in the following way:

- (4) *OpenDialogue* place = "*The car is safe*". This place represents the dialogue topic.
- (5) *P* place = ("*P*", [], [("*The car is safe*", "*it has an airbag*")], "*claimSender*", "", "", [], "*O*"). This place represents the arguments of agent *P*.
- (6) *O* place = ("*O*", [], [("*it has an airbag*", "*The car is safe*")], "*claimReceiver*", "", "", [], "*P*"). This place represents arguments of agent *O*.

Proposal	Other Agents
<p>a(proposalSender_{proposal}(AgentList,NAgent,NSupporters,Topic),ID_{proposal}))::=</p> <p>proposal(Topic) => a(proposalReceiver_{ID}(KB_{ID},CS_{ID},ID_{proposal}), ID)</p> <p>←getAgentIDFromList (AgentList,otherAgents,ID) and addTopicToCS(Topic,CS_{proposal})</p> <p>then</p> <p>(</p> <p> a(replyToProposalReceiver_{proposal} (AgentList,NAgent,NSupporters,Topic,0,[],[],0,0), ID_{proposal}) ← agentListEmpty(AgentList)</p> <p> or</p> <p> a(proposalSender_{proposal} (OtherAgents,NAgent,NSupporters,Topic), ID_{proposal})</p> <p>).</p>	<p>a(proposalReceiver_{ID}(KB_{ID},CS_{ID},ID_{proposal}), ID)::=</p> <p>proposal(Topic)<=</p> <p>a(proposalSender_{proposal}(AgentList,NAgent,NSupporters,Topic), ID_{proposal})</p> <p>then</p> <p>a(replyToProposalSender(KB_{ID},CS_{ID},Topic,ID_{proposal}), ID).</p>
<p>a(replyToProposalReceiver_{proposal}(AgentList,NAgent,NSupporters,Topic,NReply,AcceptingList,RejectingList,NAccepting,NRejecting), ID_{proposal}))::=</p> <p>(</p> <p> addIDToList(SendingList,OtherSedingList,ID) and addToAcceptingList(AcceptingList,AccList,ID) and increaseAccepting(NAccepting,NAcc) and RejList= RejectionList and NRej is NRejection</p> <p> ← accept(Topic)</p> <p> <= a(replyToProposalSender_{ID}(KB_{ID},CS_{ID},Topic,ID_{proposal}), ID)</p> <p> or</p> <p> addToRejectingList(RejectingList,RejList,ID) and increaseRejecting(NRejecting,NRej) and increaseReply (NReply,NRep) and AccList=AcceptingList and NAcc is NAccepting</p> <p> ← reject(Topic) <=</p> <p> a(replyToProposalSender_{ID}(KB_{ID},CS_{ID},Topic,ID_{proposal}), ID)</p> <p>)</p> <p>then</p> <p>a(resultSender_{proposal} (AgentList,NAgent,NSupporters,Topic,NReply,AcceptingList, RejectionList,NAccepting,NRejection), ID_{proposal}) ← isEqual(NRep,NAgent).</p>	<p>a(replyToProposalSender_{ID}(KB_{ID},CS_{ID},Topic,ID_{proposal}), ID) ::=</p> <p>(</p> <p> accept(Topic) =></p> <p> a(replyToProposalReceiver_{proposal} (_ , _ , _ , _ , _ , _ , _ , _ , _ , _ , ID_{proposal}))</p> <p> ← findTopicInKB(Topic, KB_{ID}) and notFindTopicInCS (Topic,CS_{ID}) and notFindOppTopicInCS (not(Topic),CS_{ID}) and addTopicToCS(Topic,CS_{ID})</p> <p> or</p> <p> reject(Topic) =></p> <p> a(replyToProposalReceiver_{proposal} (_ , _ , _ , _ , _ , _ , _ , _ , _ , _ , ID_{proposal}))</p> <p> ← notFindTopicInKB(Topic,KB_{Proposal}) and notFindTopicInCS(Topic,CS_{Proposal})</p> <p>)</p> <p>then</p> <p>a(resultReceiver_{ID}(KB_{ID},CS_{ID},Topic,ID_{proposal}), ID) .</p>

Figure C.8(a): Generated LCC Protocol for Persuasion Dialogue (Part 1)

Proposal	Other agents
<pre> a(resultSender_{proposal}(AgentList,NAgent, NSupporters,Topic,AcceptingList,RejectionL ist, AgentGroup), ID_{proposal}) ::= a(sendReachAgreement_{proposal} (AgentList,NAgent,Topic),ID_{proposal}) ← greaterThanOrEqual(NAccepting, NSupporters) or a(divideGroup_{proposal} (AgentList , NAgent,NSupporters ,Topic,AcceptingList,RejectionList, []) ,ID_{proposal}) ← (lessThan(NAccepting ,NSupporters) and isEmpty(RejectionList) and isEmpty(AcceptingList)). a(sendReachAgreement_{proposal} (AgentList, Topic),ID_{proposal}) ::= reachAgreement(Topic) => a(resultReceiver_{ID}(KB_{ID},CS_{ID},ID_{proposal}), ID) ← getAgentIDFromList (AgentList,otherAgents,ID) then (null ← isEmpty(AgentList) or a(sendReachAgreement_{proposal} (OtherAgents, Topic), ID_{proposal})). a(divideGroup_{proposal} (AgentList, NAgent,NSupporters ,Topic, AcceptingList,RejectionList,AgentGroup), ID_{proposal}) ::= (argueWith (Topic,P,O) => a(resultReceiver_P (KB_P,CS_P, CS_O,Topic,ID_{proposal}), P) ← creatOneAgentGroup(Rejecting,Re,Accepting,Ac, AgentGroup, AGroup,P,O) then argueWith (Topic,O,P) => a(resultReceiver_O (KB_O,CS_O, CS_P, Topic,ID_{proposal}), O)) then (a(recurs_{proposal} (AgentList, NAgent,NSupporters ,0 ,Topic),ID_{proposal}) ← isEmpty(Re) or isEmpty(Ac) or a(divideGroup_{proposal}(AgentList ,NAgent,NSupporters,Topic,Ac,Re,AGroup), ID_{proposal})). </pre>	<pre> a(resultReceiver_P(KB_P,CS_P,CS_O,Topic, ID_{proposal}),P) ::= reachAgreement(Topic) <= a(sendReachAgreement_{proposal} (AgentList, Topic),ID_{proposal}) or (argueWith(Topic,P,O) <= a(divideGroup_{proposal} (AgentList,NAgent,NSupporters,Topic,Acce ptingList,RejectionList, AgentGroup),ID_{proposal}) then a(startDID (KB_P,CS_P, CS_O,Topic, ID_{proposal}, O),P)). </pre>

Figure C.8(b): Generated LCC Protocol for Persuasion Dialogue (Part 2)

Proposal	Other agents
<p> a(recurs_{Proposal} (AgentList, NAgent, NSupporters, replyN, Topic), ID_{Proposal}) ::= (N = replyN + 1 \leftarrow end(Topic) \leq a(replyToClaimSender_O (KB_O, CS_O, CS_P, T, ID_{Proposal}, ID_P), ID_O) or N = replyN + 1 \leftarrow end(Topic) \leq a(replyToClaimReceiver_P (KB_P, CS_P, CS_O, T, ID_{Proposal}, ID_O), ID_P) or N = replyN + 1 \leftarrow end(Topic) \leq a(replyToWhySender_P(KB_P, CS_P, CS_O, T, ID_{Proposal}, ID_O), ID_P) or N = replyN + 1 \leftarrow end(Topic) \leq a(replyToWhyReceiver_O (KB_O, CS_O, CS_P, T, ID_{Proposal}, ID_P), ID_O) or N = replyN + 1 \leftarrow end(Topic) \leq a(replyToArgueSender_O (KB_P, CS_P, CS_O, T, Pre, ID_{Proposal}, ID_O), ID_O) or N = replyN + 1 \leftarrow end(Topic) \leq a(replyToArgueReceiver_P (KB_O, CS_O, CS_P, T, Pre, ID_{Proposal}, ID_P), ID_P)) then (a(proposalSender_{proposal} (AgentList, NAgent, NSupporters, Topic), ID_{proposal}) \leftarrow isEqual(N, NAgent) or a(recurs_{Proposal} (AgentList, NAgent, NSupporters, N, Topic), ID_{Proposal})). </p>	<p> a(startDID_P(KB_P, CS_P, CS_O, T, ID_{Proposal}, ID_O), ID_P) ::= a(claimSender (KB_P, CS_P, CS_O, T, ID_{Proposal}, ID_O), ID_P) \leftarrow addTopicToCS(T, CS_P) or a(claimReceiver (KB_P, CS_P, CS_O, T, ID_{Proposal}, ID_O), ID_P). a(startDID_O(KB_O, CS_O, CS_P, T, ID_{Proposal}, ID_P), ID_O) ::= a(claimSender (KB_O, CS_O, CS_P, T, ID_{Proposal}, ID_P), ID_O) \leftarrow addTopicToCS(T, CS_P) or a(claimReceiver (KB_O, CS_O, CS_P, T, ID_{Proposal}, ID_P), ID_O). </p>

Figure C.8(c): Generated LCC Protocol for Persuasion Dialogue (Part 3)

Agent P	Agent O
<p>a(claimSender_P(KB_P,CS_P,CS_O,T,ID_O),ID_P) ::=</p> <p>claim(T) => a(claimReceiver_O(KB_O,CS_O,CS_P,ID_P),ID_O) ← addTopicToCS(T,CS_P)</p> <p>then</p> <p>a(replyToClaimReceiver_P(KB_P,CS_P,CS_O,T,ID_O),ID_P).</p>	<p>a(claimReceiver_O(KB_O,CS_O,CS_P,ID_P),ID_O) ::=</p> <p>claim(T) <= a(claimSender_P(KB_P,CS_P,CS_O,T,ID_O),ID_P)</p> <p>then</p> <p>a(replyToClaimSender_O(KB_O,CS_O,CS_P,T,ID_P),ID_O).</p>
<p>a(replyToClaimReceiver_P(KB_P,CS_P,CS_O,T,ID_O),ID_P) ::=</p> <p>(concede(T) <= a(replyToClaimSender_O(KB_O,CS_O,CS_P,T,ID_P),ID_O) then end(Topic)=> a(recurs_{Proposal}(AgentList,NAgent,NSupporters,NReply,Topic),ID_{Proposal}) then a(proposalReceiver_{ID}(KB_{ID},CS_{ID},ID_{proposal}),ID)) or (why(T) <= a(replyToClaimSender_O(KB_O,CS_O,CS_P,T,ID_P),ID_O) then a(replyToWhySender_P(KB_P,CS_P,CS_O,T,ID_O),ID_P)).</p>	<p>a(replyToClaimSender_O(KB_O,CS_O,CS_P,T,ID_P),ID_O) ::=</p> <p>(concede(T) => a(replyToClaimReceiver_P(KB_P,CS_P,CS_O,T,ID_O),ID_P) ← (findTopicInKB(T,KB_O) and notFindTopicInCS(T,CS_O) and notFindOppTopicInCS(not(T),CS_O) and addTopicToCS(T,CS_O)) then end(Topic)=> a(recurs_{Proposal}(AgentList,NAgent,NSupporters,NReply,Topic),ID_{Proposal}) then a(proposalReceiver_{ID}(KB_{ID},CS_{ID},ID_{proposal}),ID)) or (why(T) => a(replyToClaimReceiver_P(KB_P,CS_P,CS_O,T,ID_O),ID_P) ← (notFindTopicInKB(T,KB_O) and notFindTopicInCS(T,CS_O)) then a(replyToWhyReceiver_O(KB_O,CS_O,CS_P,T,ID_P),ID_O)).</p>

Figure C.8(d): Generated LCC Protocol for Persuasion Dialogue (Part 4)

Agent P	Agent O
<pre> a(replyToWhySender_P (KB_P,CS_P, CS_O,T,ID_O), ID_P) ::= (retract(T) => a(replyToWhyReceiver_O (KB_O,CS_O, CS_P, T,ID_P),ID_O) ← (notFindPreInKB(T, KB_P) and findTopicInCS (T, CS_P) and subtractFromCS(T, CS_P) then end(Topic)=> a(recurs_{Proposal} (AgentList,NAgent,NSupporters,NReply,Topic), ID_{Proposal}) then a(proposalReceiver_{ID} (KB_{ID},CS_{ID},ID_{proposal}), ID)) or (argue(Pre,T) => a(replyToWhyReceiver_O (KB_O,CS_O, CS_P, T,ID_P),ID_O) ←(Pre= findPremise (T,KB_P, CS_P) and addPreToCS(T,Pre,CS_P)) then a(replyToArgueReceiver_P (KB_P,CS_P, CS_O, T,Pre,ID_O), ID_P)). </pre>	<pre> a(replyToWhyReceiver_O (KB_O,CS_O, CS_P,T,ID_P),ID_O) ::= (retract(T) <= a(replyToWhySender_P (KB_P,CS_P, CS_O, T,ID_O),ID_P) then end(Topic)=> a(recurs_{Proposal} (AgentList,NAgent,NSupporters,NReply,Topic), ID_{Proposal}) then a(proposalReceiver_{ID} (KB_{ID},CS_{ID},ID_{proposal}), ID)) or (argue(Pre,T) <= a(replyToWhySender_P(KB_P,CS_P, CS_O, T,ID_O),ID_P) then a(replyToArgueSender_O (KB_O,CS_O, CS_P, T,Pre,ID_P),ID_O)). </pre>

Figure C.8(e): Generated LCC Protocol for Persuasion Dialogue (Part 5)

Step One: Automated Transformation from LCC to CPN/XML

The generated LCC protocol of the persuasion dialogue in Figures C.1(a) and C.1(b) was used as input to the verification tool. The verification tool generated a persuasion dialogue CPN/XML file which has:

- (1) The declaration of three colour sets (Topic, Message, Role) and thirteen functions. (see chapter 6 section 6.1.1)
- (2) Eight CPN subpages generated by the *GenerateLCCProtocol* tool (one subpage for each LCC role in the Figures C.1(a) and C.1(b)).

Agent P	Agent O
<pre> a(replyToArgueReceiver_P(KB_P,CS_P, CS_O,T,Pre,ID_O),ID_P) ::= (concede(T) <= a(ReplyToArgueSender_O (KB_O,CS_O, CS_P, T,Pre,ID_P), ID_O) then end(Topic)=> a(recurs_{Proposal} (AgentList,NAgent,NSupporters,NReply,Topic), ID_{Proposal}) then a(proposalReceiver_{ID} (KB_{ID},CS_{ID},ID_{proposal}), ID)) or (argue(Def,T) <= a(replyToArgueSender_O (KB_O,CS_O, CS_P, T,Pre,ID_P), ID_O) then a(replyToArgueSender_P (KB_P,CS_P, CS_O, T,Pre,Def,ID_O), ID_P)) or (why(Pre) <= a(replyToArgueSender_O (KB_O,CS_O, CS_P, T,Pre,ID_P), ID_O) then a(replyToWhySender_P (KB_P,CS_P, CS_O, T,Pre,ID_O),ID_P)). </pre>	<pre> a(replyToArgueSender_O(KB_O,CS_O, CS_P, T,Pre,ID_P), ID_O) ::= (concede(T) => a(replyToArgueReceiver_P (KB_P,CS_P, CS_O, T,Pre,ID_O),ID_P) ←(findPreInKB(Pre, KB_O) and notFindPreInCS(Pre, CS_O) and notFindOppPreInCS(not(Pre), CS_O) and addPreToCS(T,Pre, CS_O)) then end(Topic)=> a(recurs_{Proposal} (AgentList,NAgent,NSupporters,NReply,Topic), ID_{Proposal}) then a(proposalReceiver_{ID} (KB_{ID},CS_{ID},ID_{proposal}), ID)) or (argue(Def,T) => a(replyToArgueReceiver_P (KB_P,CS_P, CS_O,T,Pre,ID_O),ID_P) ← (Def =findDefeats(T,Pre,KB_O, CS_O) and addDefeatToCS(Def, CS_O)) then a(replyToArgueReceiver_O (KB_O,CS_O, CS_P, T,Pre,Def,ID_P), ID_O)) or (why(Pre) => a(replyToArgueReceiver_P (KB_P,CS_P, CS_O,T,Pre,ID_O),ID_P) ←(notFindPreInKB(Pre,KB_O) and notFindPreInCS(Pre,CS_O)) then a(replyToWhyReceiver_O (KB_O,CS_O, CS_P, T,Pre,ID_P),ID_O)). </pre>

Figure C.8(f): Generated LCC Protocol for Persuasion Dialogue (Part 6)

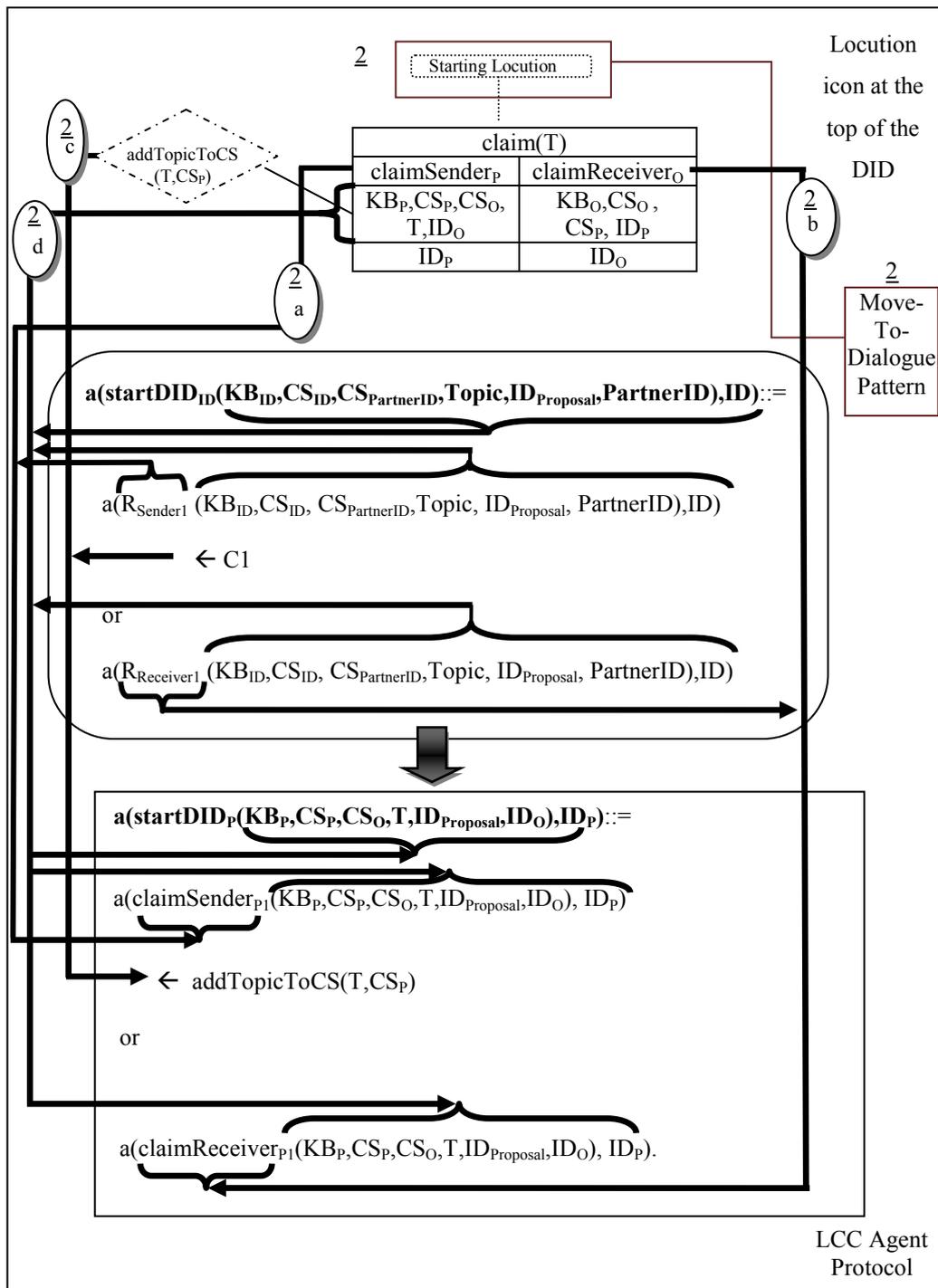


Figure C.9 (a): Step 2 of Protocol Generation (Matching the Move-To-Dialogue Pattern)

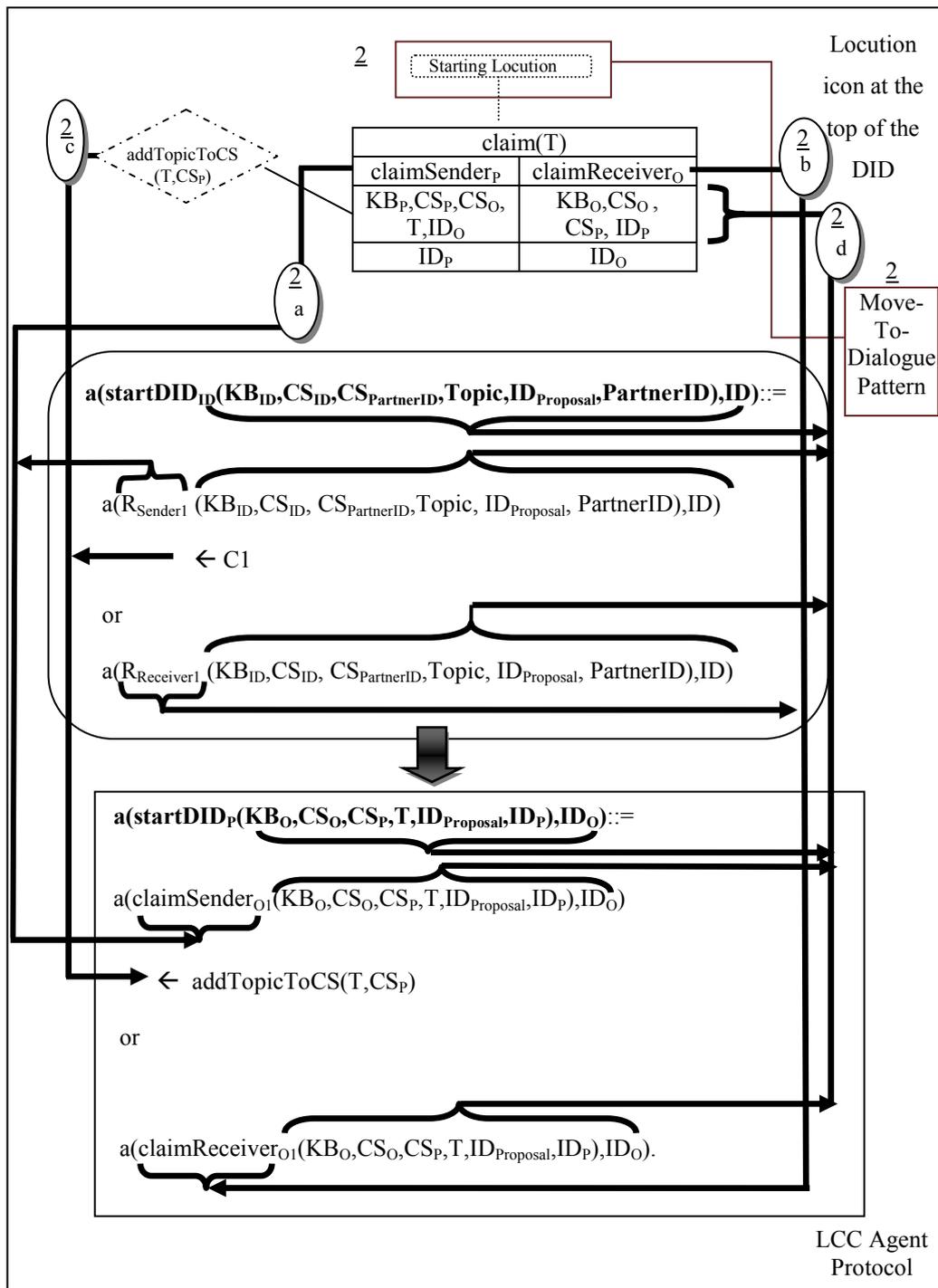


Figure C.9 (b): Step 2 of Protocol Generation (Matching the Move-To-Dialogue Pattern)

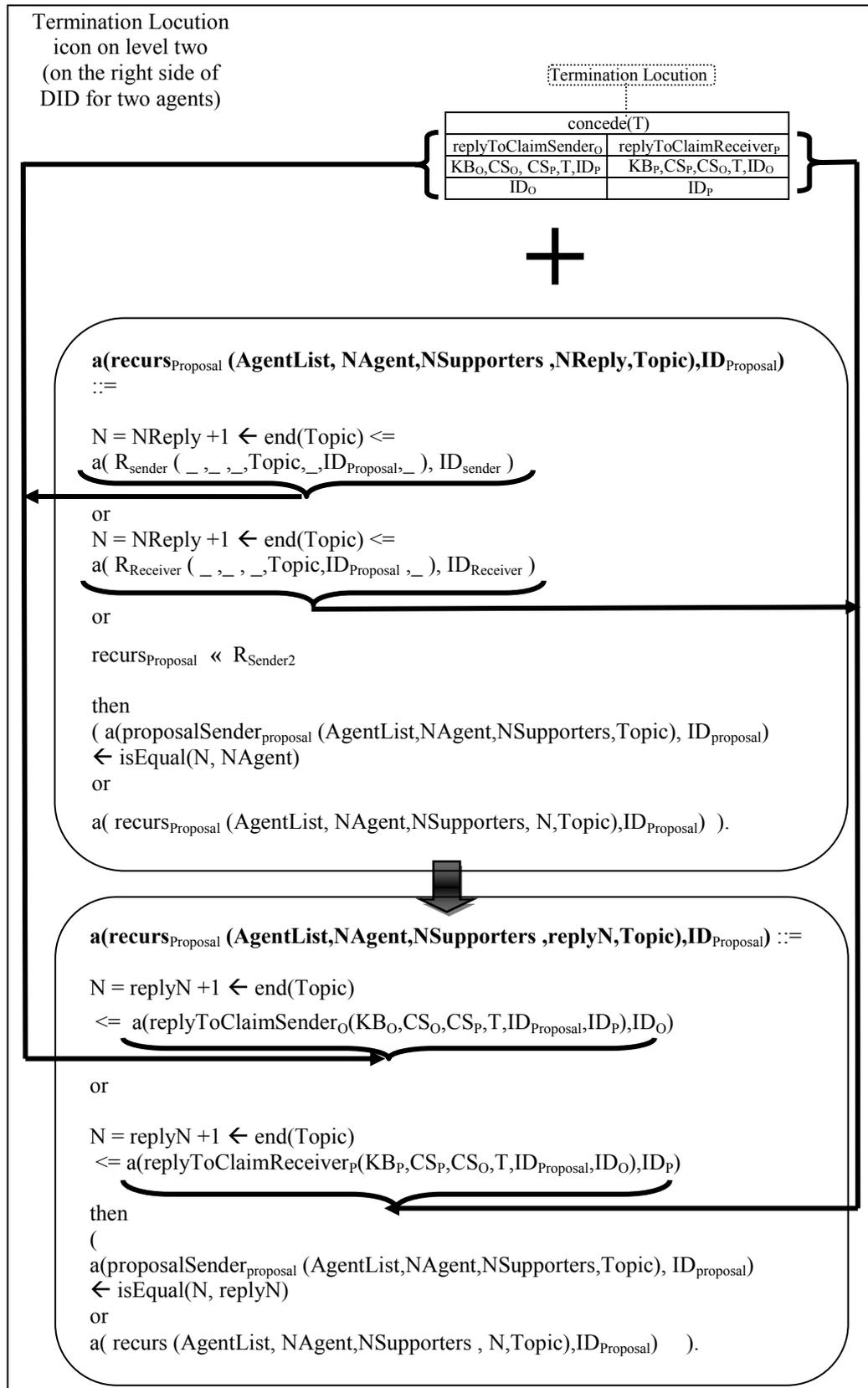


Figure C.9 (c): Step 3 (Part 1) of Protocol Generation (Matching the Rewriting Methods of the Recurs-To-N-Dialogue Pattern)

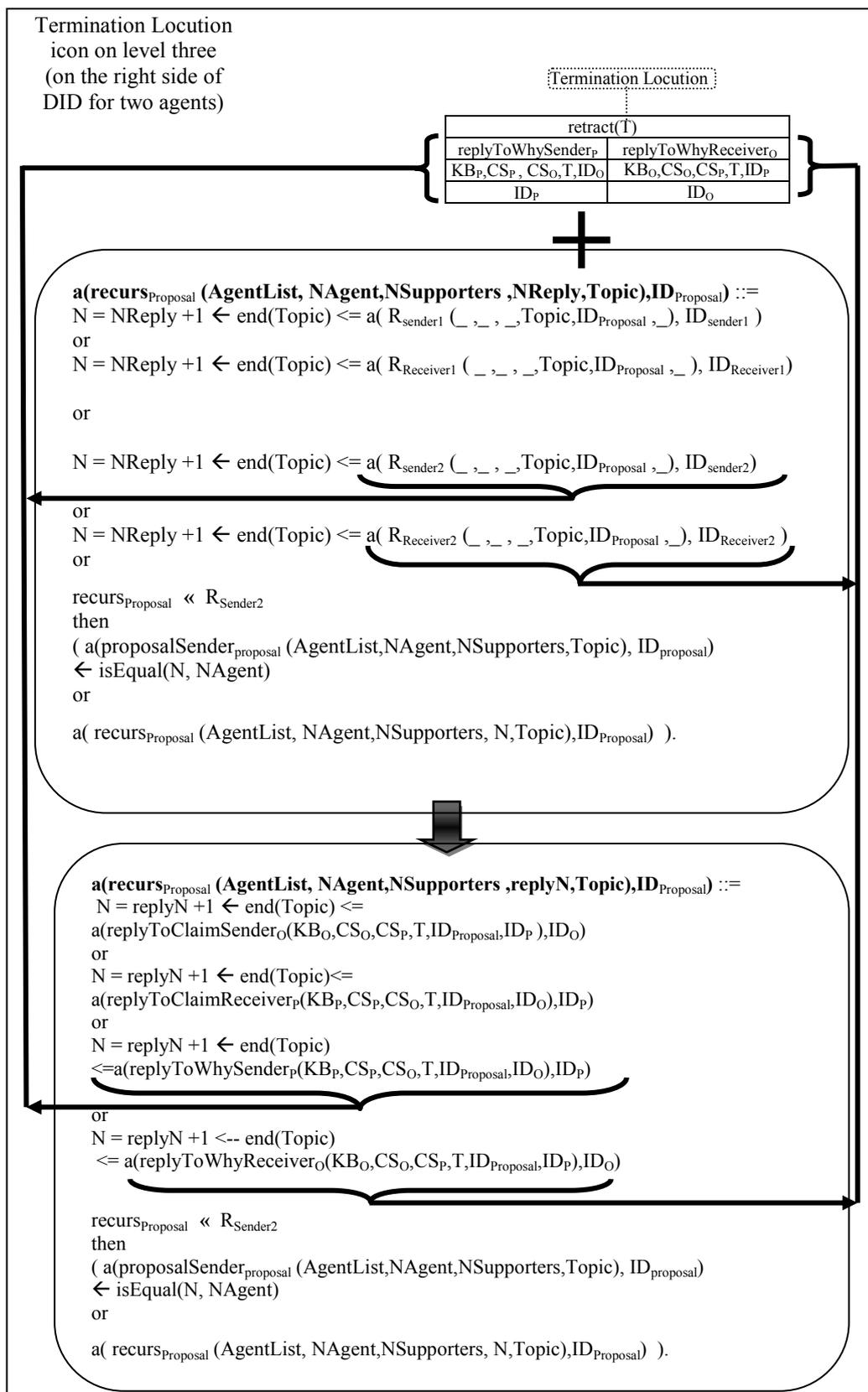


Figure C.9 (d): Step 3 (Part 2) of Protocol Generation (Matching the Rewriting Methods of the Recurs-To-N-Dialogue Pattern)

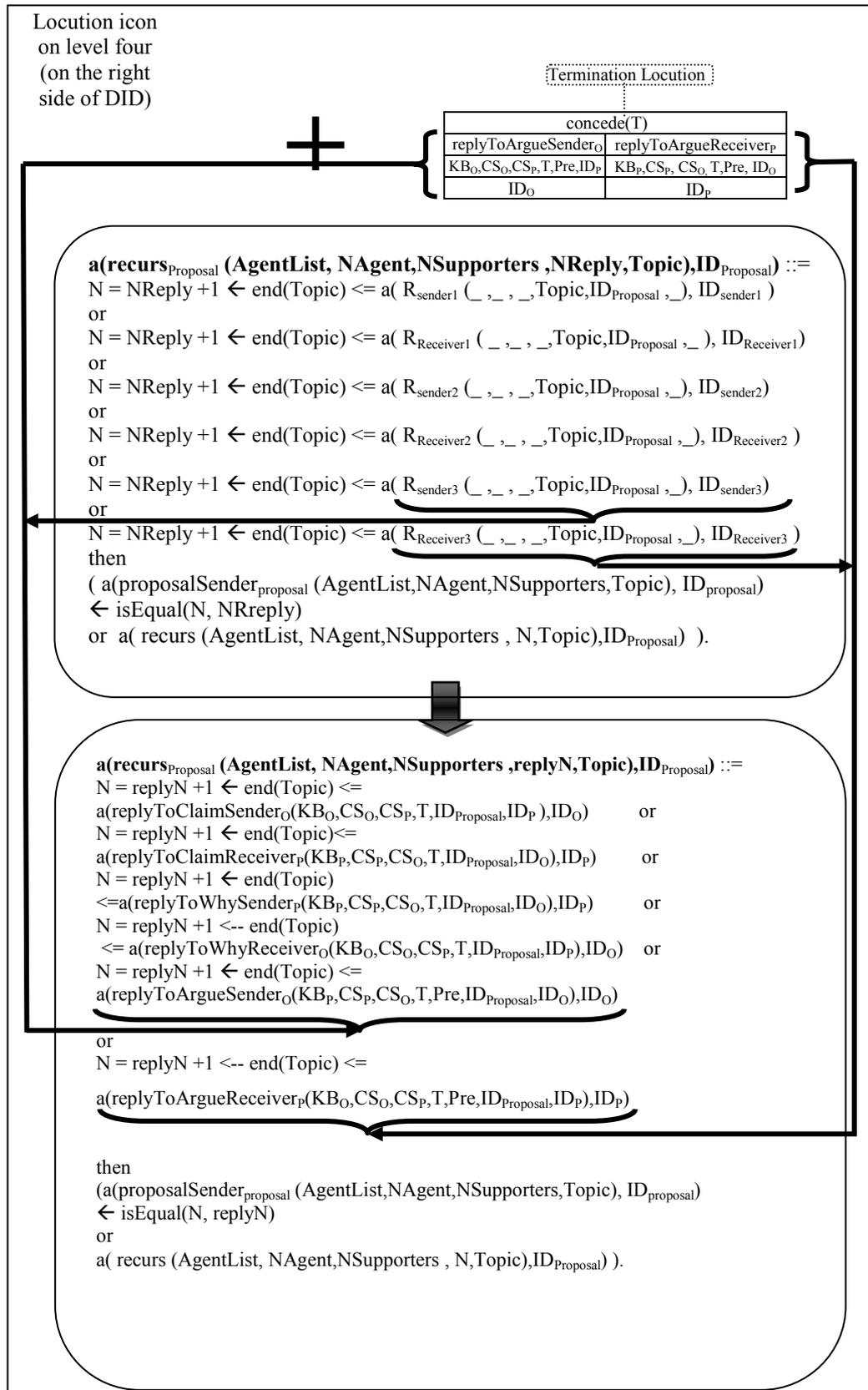


Figure C.9 (e): Step 3 (Part 3) of Protocol Generation (Matching the Rewriting Methods of the Recurs-To-N-Dialogue Pattern)

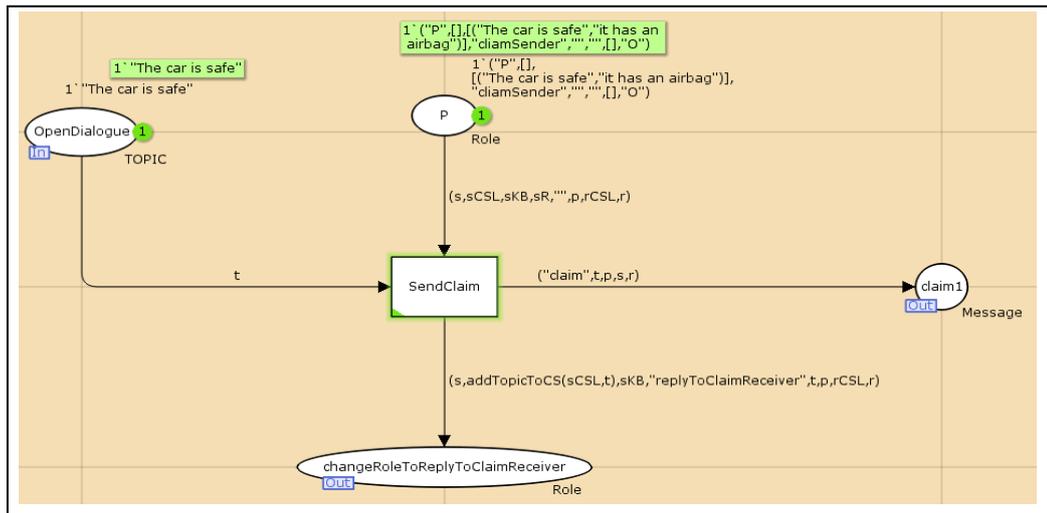


Figure C.10: The *claimSender_P* CPN Subpage

- Figure C.10 shows the *claimSender_P* role CPN subpage. This subpage has one input place *OpenDialogue* which represents the dialogue topic (In this example, the *initial marking* of this place = "The car is safe"). The place *P* represents the role arguments (In this example, the *initial marking* of this place is equal to ("P",[],[("The car is safe", "it has an airbag")], "claimSender", "", "", [],"O")). When the *SendClaim* transition occurs (when places *OpenDialogue* and *P* are active), *claimSender_P* role CPN subpage sends *claim* message using *claim1* output place and change its role to *ReplyToClaimSender* using *ChangeRoleToReplyToClaimSender* output place.
- Figure C.11 shows the *claimReceiver_O* role CPN subpage. In this page, the place *O* represents the role arguments (In this example, the *initial marking* of this place is equal to ("O",[], [("it has an airbag", "The car is safe")], "claimReceiver", "", "", [],"P")). This subpage receives the *claim* message using *claim1* input place. Then, when the *ReceiveClaim* transition occurs (when places *claim1* and *O* are active), it changes its role to *ReplyToClaimReceiver* using *ChangeRoleToReplyToClaimReceiver* output place.

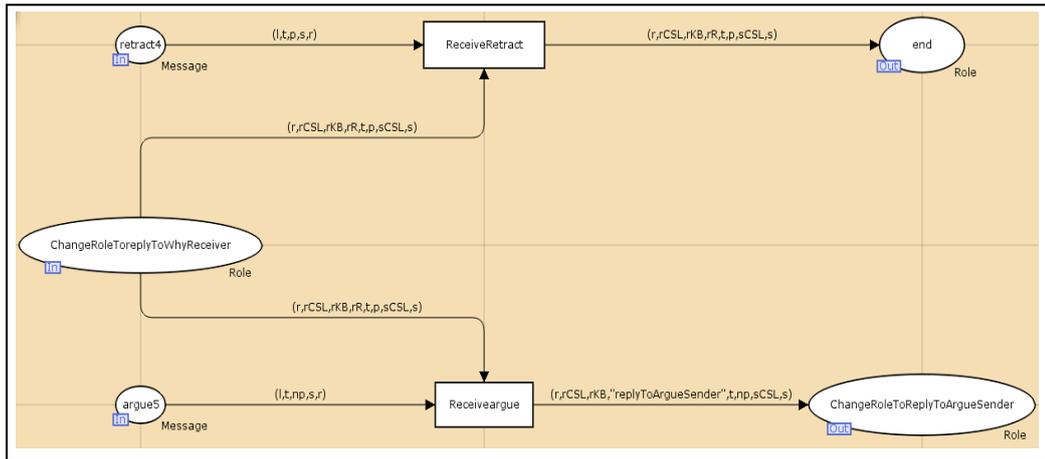


Figure C.15: The replyToWhyReceiver₀ CPN Subpage

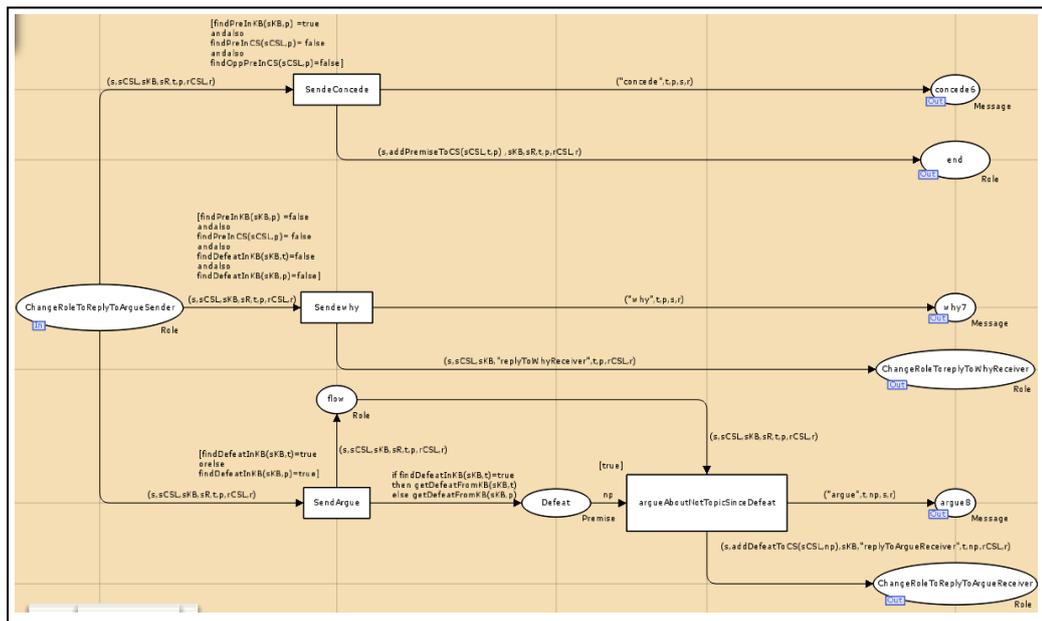


Figure C.16: The replyToArgueSender₀ CPN Subpage

output place; (2) sends *retract* message using *retract4* output place and then ends the dialogue using *end* output place.

- Figure C.15 shows the *replyToWhyReceiver₀* role CPN subpage. This subpage receives two messages (*argue* or *retract*) and generates responses depending on some conditions. If it receives the *retract* message using *retract4* input place, it responds by ending the dialogue using *end* output place. Otherwise, if it receives the *argue* message using *argue5* input place, it responds by changing its role to *ReplyToArgueSender* using *ChangeRoleToArgueSender*.

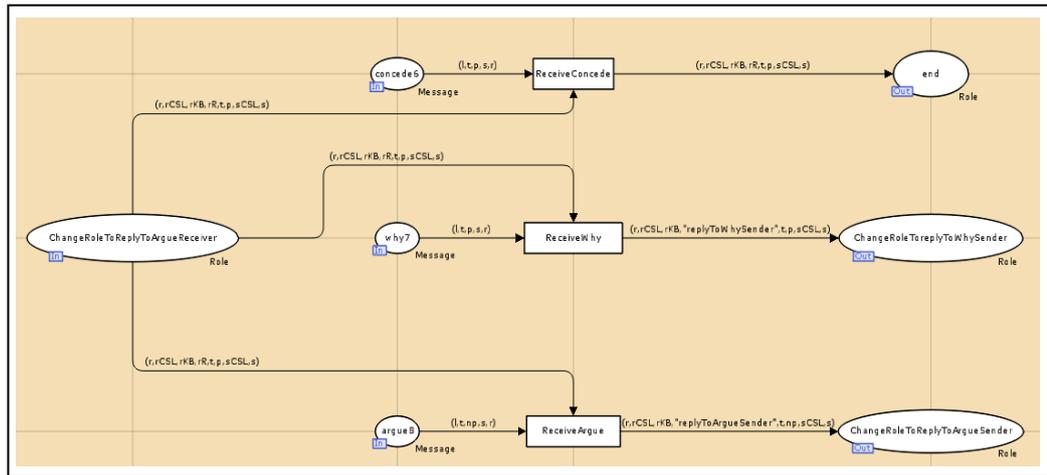


Figure C.17: The *replyToArgueReceiver_P* CPN Subpage

- Figure C.16 shows the *replyToArgueSender_O* role CPN subpage. This subpage sends three messages: (1) sends *concede* message using *concede6* output place and then ends the dialogue using *end* output place; (2) sends *why* message using *why7* output place and changes its role to *ReplyToWhyReceiver* using *ChangeRoleToWhyReceiver* output place; (3) sends *argue* message using *argue8* output place and changes its role to *ReplyToArgueReceiver* using *ChangeRoleToArgueReceiver* output place;
- Figure C.17 shows the *replyToArgueReceiver_P* role CPN subpage. This subpage receives three messages (*argue*, *why* or *concede*) and generates responses depending on some conditions. If it receives the *concede* message using *concede6* input place, it responds by ending the dialogue using *end* output place. If it receives the *why* message using *why7* input place, it responds by changing its role to *ReplyToWhySender* using *ChangeRoleToWhySender*. If it receives *argue* message using *argue8* input place, it responds by changing its role to *ReplyToArgueSender* using *ChangeRoleToArgueSender*.

(3) One CPN superpage generated by the *GenerateLCCProtocol* tool. This page connects the eight CPN subpages (*claimSender_P*, *claimReceiver_O*, *replyToclaimSender_O*, *replyToclaimReceiver_P*, *replyToWhySender_P*,

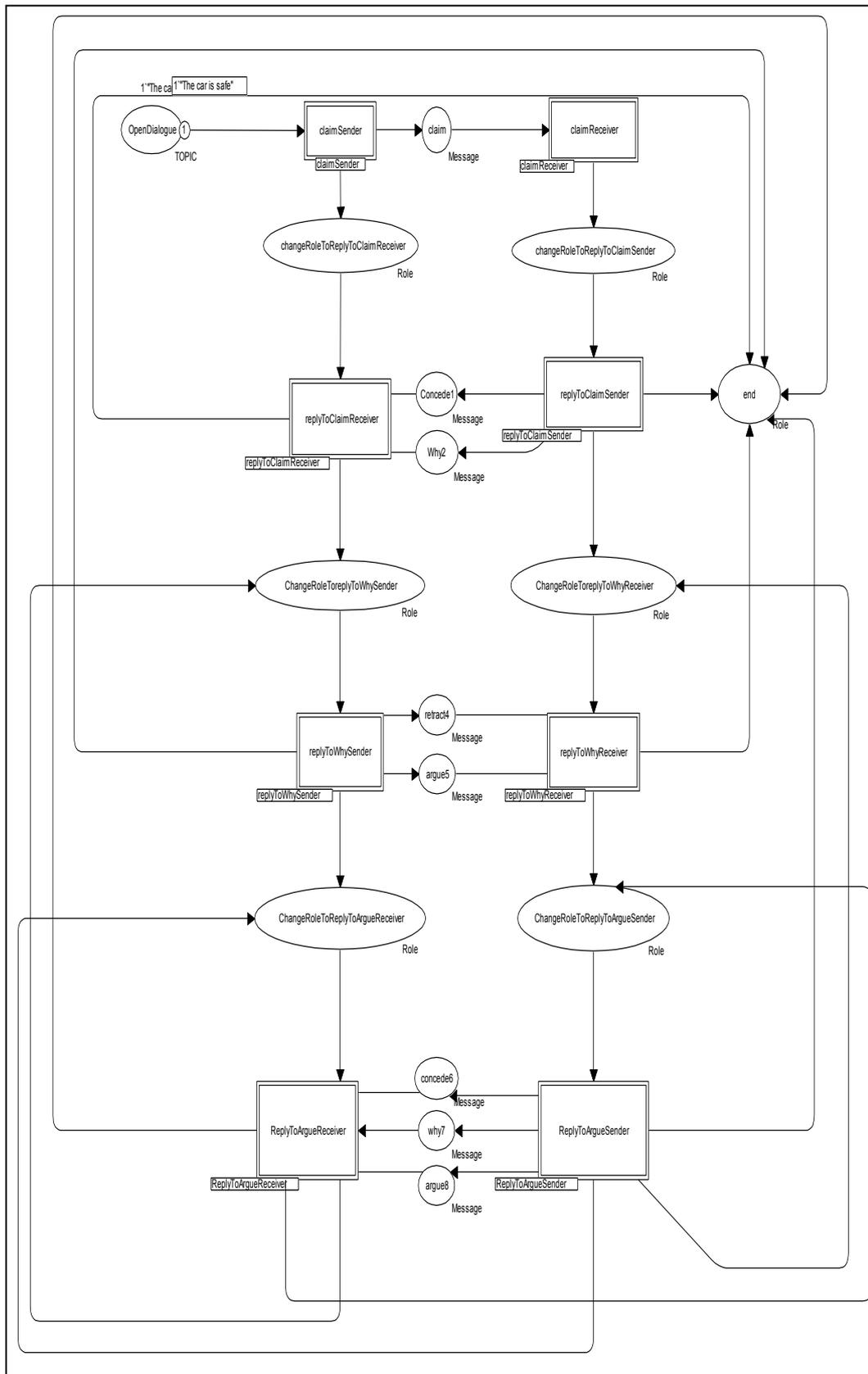


Figure C.18: The protocol CPN Superpage

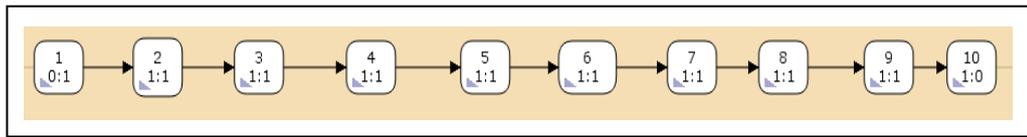


Figure C.19: The State Space Graph

replyToWhyReceiver_O, *replyToArgueSender_O* and *replyToArgueReceiver_P* together and describes the interaction between these eight subpages. See Figure C.18.

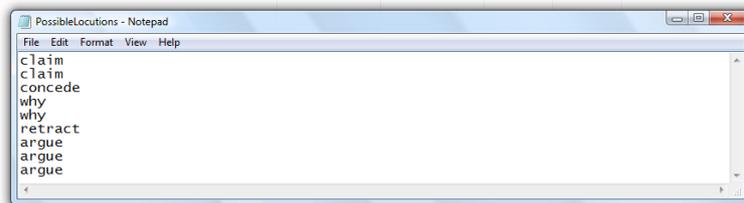
Step Two: Construction of State Space

The state space (shown in Figure C.19) for the CPN model of an LCC protocol for a persuasion dialogue is generated using the SS tool palette in CPN Tools (see chapter 6, section 6.2). Figure C.19 has ten nodes and nine arcs.

Step Three: Automated creation of DID properties files

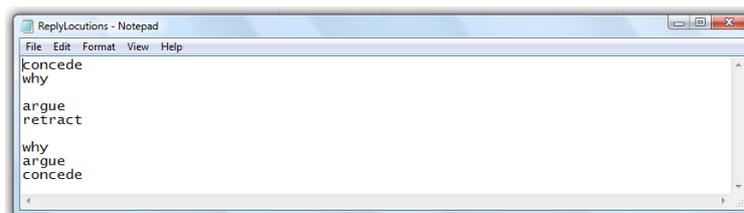
In this step, the verification tool creates ten property files automatically:

(1) Possible Locutions file:



In this example, *Possible Locutions* file contains the following set of permitted messages: *claim*, *concede*, *why*, *retract* and *argue*. Please note that, this file is connected with *Reply Locutions* file (see *Reply Locutions* file).

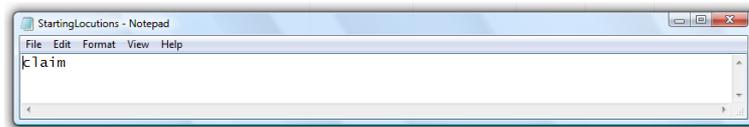
(2) Reply Locutions file:



Bridging the Specification Protocol Gap in Argumentation

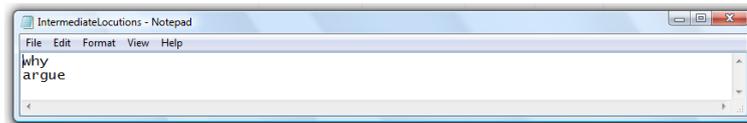
In this example, *Reply Locutions* file contains three sets of legal reply locutions: 1) *concede* and *why* (legal reply to *claim*); 2) *argue* and *retract* (legal reply to *why*); 3) *why*, *argue* and *concede* (legal reply to *argue*). Please note that, this file is connected with *Possible Locutions* file where each line in the *Reply Locutions* file represents the legal reply of the locution in the same line in the *Possible Locutions* file (e.g. *concede* in the first line of the *Reply Locutions* file represents the legal reply of the *claim* locution in the first line in the *Possible Locutions* file).

(3) Starting Locutions file:



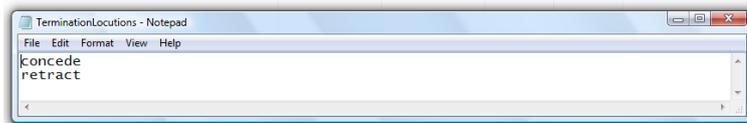
In this example, *Starting Locutions* file contains one message name *claim* which is used to begin the persuasion dialogue.

(4) Intermediate Locutions file:



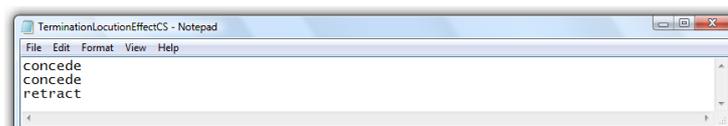
In this example, *Intermediate Locutions* file contains two message names *why* and *argue* which are used to remain in the dialogue.

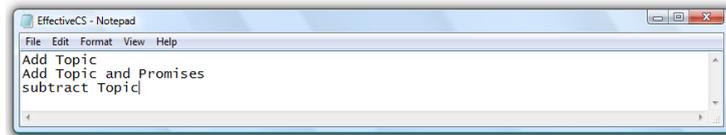
(5) Termination Locutions file:



In this example, *Termination Locutions* file contains two message names *concede* and *retract* which are used to terminate the persuasion dialogue;

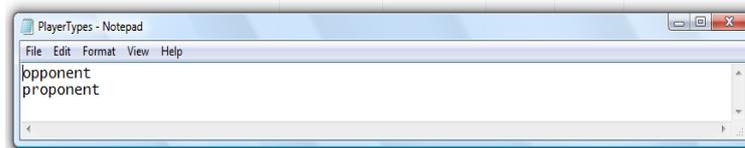
(6) Termination Locutions Effect CS and Effective CS files:





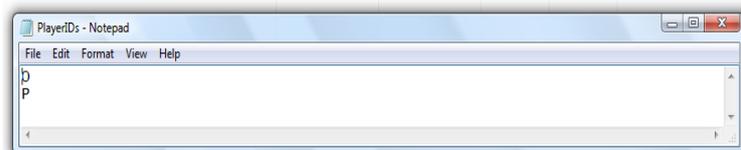
In this example, the tool creates two connected files *Termination Locutions Effect CS* which contains the termination messages (*concede* after *claim*, *concede* after *argue* and *retract* after *why*) and *Effective CS Locutions* file which contains the effect of the termination message to the sender commitment store CS (*concede* after *claim* = Add Topic to CS, *concede* after *argue* = Add Topic and Promises to CS and *retract* after *why* = subtract Topic from CS).

(7) Player Types file:



In this example, *Player Types* file contains *opponent* (the audience) and *proponent* (the speaker who is responsible for opening the persuasion dialogue) as player types.

(8) Player IDs file:



In this example, *Player IDs* file contains *O* and *P* as player IDs. Please note that, this file is connected with *Player Types* file (*O* represents the ID of the *opponent* and *P* represent the ID of the *proponent*).

(9) Termination Role Names file:





Figure C.20: Dialogue Opening Property Page

In this example, *Termination Role Names* file contains three role names *replyToClaimReceiver*, *replyToArgueReceiver* and *replyToWhyReceiver*. Please note that, this file is connected with *Termination Locutions Effect CS* file (*replyToClaimReceiver* role receives *concede* after *claim*, *replyToArgueReceiver* role receives *concede* after *argue* and *replyToWhyReceiver* role receives *retract* after *why*).

Step Four: Applying Verification Model

The generated CPN model from step two has five properties CPN pages (Dialogue opening property, Termination of a dialogue property, Turn taking between agents property, Message sequencing property and Recursive message property). To verify these five basic properties the following actions were performed:

- (1) Open the CPN model by using the CPN Tool;
- (2) Select the *Evaluates a Text as ML Code(ML!)* icon in the simulation tool palette and apply it to these five basic properties pages (Figures C.20, C.21, C.22, C.23 and C.24 show the properties pages after applying the *ML!* to them);
- (3) Select the *Show Verification Result* from the verification menu bar in the *GenerateLCCProtocol* tool to show the verification result (Figure C.25 shows the verification result of the five basic properties).

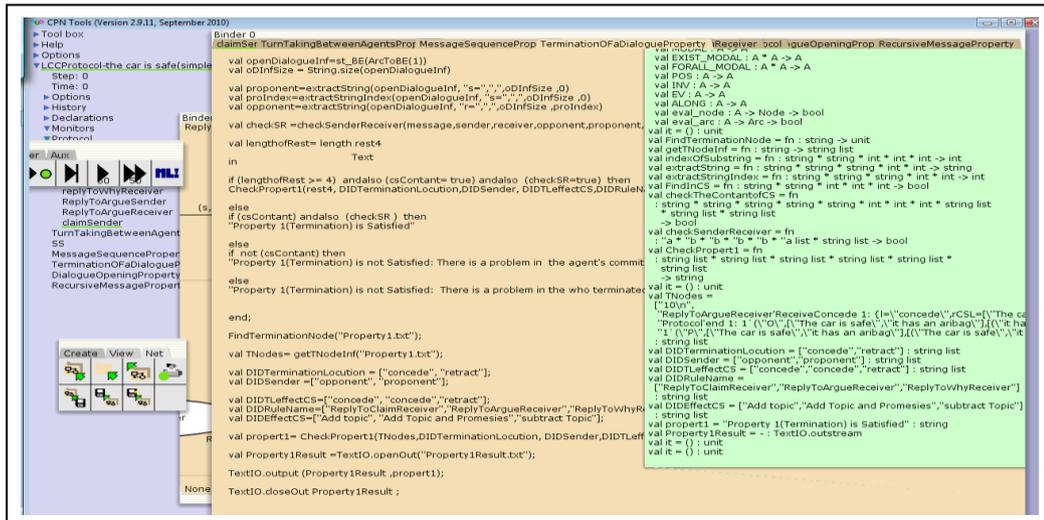


Figure C.21: Termination of a Dialogue Property Page

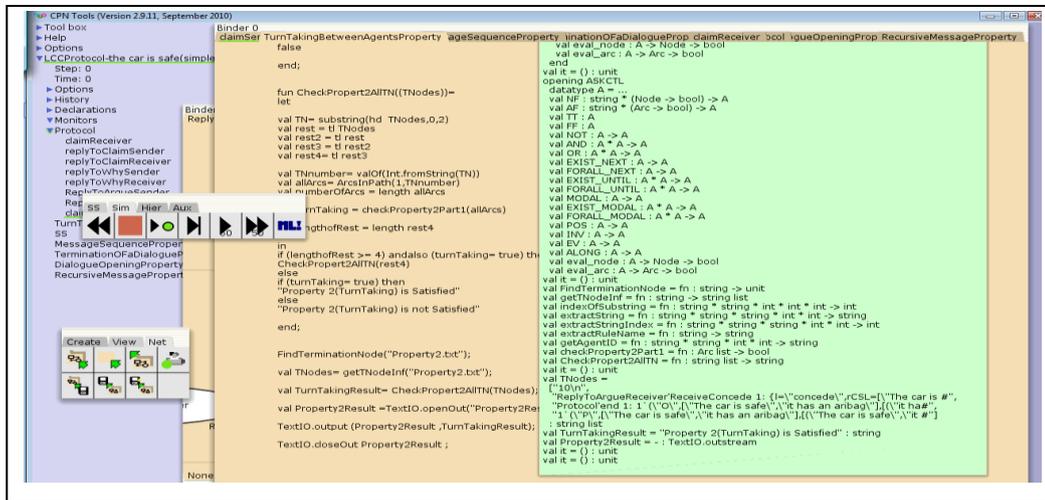


Figure C.22: Turn Taking between Agents Property Page

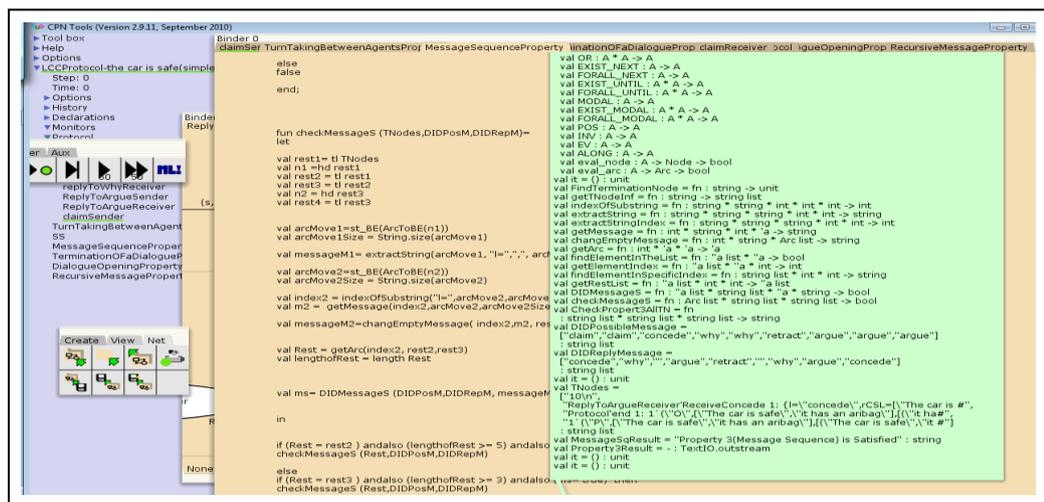


Figure C.23: Message Sequencing Property Page

Bridging the Specification Protocol Gap in Argumentation

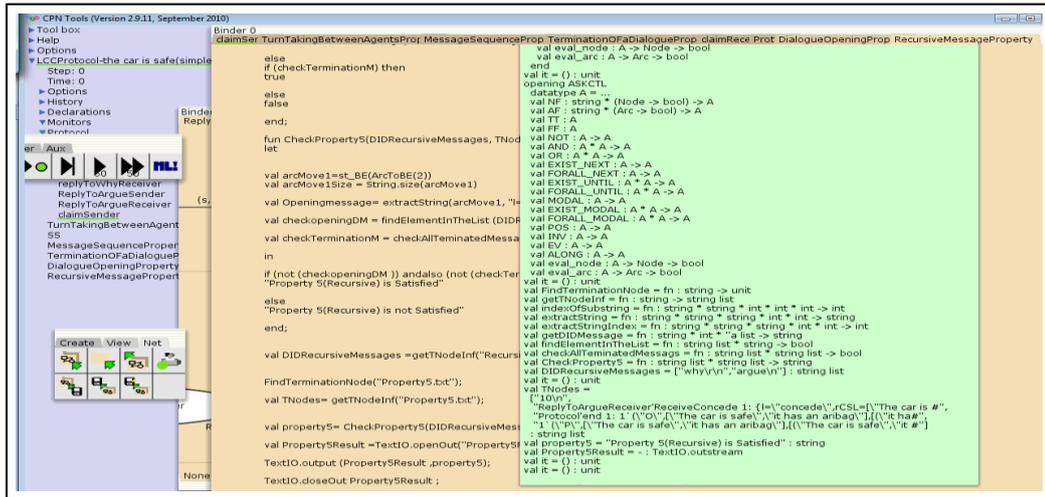


Figure C.24: Recursive Message Property Page

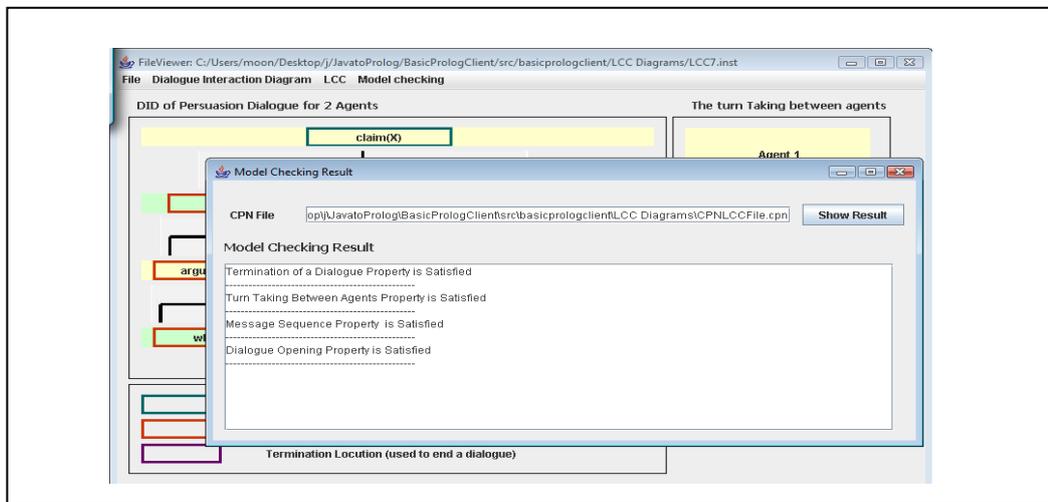


Figure C.25: The Verification Result of the Five Basic Properties

Appendix D

CPN Functions

This appendix presents basic CPN functions code, where²³:

ins_new = Inserts an item into the list

mem = return true if it is able to find an item in the list

union = Inserts more than one item into the list

rma = removes an item from the list

CPN Functions

(14) Add an argument 't' to a commitment store list 'sCSL':

```
fun addTopicToCS(sCSL,t) = ins_new sCSL t;
```

(15) Add a premise of an argument 't' to a commitment store list 'sCSL':

```
fun addPremiseToCS(sCSL,t,p) =  
  
  if (mem sCSL t) then ins_new sCSL p  
  
  else union sCSL [t,p] ;
```

(16) Add a defeat of a premise or an argument to a commitment store list 'sCSL':

```
fun addDefeatToCS(sCSL,def) = ins_new sCSL def;
```

²³ http://cpntools.org/documentation/concepts/colors/declarations/colorsets/implementation_of_list_fu

- (17) Subtract an argument 't' from a commitment store list 'sCSL':

```
fun subtractFromCS(sCSL,t) = rml t sCSL;
```

- (18) Find an argument 't' in a commitment store list 'sCSL':

```
fun findTopicInCS(sCSL,t) = mem sCSL t;
```

- (19) Find a premise of an argument 't' in a commitment store list 'sCSL':

```
fun findPreInCS(sCSL,t) = mem sCSL t;
```

- (20) Find an argument in a knowledge base list 'KBlist' where 'f' represents a fact and 'pre' represents a premise:

```
fun findTopicInKB((f,pre)::KBlist,t)=  
  
  if ((f = t)) then true  
  
  else if (length KBlist >=1) then findTopicInKB(KBlist,t)  
  
  else false;
```

- (21) Find a premise of an argument in a knowledge base list 'KBlist' where 'f' represents a fact and 'pre' represents a premise:

```
fun findPreInKB((f,pre)::KBlist,t)=  
  
  if (f=t) then true  
  
  else if (length KBlist >=1) then findPremiseInKB(KBlist,t)  
  
  else false;
```

- (22) Find a defeat of a premise or an argument in a knowledge base list 'KBlist' where 'f' represents a fact and 'def' represents a defeat of a premise 'pre':

```

fun findDefeatInKB((f,def)::KBlist,t)=

  if (substring(f,0,3)="not") andalso (substring(f,4,(String.size t))= t)

  then true

  else if (length KBlist >=1) then findDefeatInKB(KBlist,t)

  else false;
    
```

- (23) Find the opposite of an argument 't' in commitment store list 'sCSL':

```

fun findOppTopicInCS(sCSL,t)=mem sCSL ("not "^t);
    
```

- (24) Find the opposite of the premise 'p' of an argument 't' in commitment store list 'sCSL':

```

fun findOppPreInCS(sCSL,p)=mem sCSL ("not "^p);
    
```

- (25) Return (get) the premise of an argument 't' from a knowledge base list 'KBlist' where 'f' represents a fact and 'pre' represents a premise:

```

fun getPremiseFromKB((f,pre)::KBlist,t)=

  if (f=t) then 1`pre

  else getPremiseFromKB(KBlist,t);
    
```

- (26) Return (get) the defeat of an argument 't' from a knowledge base list 'KBlist' where 'f' represents a fact and 'def' represents a defeat of a premise 'pre':

```

fun getDefeatFromKB((f,def)::KBlist,t)=

  if (substring(f,0,3)="not") andalso (substring(f,4,(String.size t))= t)

  then 1`def

  else getDefeatFromKB(KBlist,t);
    
```

Appendix E

***GenerateLCCProtocol* Tool Graphical User Interface**

This appendix explains how the user can interact with the *GenerateLCCProtocol* tool. It begins by a description of the graphical user interface for synthesis of concrete protocols screens in Section E.1. A description of the graphical user interface for verification model screens is represented in Section E.2. This appendix does not provide details of the underlying tool implementation.

E.1 Graphical User Interface for Synthesis of Concrete Protocols (Part One)

E.1.1 Dialogue Interaction Diagram

Generate LCC Protocol Tool Main Screen

A screenshot of the *GenerateLCCProtocol* tool main screen is shown in Figure E.1:

- (1) The first button is used to open the DID library screen (as shown in Figure E.2). The DID library screen displays a set of current DID diagrams.
- (2) The second button is used to create a new DID diagram screen (as shown in Figure E.3).

Dialogue Interaction Diagram Library Screen

Chapter 4 describes the DID language in detail. DID is used to specify the dialogue game protocol in an abstract way. It provides mechanisms to represent interaction

Bridging the Specification Protocol Gap in Argumentation

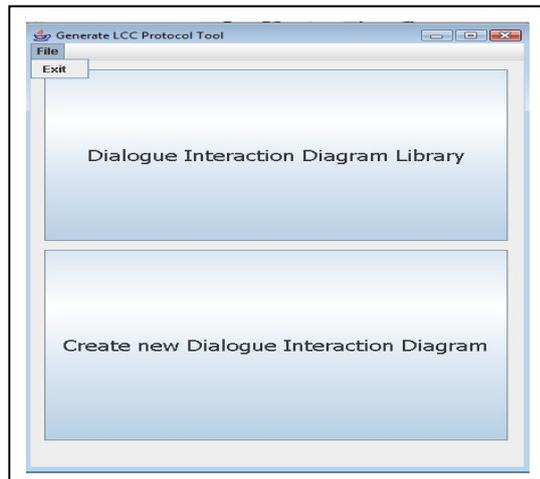


Figure E.1: Generate LCC Protocol Tool Main Screen

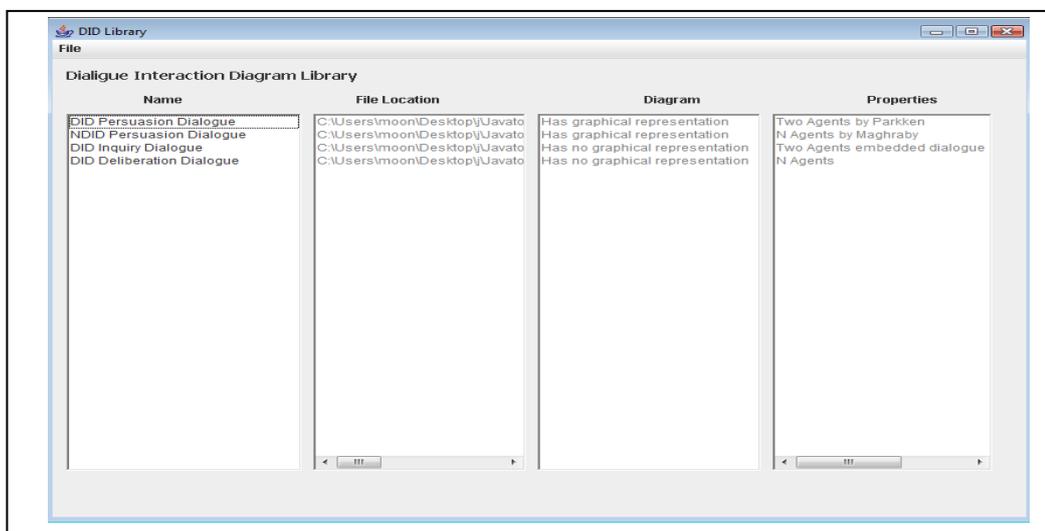


Figure E.2: Dialogue Interaction Diagram Library Screen

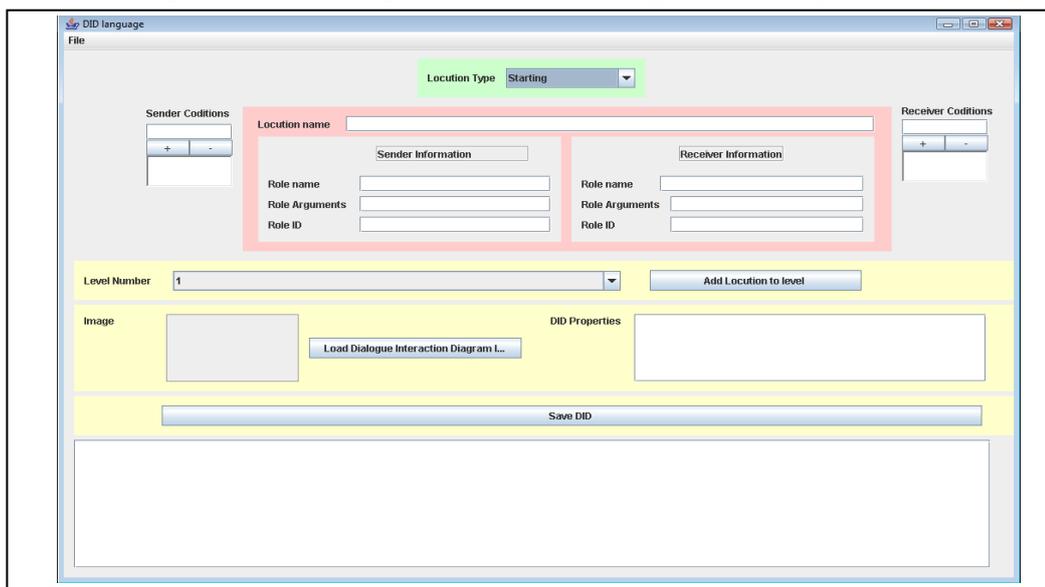


Figure E.3: Create New Dialogue Interaction Diagram Screen

protocol rules between two agents, by allowing the designer to specify the permitted messages (moves or locutions) and their relationship to each other.

A screenshot of the DID library screen is shown in Figure E.2. It contains all current DID diagram information:

- (1) Name: the name of the DID file has no formal meaning. However, expressive DID names have a positive impact on the human reader; consequently, providing a name that the human reader can understand is important.
- (2) File location: specifies the DID file directory name. It specifies a unique location in the user file system.
- (3) Diagram: specifies whether or not the DID has a graphical representation.
- (4) Properties: specifies the DID properties which could indicate the number of players and the dialogue game rules. These properties of the DID file have no formal meaning. These properties enable a better understanding of the DID file.

The four pieces of information presented above are provided by the designer during the creation process of DID diagram (see next section for more information).

Open DID

To open an existing DID diagram, the user needs to double click on the DID file name:

- (1) If the DID file has a graphical representation, a simple graphical representation version of the DID will be displayed. For example, if the user double clicks the DID persuasion dialogue (in Figure E.2), the DID of a persuasion dialogue screen will open with a simple graphical representation version of the DID diagram reply structure rules (as shown in Figure E.4). Figure 4.3 in chapter 4 illustrates the full DID graphical representation of this persuasion dialogue.

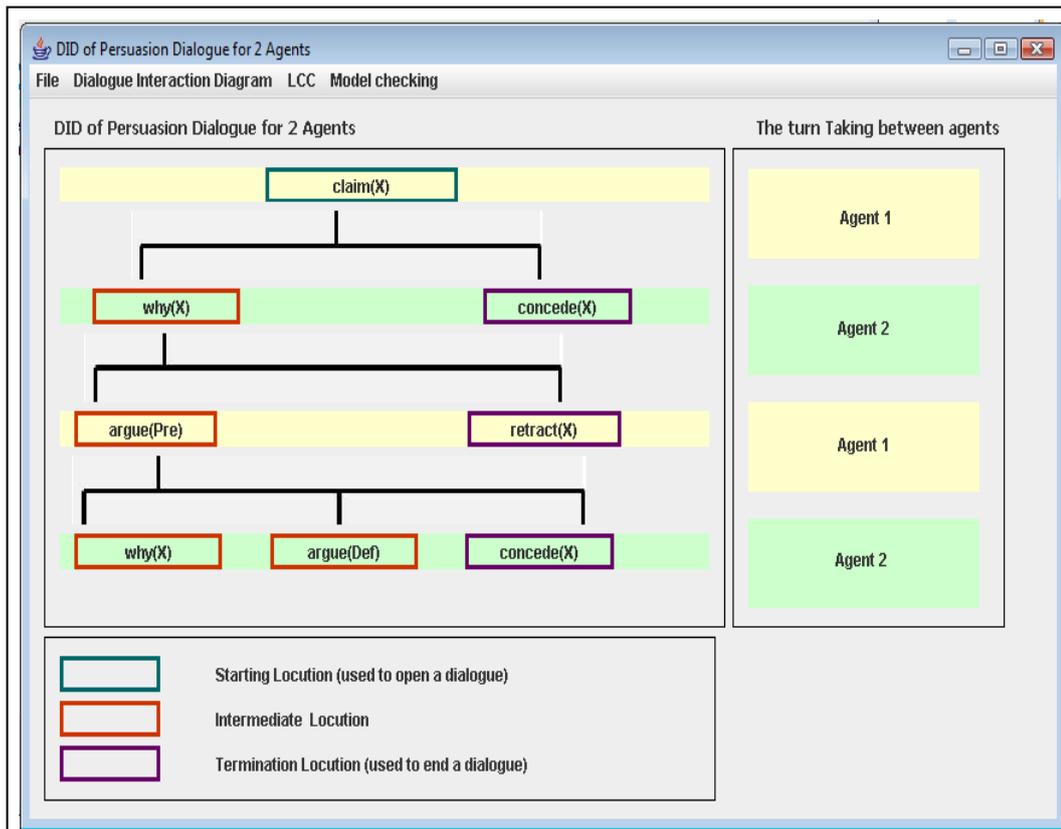


Figure E.4: Simple DID Graphical Representation of a Persuasion Dialogue

```

FileViewer: C:\Users\moon\Desktop\JavatoProlog\BasicPrologClient\src\basicprologclient\LCC Diagrams\DIInquiry.txt
File LCC Model checking

DID
1:
Locution(Starting,open(y),null),
Sender-Information[OpenSender,(KBpp, CSp, CSp, QS, y),IDp,findInKB(y,KBPP) and emptyCS(CSPP) and addToQueryStore(y,QS) and addToOpenReceiver-Information[OpenReceiver,(KBp, CSp, CSp, QS, y),IDp].
2:
Locution[Recursive,assert(F,y),open(y)],
Sender-Information[replyToOpenSender,(y,KBp, CSp, CSp, QS),IDp,findInQS(y) and notFindInCS(F,CSp) and find((F,y),KBp, CSp) and addToCS(F,Receiver-Information[replyToOpenReceiver,(y, KBpp, CSp, CSp, QS),IDp].
2:
Locution[Recursive,open(g),open(y)],
Sender-Information[replyToOpenSender,(y,KBp, CSp, CSp, QS),IDp,isRelationship(g,y) and findInQS(QS,y) and findInKB(g,KB) and notFindInQS(QS,Receiver-Information[replyToOpenReceiver,(y, KBpp, CSp, CSp, QS),IDp].
2:
Locution[Recursive,close(y),open(y)],
Sender-Information[replyToOpenSender,(y,KBp, CSp, CSp, QS),IDp,findInOpenDialogue(y,OpenD) findInOpenDialogue(y,OpenD) and allSubDialogueReceiver-Information[replyToOpenReceiver,(y, KBpp, CSp, CSp, QS),IDp].
3:
Locution[Recursive,close(g),open(g)],

```

Figure E.5: DID Formal Representation of an Inquiry Dialogue

- (2) If the DID file has no graphical representation, a formal representation version of the DID will be displayed. For example, if the user double clicks the DID inquiry dialogue(in Figure 7.2), the DID of an inquiry dialogue screen will open with a formal representation version of the DID diagram reply structure rules (as shown in Figure E.5). Figure 4.9 in chapter 4 illustrates the DID graphical representation of this inquiry dialogue.

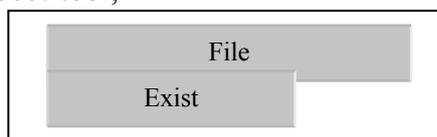
Simple Version of DID Graphical Representation Screen

This screen displays a simple version of the DID graphical representation of a dialogue game (as shown in Figure E.4). This graph represents the permitted messages (moves or locutions) and their relationship to each other and the turn-taking between agents. However, to make it simple for a human reader, both pre-conditions and post-conditions for messages are not shown in this screen.

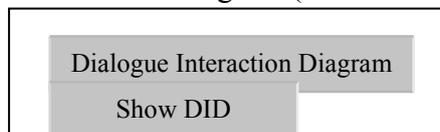
The lower part of this screen shows the messages (locutions) types (see section 4.2.1 in chapter 4 for more detail).

The upper part of this screen shows four menu bars:

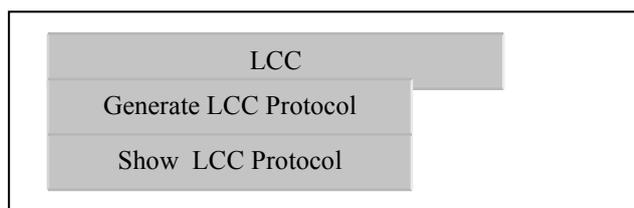
- (1) File menu bar: this menu has an exist button which it is used to exist the *GenerateLCCProtocol* tool;



- (2) Dialogue Interaction Diagram menu bar: this menu shows the DID button which is used to display the full DID diagram (as shown in Figure E.6).



- (3) LCC menu bar: this menu has three buttons:



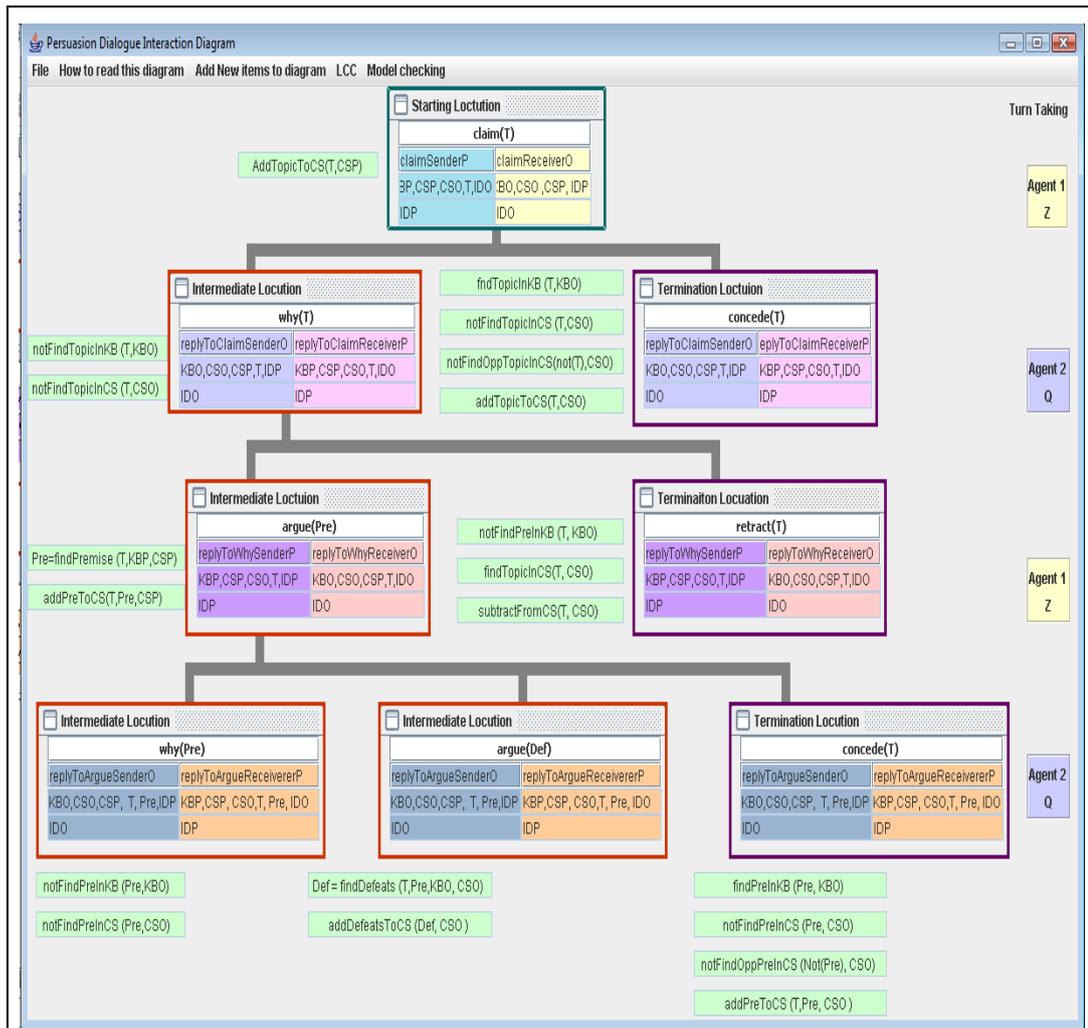
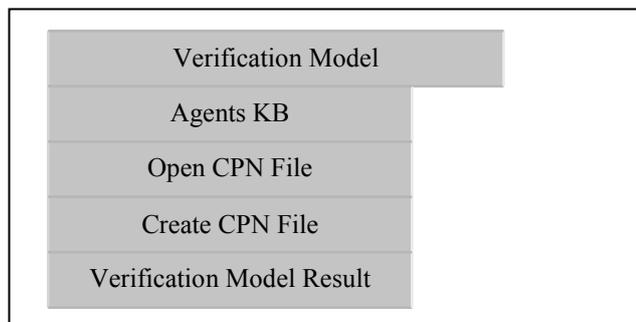


Figure E.6: Full DID Graphical Representation of a Persuasion Dialogue

- a) Generate LCC Protocol: used to generate an LCC protocol from a DID diagram;
- b) Show LCC Protocol: used to display the generated LCC protocol.

Section E.1.2 explains these three buttons in more detail.

(4) Verification Model menu bar: this menu has four buttons:



- a) Agents KB: used to get the agents Knowledge Base (KB) from the user;
- b) Create CPN File: used to create a CPN file from the generated LCC protocol;
- c) Open CPN File: used to display the created CPN file;
- d) Verification Model Result: used to display the verification model result of the five basic properties (Dialogue opening property, Termination of a dialogue property, Turn taking between agents property, Message sequencing property and Recursive message property).

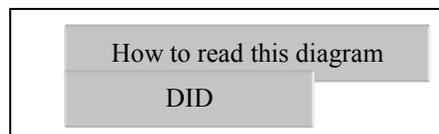
Section E.2 explains these four buttons in more detail.

Full Version of DID Graphical Representation Screen

This screen displays a full version of the DID graphical representation of a dialogue game (as shown in Figure E.6). This graph represents the permitted messages (moves or locutions) and their relationship to each other, the turn-taking between agents, pre-conditions and post-conditions for the messages. Figure 4.3 in chapter 4 illustrates the same DID graphical representation of the persuasion dialogue.

The upper part of this screen shows five menu bars:

- (1) File menu bar (see above explanations of file menu);
- (2) How to read this diagram: this menu has the DID button which is used to display how to read DID screen (as shown in Figure E.7 (a) and (b)).



How to read the DID screen (Figure E.7 (a) and (b)) has five tabs. If the user selects a tab by clicking it, the tabbed panel displays the information corresponding to the tab:

Bridging the Specification Protocol Gap in Argumentation

The figure consists of three screenshots of a software interface titled "How to read DID".

Locution icon tab: This screenshot shows a window with several tabs: "Locution icon", "Users are allowed to change", "Meaning of varibale", "Meaning of color", and "Locution type". The "Locution type" tab is active, displaying a diagram of a locution icon. The icon is a rectangle divided into three sections. The top section is labeled "Locution name" and contains "Sender Role name" and "Receiver Role name". The middle section is labeled "Recursive arguments" and contains "Sender Role arguments" and "Receiver Role arguments". The bottom section is labeled "Sender Conditon" and contains "Sender ID" and "Receiver ID". A "Receiver Conditon" label is also present. Below the diagram is a text box explaining the structure of a locution icon.

Users are allowed to change Tab: This screenshot shows the "Users are allowed to change" tab. It contains a text box stating "User are only allowed to change roles name and locations name". Below this is a diagram showing a "Locution Lactum" box with "Role name" and "Location name" labels. The "Role name" label is connected to a box containing "replyToClaims", "denyClaims", "IDG", and "IDZ". The "Location name" label is connected to a box containing "consider(X)". Below the diagram are two text boxes: "User are allowed to add new conditions by clicking on 'Add new item to diagram'" and "User are allowed to add new arguments by clicking on 'Add new item to diagram'". At the bottom, there is a large button labeled "Add new item to diagram" with a sub-menu showing "Argument" and "Condition".

Meaning of Variable Tab: This screenshot shows the "Meaning of varibale" tab. It displays two tables. The first table is titled "Persuasion Dialogue Interaction Diagram" and lists variables and their meanings. The second table is titled "Persuasion Dialogue Interaction Diagram for N-agent" and lists variables and their meanings.

Variable	Meaning
T	dialogue topic
KB	knowledge base list: represents the domain of discourse(Agent believes)
CS	Int store: contains a set of propositions to which the player is committed in the discussion
Pre	claim supporting premises
Def	defeat premises for claim

Variable	Meaning
Topic	dialogue topic
Acceptance	list of agents that have accepted the claim
Rejection	list of agents that have rejected the claim
NAccAgents	Number of agents that have accepted the claim
NRejAgents	Number of agents that have rejected the claim
AgentGroup	Group of agents

Figure E.7 (a): How to Read DID

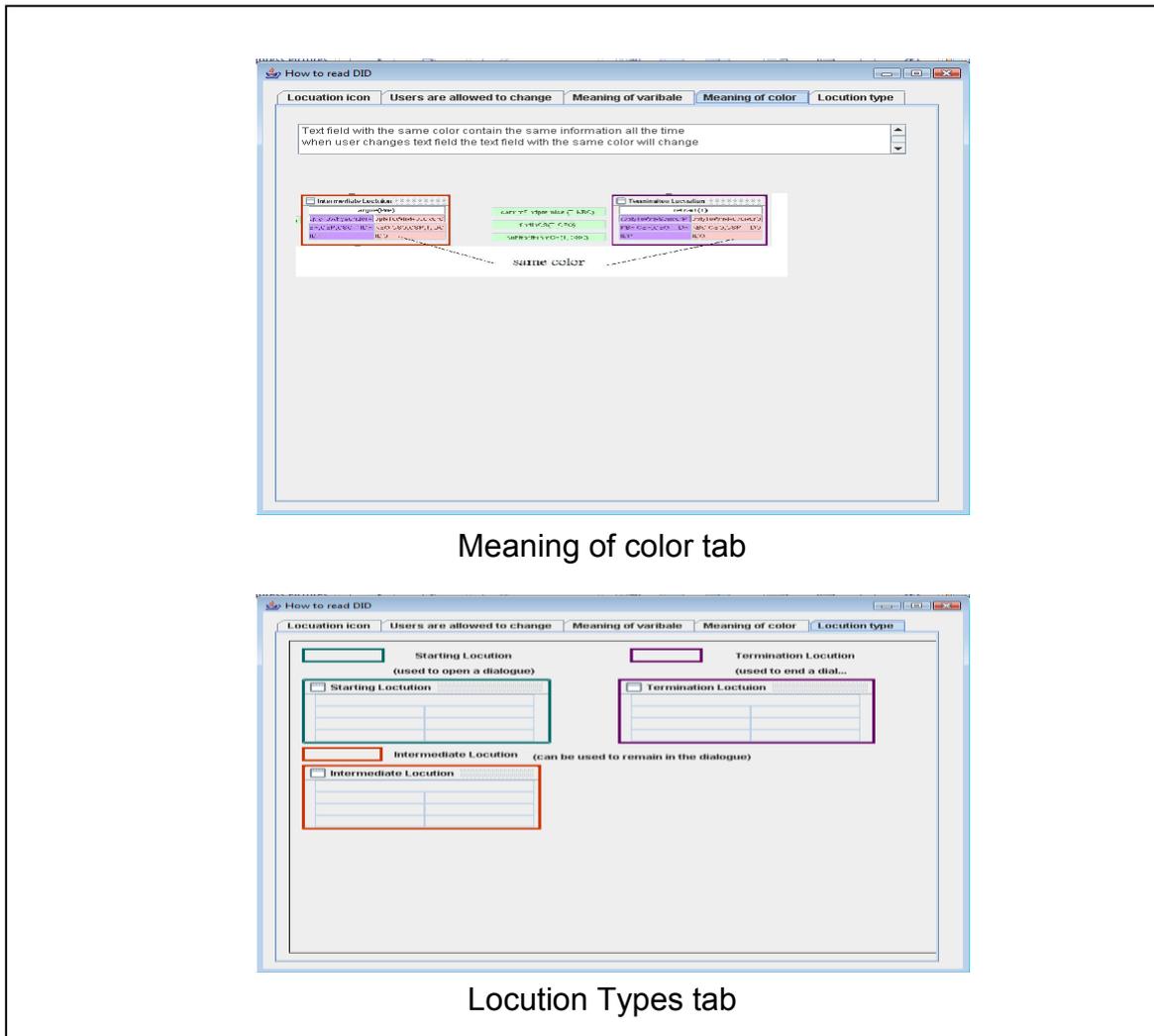


Figure E.7 (b): How to Read DID

- a) Locution icon tab: explains a locution icon (see section 4.2 in chapter 4 for more details about the locution icon);
- b) The users are allowed to change tab: it explains that the current user is allowed to change the locution icon information and to add new arguments and conditons;
- c) Meaning of variables tab: displays a brief description of each variable (argument) in the DID;
- d) Meaning of color tab: the sender (or receiver) role name, arguments and agent ID with the same colours have the same values and therefore the role

information must be the same for all locutions (with the same colours) at the same level since each level has one role. In other words, text fields with the



Figure E.8: Add New Argument Subscreen

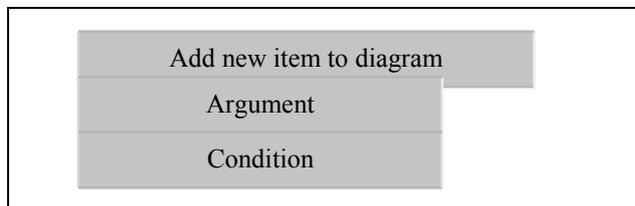


Figure E.9: Add New Condition Subscreen

same color contain the same information all the time. When the user changes one text field, text fields with the same color will change;

- e) Locution types tab: displays the three locution icon types (see section 4.2 in chapter 4 for more details about locution types).

(3) Add new item to diagram menu: this menu has two buttons:



- a) Argument: used to add a new argument to either a specific role or all roles. When the user clicks on the argument button, a new subscreen appears (as shown in Figure E.8). For example, if the user want to add an argument 'L' to 'claimSenderP' role, he/she needs to write the argument name 'L' in the argument text field, then select 'Add to specific roles', and then select the 'claimSenderP' role from roles list and finally click on the apply button which adds the argument 'L' to the 'claimSenderP' role.

- b) Condition: used to add new conditions to a specific role. When the user clicks on the condition button, a new subscreen appears (as shown in Figure E.9). For example, if the user wants to add the condition 'add(T,CSO)' to the 'claimReceiverO' role, he/she needs to select the locution name 'claim(T)' from the locution list, then select the role name 'claimReceiverO' from the roles list, and then write the new condition 'add(T,CSO)' in the condition text field and finally click on the apply button which adds the condition 'add(T,CSO)' to the 'claimReceiverP' role.

(4) LCC menu bar (see above explanations of LCC menu);

(5) Verification Model menu bar (see above explanations of Verification Model menu);

Textual Version of DID Screen

Unfortunately, some DID files have no graphical representation (see section 8.3 in chapter 8 and chapter 9 for more details). However, all the DID specifications have a textual representation. Figure E.5 illustrates an example of the DID formal representation of an inquiry dialogue (Figure 4.9 in chapter 4 illustrates the DID graphical representation of this inquiry dialogue). The user does not have to learn the formal representation of the DID, unless he needs to edit it (e.g. user needs to add new condition to a specific locution icon).

- | |
|--|
| <ol style="list-style-type: none">1. Level number:2. Locution[Locution Type,Locution, Structural rules],3. Sender-Information[Role Name,Role arguments,Agent ID,Conditions],4. Receiver-Information[Role Name,Role arguments,Agent ID,Conditions]. |
|--|

Figure E.10: DID Textual Representation

DID Textual Representaion

The DID textual representation describes each locution icon by using 4 lines (as shown in Figure E.10):

- (1) Line 1: represents the DID level. The DID levels are ordered by number, beginning with level number 1.
- (2) Line 2: represents the locution icon information where:
 - a) Locution Types: there are only three types of locutions: *Starting*, *Intermediate* and *Termination*;
 - b) Locution name: represents the locution (message or move) name (e.g. *claim(T)*);
 - c) Structural rules: represents the previous locution (message or move) name. Note that if the locution type is *Starting*, the Structural rules = *null*.
- (3) Line 3: represents sender role information (sender role name, sender role arguments, sender agent ID and sender role pre-conditions).
- (4) Line 4: represents receiver role information (receiver role name, receiver role arguments, receiver agent ID and receiver post-conditions).

Figure E.11 illustrates this with an example of a textual definition of claim locution of a persuasion dialogue which is shown in Figure E.6:

- (1) Line 1: represents DID level *1* (since claim is the first locution in the DID).
- (2) Line 2: represents locution icon information where:
 - a) Locution Type = *Starting*;
 - b) Locution name = *claim(T)*;
 - c) Structural rules = *null* (since Locution type= *Starting*).
- (3) Line 3: represents the sender role information where:
 - a) Role name = *claimSender*;
 - b) Role arguments = *KB_P, CS_P, CS_O, T, ID_O*;
 - c) Agent ID = *ID_P*;

1. 1:
2. **Locution**[Starting,claim(T),null],
3. **Sender-Information**[claimSender_p, (KB_p, CS_p, CS_o, T, ID_o), ID_p, addTopicToCS(T, CS_p)],
4. **Receiver-Information**[claimReceiver_o, (KB_o, CS_o, CS_p, ID_p), ID_o].

Figure E.11: DID Textual Representation of Claim Locution

d) Sender conditions = *addTopicToCS(T, CS_p)*.

(4) Line 4: represents the receiver role information where

- a) Role name = *claimReceiver_o*;
- b) Role arguments = *KB_o, CS_o, CS_p, ID_p*;
- c) Agent ID = *ID_o*;
- d) Receiver conditions = *null*.

Create Dialogue Interaction Diagram Screen

This screen allows the user to create new DID diagrams (as shown in Figure E.12) by writing one locution icon information (locution type, locution structural rules locution name, sender information, receiver information and locution level number) at a time beginning from the locution in the top of the DID (see chapter 4). This screen also allows the user to describe the DID diagram by writing some of its properties in the properties text field as well as loads the DID image by clicking on the 'Load DID image' (if there is an image or graphical representation for this dialogue). Please note the following:

- (1) Clicking on the 'Add locution to level' button adds the locution icon's to the DID textual representation (see DID Textual Representaion section).
- (2) Clicking on the 'Save DID' button saves the DID file and shows a dialog box which asks the user if he/she would like to open the DID file (see Figure E.13). The DID file textual representation screen will appear when the user click on 'Yes' button (see Figure E.5).

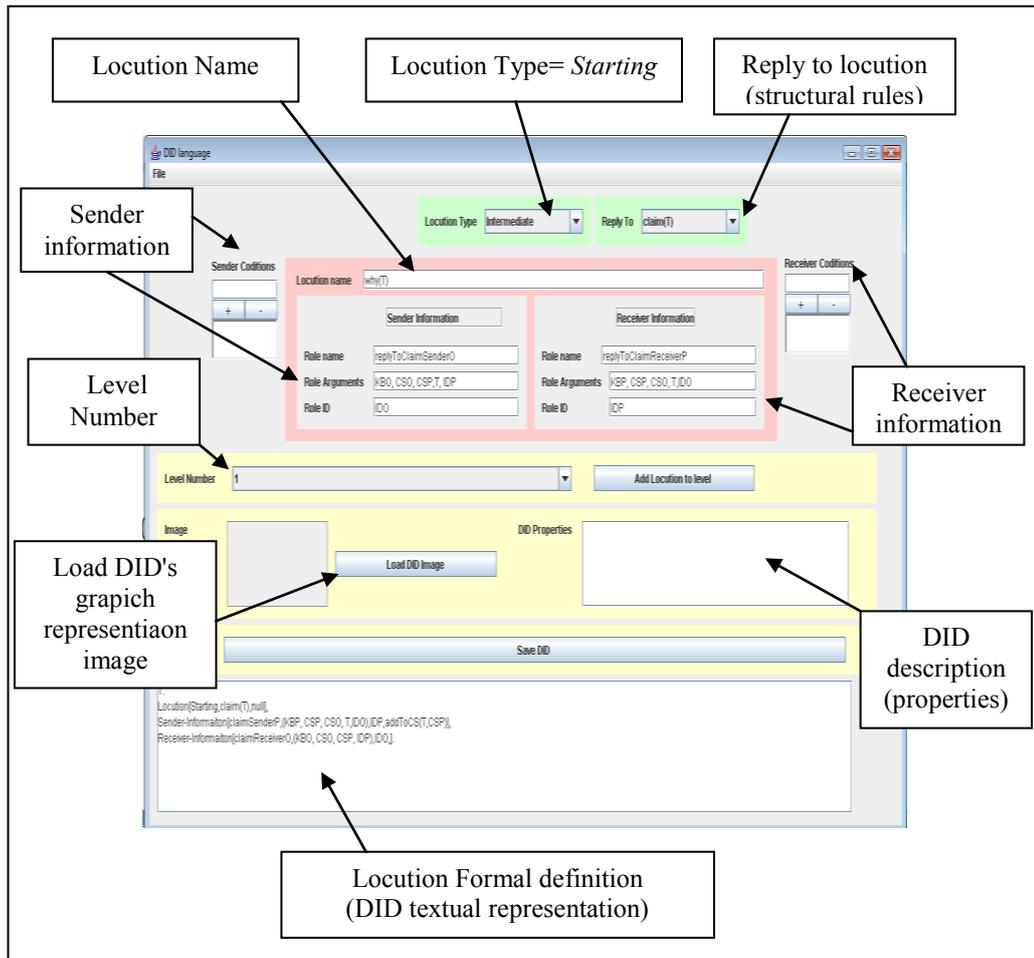
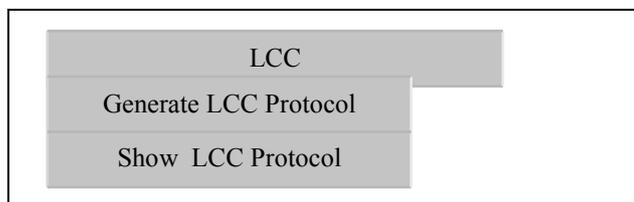


Figure E.12: Create New Dialogue Interaction Diagram Screen



Figure E.13: Open DID File Dialog Box

E.1.2 Synthesising Concrete LCC Protocols from DID Specifications

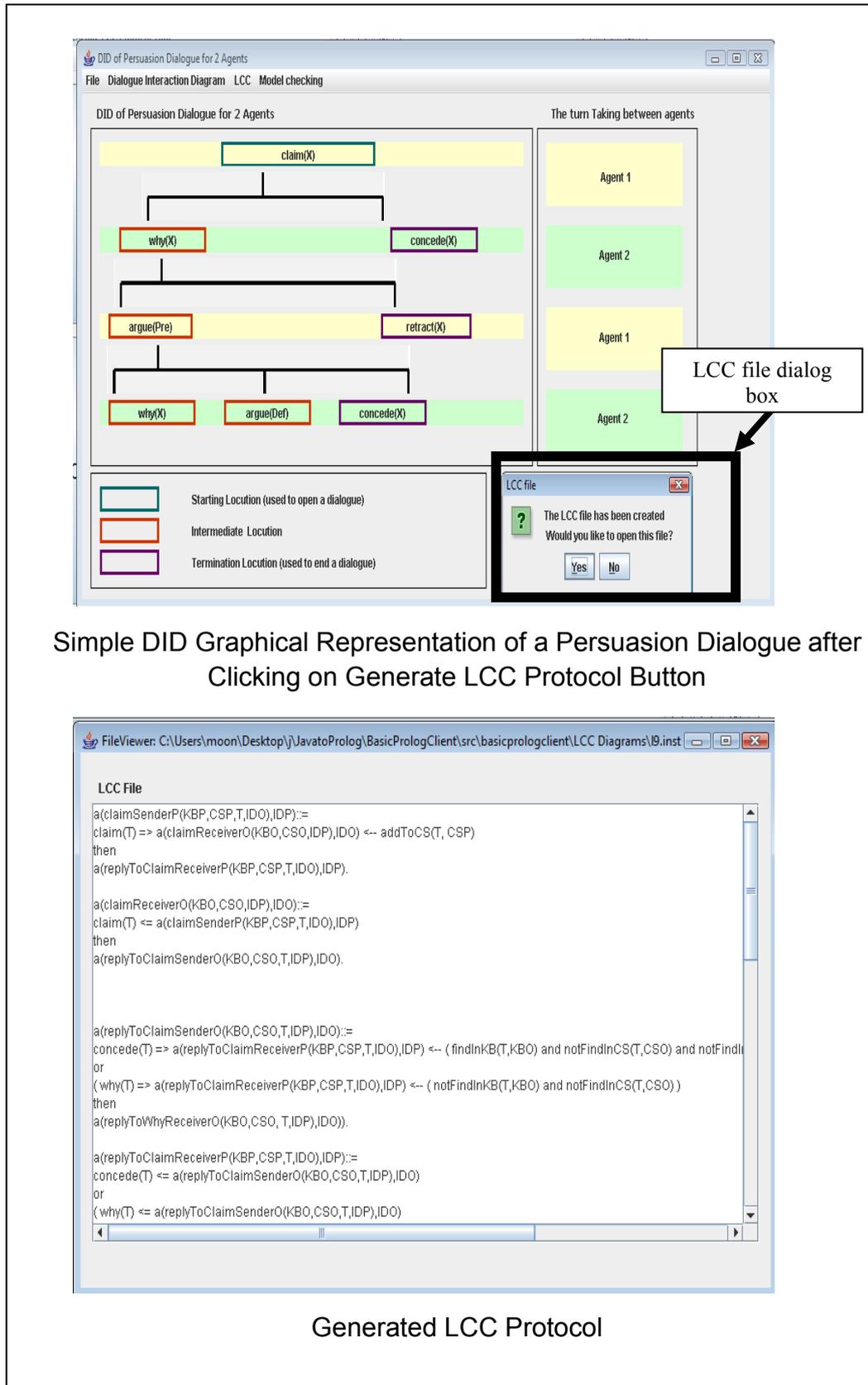


From the LCC menu bar (the LCC menu bar appears on the simple DID graphical representation screen, on the full DID graphical representation screen and also on the DID formal representation screen) the user can:

- (1) Generate concrete LCC protocols from the DID specifications automatically, by clicking on the 'Generate LCC Protocol' button. Synthesise LCC protocols from the DID specifications process by recursively applying the LCC-Argument patterns. This process will be fully automatic (requiring no human assistance). The LCC-Argument patterns and the automated synthesis process are exhibited in chapter 5. When the user clicks on the 'Generate LCC Protocol' button (for instance, in the simple DID graphical representation screen of a persuasion dialogue in Figure E.4), the tool will generate the LCC protocol and the LCC file dialog box will appear. The user has to click on the 'Yes' button to display the generated LCC protocol (as shown in Figure E.14). Appendix C gives a detailed description of how to synthesise a DID of a persuasion dialogue to an LCC protocol by using LCC-Argument patterns. In the case of N-agent, the user needs to select the DID for two agents, then select the divided group condition and finally click on the 'Generate LCC Protocol' button (as shown in Figure E.15).
- (2) Display the generated LCC protocols by clicking on the 'Show LCC Protocol' button. For example, if the user wants to see the generated LCC protocol of a persuasion dialogue, he/she needs to click on the 'Show LCC Protocol' button and then load the LCC persuasion dialogue file by clicking on the 'Load file' button (as shown in Figure E.16);

E.2 A Graphical User Interface for Verification Model (Part Two)

From the Verification Model menu bar (the Verification Model menu bar appears on: the simple DID graphical representation screen, on the full DID graphical representation screen and also on the DID formal representation screen) the user can (see Figure E.17):



Simple DID Graphical Representation of a Persuasion Dialogue after Clicking on Generate LCC Protocol Button

Figure E.14: Generate a Concrete LCC Protocol for the Persuasion Dialogue

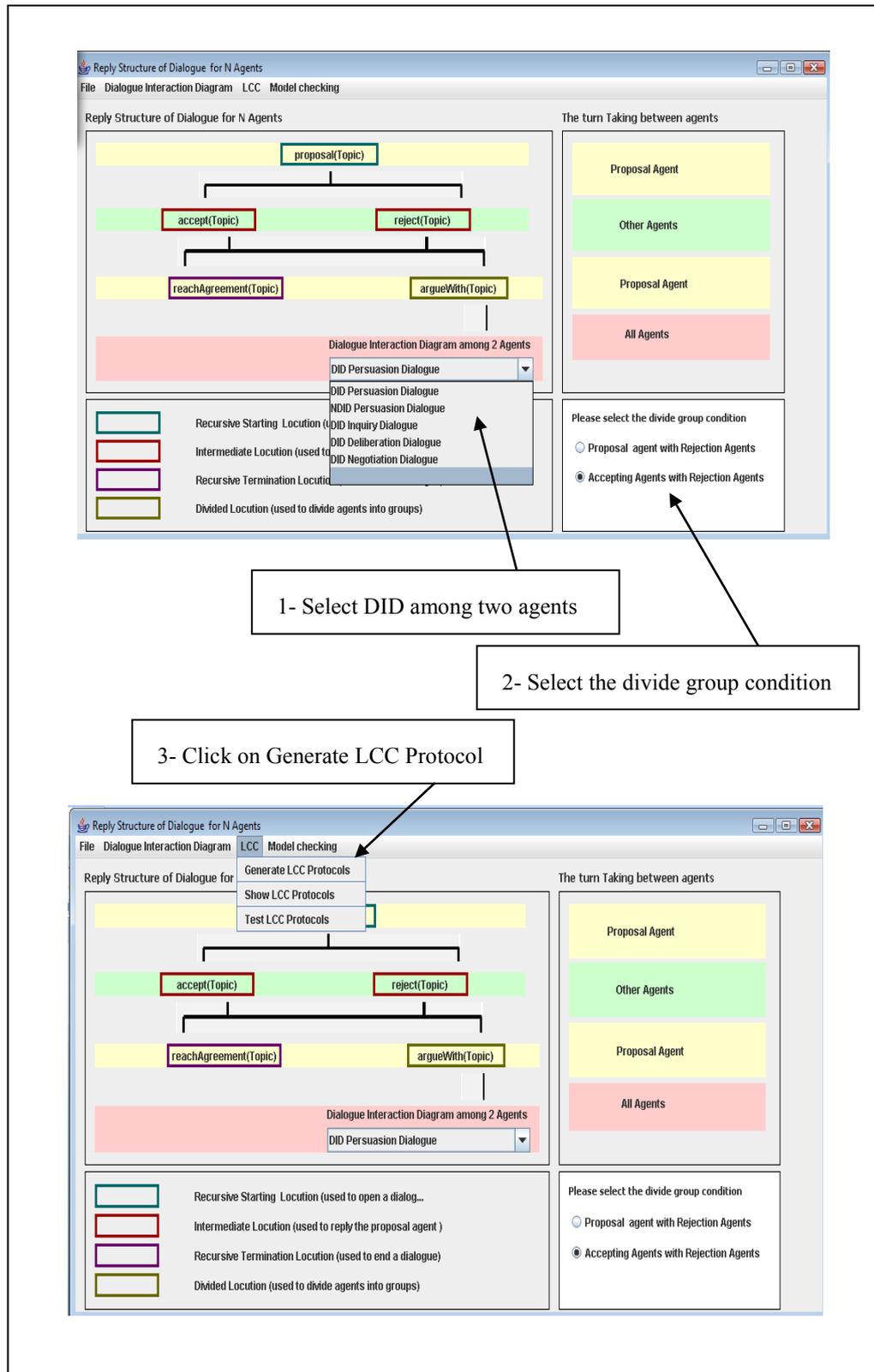


Figure E.15: Generate a Concrete LCC Protocol for the Persuasion Dialogue among N-agent

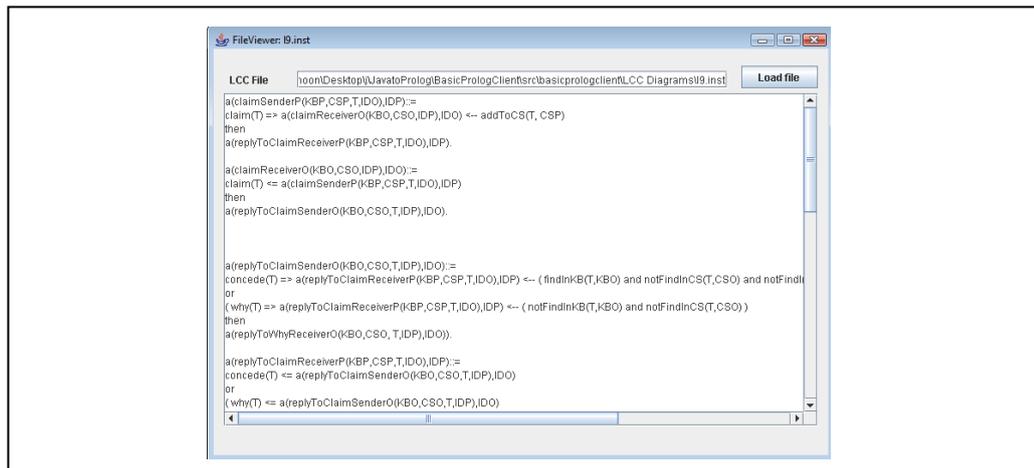


Figure E.16: Show Generated LCC Protocols Screen

- (1) Specify agents Knowledge Base (KB) by clicking on the 'Agents KB' button (see chapter 7).
- (2) Create a CPN model (CPNXML) file from the generated LCC protocol and create the DID properties files by click on the 'Create CPN File' button (see chapter 7).
- (3) Display the created CPN model file by click on the 'Open CPN File' button (see chapter 7).
- (4) Display the verification model result of the five basic properties (Dialogue opening property, Termination of a dialogue property, Turn taking between agents property, Message sequencing property and Recursive message property) by click on the 'Verification Model Result' button (see chapter 7).

Appendix F

Published Papers

The published papers of this research are:

- (1) MAGHRABY ASHWAG, ROBERTSON DAVE, GRANDO ADELA and ROVATSOS, MICHAEL. Automated Deployment of Argumentation Protocols. In VERHEIJ BART, SZEIDER STEFAN and WOLTRAN STEFAN, *Computational Models of Argument*. Vienna, Austria IOS Press, 2011.
<http://homepages.inf.ed.ac.uk/mrovatso/papers/maghrabyetal-comma2012.pdf>
<http://homepages.inf.ed.ac.uk/s0961321/AshwagMagharby-PaperCOMMA.pdf>
- (2) MAGHRABY ASHWAG, ROBERTSON DAVE, GRANDO ADELA and ROVATSOS, MICHAEL. *Bridging the specification protocol gap in argumentation*. Argumentation in Multiagent Systems (ArgMAS), Valencia, Spain, June 2012.
<http://www.mit.edu/~irahwan/argmas/argmas12/>
<http://homepages.inf.ed.ac.uk/s0961321/AshwagMagharby-PaperArgMAS.pdf>
- (3) MAGHRABY ASHWAG. *Automatic Agent Protocol Generation from Argumentation*. 13th European Agent Systems Summer School (EASSS 2011), Girona, Catalonia (Spain), July 2011.
<http://eia.udg.edu/easss2011/resources/docs/paper1.pdf>
<http://homepages.inf.ed.ac.uk/s0961321/AshwagMagharby-PaperEASSS.pdf>
- (4) MAGHRABY ASHWAG, ROBERTSON DAVE, GRANDO ADELA and ROVATSOS, MICHAEL. *Bridging the Specification-Protocol Gap in Argumentation*. 5th Saudi International Conference (SIC2011), The University of Warwick, Coventry, June 2011.
<http://homepages.inf.ed.ac.uk/s0961321/AshwagMagharby-Paper2011.pdf>

Bibliography

- [Aalst, 2005] AALST, WIL VAN DER. Pi Calculus Versus Petri Nets: Let Us Eat Humble Pie Rather Than Further Inflate the Pi Hype. *BPTrends*, 3(5): May 2005:1-11.
- [Aalst and Stahl, 2011] AALST, Wil Van Van Der and STAHL, Christian. *Modeling Business Processes: A Petri Net-Oriented Approach*. Cambridge, Mass./US, MIT Press, 2011.
- [Alexander et.al,1977] ALEXANDER, Christopher, ISHIKAWA Sara and SILVERSTEIN Murray. *A pattern language: towns, buildings, construction*. New York, Oxford University Press,1977.
- [Amogud et.al.2000] AMOGUD, LEILA, PARSONS, SIMON and MAUDET, NICOLAS. Arguments, dialogue, and negotiation. *Journal of Artificial Intelligence Research*, (23):August 2000:338-342.
- [Appleton,1998] APPLETON, BRAD. Patterns and Software: Essential Concepts and Terminology. *Object Magazine Online*, 3(5):May 1998:20-25.
- [Aridor and Lange, 1998] ARIDOR, YARIV and LANGE, DANNY. Agent Design Patterns: Elements of Agent Application Design. AGENTS '98, *In Proceedings of the second international conference on Autonomous agents*. New York, ACM Press, 1998.
- [Atkinson et al., 2005] ATKINSON, KATIE, BENCH-CAPON, TREVOR and MCBURNEY, PETER. A Dialogue Game Protocol for Multi-Agent Argument over Proposals for Action. *Autonomous Agents and Multi-Agent Systems*, 11(2):2005:153–171.
- [Baeten,2005] BAETEN, J.C.M.. A Brief History of Process Algebra. *Theoretical Computer Science*, 2-3(335):23 May 2005:131-146.

[Bauer et.al., 2001] BAUER, BERNHARD, MÜLLER, JÖRG and ODELL, JAMES. Agent UML: A Formalism for Specifying Multiagent Interaction. *Software Engineering and Knowledge Engineering*, (11): 2001: 91-103.

[Besana, 2009] BESANA, PAOLO. *Comparison between choreography languages*. Edinburgh, The university of Edinburgh, 2009.

[Besana and Barker, 2009] BESANA, PAOLO and BARKER, ADAM. An Executable Calculus for Service Choreography. In MEERSMAN ROBERT, DILLON THARAM and HERRERO PILAR, *On the Move to Meaningful Internet Systems: OTM 2009*. Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009, Vilamoura, Portugal, Springer Berlin Heidelberg, 2009.

[Besnard and Hunter, 2008] BESNARD, Philippe and HUNTER, Anthony. *Elements of Argumentation*. Cambridge, Massachusetts London, England, MIT Press, 2008.

[Billington et al., 2003] BILLINGTON, JONATHAN, CHRISTENSEN, SØREN, HEE, KEES, KINDLER, EKKART, KUMMER, OLAF, PETRUCCI, LAURE, POST, REINIER, STEHNO, CHRISTIAN and WEBER, MICHAEL. *The Petri Net Markup Language: Concepts, Technology, and Tools*. In AALST WIL and BEST EIKE, *Applications and Theory of Petri Nets 2003*. The Netherlands, 24th International Conference, ICATPN 2003 Eindhoven, 2003.

[Black and Hunter, 2009] BLACK, ELIZABETH and HUNTER, ANTHONY. An inquiry dialogue system. *Autonomous Agents and Multi-Agent Systems*, 2(19): 2009:10-1007.

[Black and Hunter, 2007] BLACK, ELIZABETH and HUNTER, ANTHONY. A generative inquiry dialogue system. In Proceedings of the Sixth International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2007). New York, ACM, 2007.

[Bowles et al., 1994] BOWLES, ANDREW, ROBERTSON, DAVE, VASCONCELOS, WAMBERTO, VARGAS-VERA, MARIA, and BENTAL,

DIANA. Applying prolog programming techniques. *International Journal of Human-Computer Studies*, 41(3):1994:329-350.

[Bowles, 1994] BOWLES, ANDREW. *A techniques editor for Prolog novices*. Internal software report, available by the author, 1994.

[Bradfield and Stirling, 2006] BRADFIELD, JULIAN and STIRLING, COLIN. Modal mu-calculi. In BLACKBURN, PATRICK, BENTHEM, JOHAN and WOLTER, FRANK, *The Handbook of Modal Logic*. Oxford, Elsevier Science, 2006.

[Budinsky et.al., 1996] BUDINSKY, FRANK, FINNIE, MARILYN, YU, PATSY and VLISSIDES, JOHN. Automatic Code Generation from Design Patterns. *IBM Systems Journal*, 2(35):1996:151-171.

[Chesnevar et al.,2007] CHESÑEVAR, CARLOS, MCGINNIS, JARRED, SANJAY, MODGIL, IYAD, RAHWAN, CHRIS, REED, GUILLERMO, SIMARI, MATTHEW, SOUTH, GERARD, VREESWIJK and WILLMOTT, STEVEN. Towards an argument interchange format. *The Knowledge Engineering Review*, 4(21):2007:293–316.

[Deugo and Weiss, 1999] DEUGO, DWIGHT and WEISS, MICHAEL. A Case for Mobile Agent Patterns. In PAPAIOANNOU TODD and MINAR NELSON, *Mobile Agents in the Context of Competition and Cooperation (MAC3) Workshop Notes*. Seattle, at Autonomous Agents'99, 1999.

[Dignum and Vreeswijk, 2003] DIGNUM, FRANK and VREESWIJK, GERARD. Towards a test bed for multi-party dialogues. In DIGNUM FRANK, *Advances in Agent Communication*. Melbourne, Australia, International Workshop on Agent Communication Languages, 2003.

[Dijkman and Dumas, 2004] DIJKMAN, REMCO and DUMAS, MARLON. Service-oriented Design: A Multi-viewpoint Approach. *International Journal of Cooperative Information Systems*, 4(13): 2004:337-378.

[Dimopoulos et. al., 2005] DIMOPOULOS, YANNIS, KAKAS, ANTONIS and MORAITIS, PAVLOS. Argumentation based Modelling of Embedded Agent Dialogues. In PARSONS, SIMON, MAUDET, NICOLAS, MORAITIS, PAVLOS and RAHWAN, IYAD, *Argumentation in Multi-Agent Systems*. Second International Workshop, ArgMAS 2005 Utrecht, The Netherlands, Springer Berlin Heidelberg, 2005.

[Ding and Su, 2008] DING, YANLAN and SU, GUIPING. A Reduction method for Verification of Security Protocol through CPN. *In process of IEEE International Conference on Networking, Sensing and Control*. Sanya, China, IEEE, 2008.

[Doutre et. al.,2005] DOUTRE, SYLVIE, MCBURNEY, PETER, WOOLDRIDGE, MICHAEL, and BARDEN, WILLIAM. *Information-seeking agent dialogs with permissions and arguments*. Technical Report ULCS-05-010, Department of computer science, University of Liverpool, Liverpool, UK. 2005, www.csc.liv.ac.uk/research/techreports/tr2005/tr05010abs.html.

[Eemeren et al., 1987] EEMEREN, Frans, GROOTENDORST, Rob and KRUIGER, Tjark. *Handbook Argumentation Theory: A critical survey of classical backgrounds and modern studies*. Dordrecht, Foris Publications, 1987.

[Eunice, 2005] EUNICE, Marta. *Model transformation support for the analysis of large-scale systems*. Texas Tech University Electronic Theses and Dissertations, Master Thesis in Software Engineering, 2005.

[Floreani et al.,1996] FLOREANI, DANIEL, BILLINGTON, JONATHAN, and DADEJ, AREK. Designing and Verifying a Communications Gateway Using Coloured Petri Nets and Design/CPN. In BILLINGTON JONATHAN and REISIG WOLFGANG, *Application and Theory of Petri Nets 1996*. Osaka, Japan, 17th International Conference on Application and Theory of Petri Nets, 1996.

[Fox et.al, 2006] FOX, JOHN, GLASSPOOL, DAVID, MODGIL, SANJAY, TOLCHINKSY, PANCHO and BLACK, LIZ. Towards a canonical framework for designing agents to support healthcare organizations. *In Proceedings of ECAI-06*

Workshop on Agents Applied in HealthCare, 17th European Conference on Artificial Intelligence. Italy, 2006.

[Gamma et.al, 1995] GAMMA, Erich, HELM, Richard, JOHNSON, Ralph, and VLISSIDES, John. *Design patterns: elements of reusable object-oriented software*. Canada, Addison Wesley, 1995.

[Goldfarb and Prescod, 2003] GOLDFARB, Charles and PRESCOD, Paul. *XML Handbook (5th Edition)*. Prentice Hall PTR, the University of Virginia, 2003.

[Gordon, 2008] GORDON, THOMAS. Constructing Legal Arguments with Rules in the Legal Knowledge Interchange Format (LKIF). In CASANOVAS, POMPEU, SARTOR GIOVANNI, CASELLAS, NURIA and RUBINO, ROSSELLA, *Computable Models of the Law*. Berlin, Heidelberg, Springer-Verlag, 2008.

[Grivas, 2005] GRIVAS, Argyrios. *A Structural Synthesis System for LCC Protocols*. PhD thesis, University of Edinburgh, 2005.

[Hamblin, 1970] HAMBLIN, Charles. *Fallacies*. London, Methuen young books, 1970.

[Hassan et.al., 2005] HASSAN, FADZIL, ROBERTSON, DAVE and WALTON, CHRIS. Addressing Constraint Failures in Agent Interaction Protocol. *In Proceedings of the 8th Pacific Rim International Workshop on Multi-Agent Systems*. Kuala Lumpur, Malasia, 2005.

[Ito and Shintani, 1997] ITO, TAKAYUKI and SHINTANI, TORAMATSU. An Agenda-scheduling System Based on Persuasion Among Agents. *In Proceedings of the International Symposium on Information Systems and Technologies for Network Society*. World Scientific, 1997.

[Ito and Shintani, 1996] ITO, TAKAYUKI and SHINTANI, TORAMATSU. Persuasion among Agents: An Approach to Implementing a Group Decision Support System Based on Multi-agent Negotiation. *In Proceedings of the 5th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1996.

[Jensen and Kristensen, 2009] JENSEN, Kurt and KRISTENSEN, Lars. *Coloured Petri Nets Modelling and Validation of Concurrent Systems*. Berlin, Springer Verlag, 2009.

[Jensen *et al.*, 2007] JENSEN, KURT, KRISTENSEN, LARS, and WELLS, LISA. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 3(9): 2007:213–254.

[Jensen *et al.*, 2006] Jensen, Kurt, Christensen, Soren and Kristensen, Lars, *CPN Tools State Space Manual*, University of Aarhus, Department of computer science, 2006, retrieved 2013, http://cpntools.org/_media/documentation/manual.pdf.

[Jensen *et al.*, 2002] Jensen, Kurt, Christensen, Soren and Kristensen, Lars, *CPN Tools State Space Manual*, University of Aarhus, Department of computer science, 2002, retrieved 2013.

[Jensen, 1992] JENSEN, Kurt. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Berlin, Springer Verlag, 1992.

[Jennings *et al.*,1998] JENNINGS, NICHOLAS, SYCARA, KATIA and WOOLDRIDGE, MICHAEL. Roadmap of agent research and development. *Journal of Autonomous Agents and Multi- Agent Systems*, 1(1):1998:7-38.

[Krauss,2008] KRAUSS, Alexander. *Defining Recursive Functions in Isabelle/HOL*, 2008, URL: <http://isabelle.in.tum.de/doc/functions.pdf>.

[Kristensen *et. al.*, 1998] KRISTENSEN, LARS, SØREN, CHRISTENSEN, and KURT, JENSEN. The Practioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2): 1998:98-132.

[Kirschenbaum *at.al*, 1989] KIRSCHENBAUM, MARC, LAKHOTIA, ARUN, and STERLING, LEON. *Skeletons and Techniques for Prolog Programming*. Centre for automation and Intelligent System Researches, Case Western Reserve University, Technical Report, 1989.

[Lloyd, 1994] LLOYD, JOHN. *Practical advantages of declarative programming*. In Joint Conference on Declarative Programming, GULP-PRODE'94, 1994.

[Luo et al., 2001] LUO, XUDONG, MIAO CHUNYAN, JENNINGS NICHOLAS, HE MINGHUA, SHEN ZHIQI, and ZHANG MINJIE. KEMNAD: A Knowledge Engineering Methodology for Negotiating Agent Development. *Computational Intelligence*, 1(28):2012:51-105.

[MANNA and WALDINGER, 1980] MANNA ZOHAR and WALDINGER RICHARD. A Deductive Approach to Program Synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):1980:90-121.

[Maudet et al., 2007] MAUDET, NICOLAS, PARSONS, SIMON, and RAHWAN, IYAD. Argumentation in multiagent system: context and recent developments. In *Proceedings of Argumentation in MultiAgent Systems (ARGMAS06)*. Japan, Springer-Verlag, 2007.

[McBurney et al., 2007] MCBURNEY, PETER, HITCHCOCK, DAVID, and PARSONS, SIMON. *The eightfold way of deliberation dialogues*. International Journal of Intelligent Systems, 1(22):2007: 95-132.

[McBurney and Parsons, 2003] MCBURNEY, PETER and PARSONS, SIMON. Dialogue Game Protocols. In HUGET MARC-PHILIPPE, *Communication in Multiagent Systems*. Germany, Springer Verlag, Berlin, 2003.

[McBurney et al., 2003] MCBURNEY, PETER, EIJK, ROGIER, PARSONS, SIMON and AMGOUD, LEILA. A Dialogue-Game Protocol for Agent Purchase Negotiations. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(7):2003:235- 273.

[McBurney and Parsons, 2002] PETER, MCBURNEY and PARSONS, SIMON. Games that agents play: A formal framework for dialogues between autonomous agents. *Journal of Logic, Language and Information*, 3(11):2002:315-334.

[McBurney et al., 2002] MCBURNEY, PETER, PARSONS, SIMON and WOOLDRIDGE, MICHAEL. Desiderata for agent argumentation protocols. In

Proceedings of the first international joint conference on Autonomous agents and multiagent systems part 1 AAMAS 02. New York, ACM, 2002.

[Milner e *et al.*, 1997] MILNER, Robin, TOFTE, Mads, HARPER, Robert, and MACQUEEN, David. *The Definition of Standard ML*. Cambridge, MA, USA, The MIT Press, revised edition, 1997.

[Modgil and Amgoud, 2008] MODGIL, SANJAY and AMGOUD, LEILA. *Agents and Arguments*. 10th European Agent Systems Summer School (EASSS 2008). Portugal, New University of Lisbon, 2008.

[Modgil and McGinnis, 2007] MODGIL, SANJAY and MCGINNIS, JARRED. Towards Characterising Argumentation Based Dialogue in the Argument Interchange Format. In RAHWAN, IYAD, PARSONS, SIMON AND REED CHRIS, *Argumentation in Multi-Agent Systems*. Honolulu, HI, USA, 2007.

[Murata, 1989] MURATA, TADAO. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 4(77): 1989:541-580.

[Nielsen and Simpson, 2000] NIELSEN, MOGENS and SIMPSON, DAN. *Application and Theory of Petri Nets 2000*. In Proceedings of 21st International Conference on Application and Theory of Petri Nets, Aarhus, Denmark, Springer, 2000.

[Norman et al.,2004] NORMAN, TIMOTHY, CARBOGIM, DANIELA, KRABBE, ERIK and WALTON, DOUGLAS. Argument and Multi-Agent Systems. In REED, CHRIS and NORMAN, TIMOTHY, *Argumentation Machines: New Frontiers in Argument and Computation*. Dordrecht, Kluwer Academic Publishers, 2004.

[Nwana et.al., 1996] NWANA H S, LEE L and JENNINGS N R. Co-ordination in software agent systems. *British Telecom Technical Journal*, 4(14):1996: 79-88.

[Odell, 2002] ODELL, JAMES. Objects and Agents Compared. *Journal of Object Technology*, 1(1):2002:41-53.

[Odifreddi and Cooper, 2012] ODIFREDDI, PIERGIORGIO and COOPER, S.BARRY, *Recursive Functions*, *The Stanford Encyclopedia of Philosophy*, ZALTA EDWARD, 2012, retrieved 2013,

<http://plato.stanford.edu/archives/fall2012/entries/recursive-functions>.

[O'Keefe,1990] O'KEEFE, Richard. *The Craft of Prolog (Logic Programming)*. Cambridge, MA, USA, The MIT Press,1990.

[Osman, 2007] OSMAN, NARDINE. *A Contextualised Trust Model for Distributed Open Systems*. In AKHGAR BABAK, *ICCS 2007, Proceedings of the 15th International Workshops on Conceptual Structures*. London, Springer-Verlag, 2007.

[Osman *et al.*, 2006] OSMAN, NARDINE, ROBERTSON, DAVID and WALTON, CHRISTOPHER. Run-Time Model Checking of Interaction and Deontic Models for MultiAgent Systems. In KLUSCH, MATTHIAS, ROVATSOS, MICHAEL and PAYNE, TERRY, *Cooperative Information Agents X: 10th International Workshop*. Edinburgh, UK, Springer, 2006.

[Parsons and McBurney, 2003] PARSONS, SIMON and MCBURNEY, PETER. Argumentation-Based Communication between Agents. In HUGET, M.-P, *Communication in Multi-Agent Systems: Agent Communication Languages and Conversation Policies, Lecture Notes in Artificial Intelligence 2650*. Berlin, Germany, Springer, 2003.

[Parsons *et al.*, 2003] PARSONS, SIMON, WOOLDRIDGE, MICHAEL and AMGOUD, LEILA. *Properties and Complexity of Some Formal Inter-agent Dialogues*. *Journal of Logic and Computation*, (13):2003:347-376.

[Parsons *et al.*, 1998] PARSONS, SIMON, SIERRA, CARLES and JENNINGS, NICK. *Agents that's reason and negotiate by arguing*. *Journal of logic and computation* 3(8):1998:261-292, 1998.

[Prakken, 2006] PRAKKEN, HENRY. Formal systems for persuasion dialogue. *The Knowledge Engineering Review*, 2(21):2006,163-188.

[Paschke et.al, 2006] PASCHKE, ADRIAN, KISS, CHRISTINE and AL-HUNATY, SAMER. *NPL: Negotiation Pattern Language- A Design Pattern Language for Decentralized (Agent) Coordination and Negotiation Protocols*. In BANDA R, *E-Negotiation - An Introduction*. ICFAI University Press, 2006.

[Prakken, 2005] PRAKKEN, HENRY. Coherence and flexibility in dialogue games for argumentation. *Journal of logic and computation*, 6(15):2005:1009-1040.

[Prakken and Vreeswijk, 2002] PRAKKEN, HENRY and VREESWIJK GERARD. Logics for defeasible argumentation. In GABBAY, DOV and GUNTHNER, F., *Handbook of Philosophical Logic*. Dordrecht, Kluwer Academic Publishers, 2002.

[Prakken, 2000] Prakken, Henry. *On dialogue systems with speech acts, arguments, and counterarguments*. In OJEDA-ACIEGO, MANUEL, GUZMÁN, INMA, BREWKA, GERHARD and PEREIRA, LUÍS, *Logics in Artificial Intelligence*. Málaga, Spain, Springer Verlag, 2000.

[Rahwan and Moraitis, 2009] RAHWAN, Iyad and MORAITIS, Pavlos. *Argumentation in Multi-Agent Systems: Fifth International Workshop, ArgMAS 2008*. Berlin, Germany, Springer-Verlag, 2009.

[Rahwan, 2006] RAHWAN IYAD. Guest Editorial: Argumentation in Multi-Agent Systems. *Autonomous Agents and Multiagent Systems*, 2(11):2006:115-125.

[Reed et al., 2008] REED, CHRIS, DEVEREUX, JOSEPH, WELLS, SIMON and ROWE ,GLENN. AIF+: Dialogue in the Argument Interchange Format. In BESNARD, PHILIPPE, DOUTRE, SYLVIE and HUNTER, ANTHONY, *Computational Models of Argument*. Toulouse, France, Proceedings of COMMA-2008, IOS Press, 2008

[Reed et al., 2010] REED, CHRIS, WELLS, SIMON, BUDZYNSKA, KATARZYNA and DEVEREU, JOSEPH. Building arguments with argumentation: the role of illocutionary force in computational models of argument. *In Proceedings of the Third International Conference on Computational Models of Argument (COMMA 2010)*. Amsterdam, The Netherlands, IOS Press, 2010.

[Reed, 1998] REED, CHRIS. Dialogue Frames in Agent Communication. In DEMAZEAU, YVES, *the Third International Conference on Multi-Agent Systems*. Washington, DC, USA, IEEE Computer Society Press, 1998.

[Robertson, 2004] ROBERTSON, DAVE. Multi-agent coordination as distributed logic programming. In DEMOEN, BART and LIFSCHITZ, VLADIMIR, *Logic programming*. Saint-Malo, France, 20th International Conference, 2004.

[Robertson, 1991] ROBERTSON, DAVE. A simple prolog techniques editor for novice users. In WIGGINS, GERAIN, MELLISH, CHRIS and DUNCAN, TIM, *3rd UK Annual Conference on Logic Programming*. Berlin, Springer-Verlag, 1991.

[Sadri et. al., 2001] SADRI, FARIBA, TONI, FRANCESCA, and TORRONI PAOLO. Logic Agents, Dialogues and Negotiation: An Abductive Approach. In STATHIS, KOSTAS and SCHROEDER, MICHAEL, the Symposium on Information Agents for E-Commerce AISB-01. York, United Kingdom, AISB, 2001.

[Sadri et. al., 2002] SADRI, FARIBA, TONI, FRANCESCA, and TORRONI, PAOLO. Dialogues for negotiation: Agent varieties and dialogue sequences. In MEYER, JOHN and TAMBE, MILIND, *Intelligent Agents VIII*, 8th International Workshop on Agent Theories, Architectures, and Language (ATAL 2001). Seattle, ATAL 2001 ,2002.

[Sagonas et al., 1994] SAGONAS, KONSTANTINOS, SWIFT, TERRANCE and WARREN, DAVID. XSB as an efficient deductive database engine. *In Proceedings of the SIGMOD '94 Proceedings of the 1994 ACM SIGMOD international conference on Management of data*. New York, ACM Press, 1994.

[Sauvage, 2004] SAUVAGE, SYLVAIN. Design Patterns for Multiagent Systems Design. In **MICAI'04: Advances in Artificial Intelligence, Third Mexican International Conference on Artificial Intelligence**. Mexico, Springer-Verlag, 2004.

[Shin et. al., 2005] SHIN, MICHAEL, ALEXANDER, LEVIS and LEE, WAGENHALS. Analyzing Dynamic Behavior of Large-Scale Systems through

Model Transformation. *The International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 1(15):2005:35-60.

[Shin et. al., 2003] SHIN, MICHAEL, ALEXANDER, LEVIS, LEE, WAGENHALS and DAESIK, KIM. Mapping of UML-based System Model to Design/CPN Model for System Model Evaluation. *In Proceedings of the Workshop on Compositional Verification of UML '03*. San Francisco, CA, 2003.

[Suriadi et al.,2009] SURIADI, SURIADI, YANG, CHUN, SMITH, JASON and FOO, ERNEST. Modeling and Verification of Privacy Enhancing Security Protocols. In BREITMAN KARIN and CAVALCANTI, ANA, *Formal Methods and Software Engineering*, 11th International Conference on Formal Engineering Methods ICFEM. Janeiro, Brazi, ICFEM, 2009.

[Sycara, 1989] SYCARA, KATIA. Argumentaion: planning other agents' paln. *In Proceeding of the 11th international joint conference on Artificial intelligence*. San Francisco, CA, USA, Morgan Kaufmann Publishers,1989.

[Tang and Parsons, 2006] TANG, YUQING and PARSONSL, SIMON. Argumentation-Based Multi-agent Dialogues for Deliberation. In PARSONS, SIMON, MAUDET, NICOLAS, MORAITIS, PAVLOS and RAHWAN, IYAD, *The Second international conference on Argumentation in Multi-Agent Systems (ArgMAS 2005)*. Heidelberg, Springer, 2006.

[Taylor and Wray, 2004] TAYLOR, GLENN and WRAY, ROBERT. Behavior Design Patterns: Engineering Human Behavior Models. *In Proceedings of the 13th Conference on Behavior Representation in Modeling and Simulation Conference (BRIMS)*. Arlington, Virginia, Curran Associates, 2004.

[Tolksdorf ,1998] TOLKSDORF ROBERT. Coordination Patterns of Mobile Information Agents. In KLUSCH, MATTHIAS and WEIß, GERHARD, *Cooperative Information Agents II*. Heidelberg, Germany, Springer-Verlag, 1998.

[Walton, 1998] WALTON, Douglas. *The New Dialectic: Conversational Contexts of Argument*. Canada, University of Toronto Press, Scholarly Publishing Division, 1998.

[Walton and Krabbe, 1995] WALTON, Douglas and KRABBE, Erik. *Commitment in Dialogue: Basic concept of interpersonal reasoning*. Albany, NY, USA, State University of New York Press, 1995.

[Walton, 1990] WALTON, DOUGLAS. What Is Reasoning? What Is An Argument?. *The Journal philosophy*, (87):1990: 399-419.

[Westergaard and Kristense, 2009] WESTERGAARD, MICHAEL AND KRISTENSEN, LARS. The access/CPN Framework: a Tool for Interacting with the CPN Tools Simulator*. In FRANCESCHINIS, GIULIANA and WOLF, KARSTEN, the 30th International Conference on Applications and Theory of Petri Nets (Petri Nets 2009). Heidelberg, Springer-Verlag, 2009.

[Westergaard and Verbeek, 2002] WESTERGAARD, MICHAEL AND VERBEEK H.M.W, *CPN Tools*, Eindhoven University of Technology, 2002, retrieved 2013, <http://cpntools.org/>.

[Willmott *et al.*, 2006] WILLMOTT, STEVEN, VREESWIJK, GERARD, CHESNEVAR, CARLOS, SOUTH, MATTHEW, MCGINNIS, JARRED, MODGIL, SANJAY, RAHWAN IYAD, REED CHRIS AND SIMARI, GUILLERMO. Towards an Argument Interchange Format for Multi-Agent Systems. In MAUDET, NICOLAS, PARSONS, SIMON and RAHWAN IYAD, *Argumentation in Multi-Agent Systems*, the 3th International Workshop on Argumentation in Multi-Agent Systems (ArgMAS2006). Japan, Springer, 2006.

[Ullman, 1998] ULLMAN, Jeffrey. *Elements of ML Programming*. Englewood Cliffs Prentice-Hall, 1998.