Bridging the Specification Protocol Gap in Argumentation

Ashwag Omar Magrhaby

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 SOFTWARE ENGINEERING DESIGN PATTERN ....................................................................... 14
  2.2.2 AGENT PROTOCOL DESIGN PATTERN ............................................................................... 14
  2.2.3 LOGIC PROGRAMMING PATTERNS (LOGIC PROGRAMMING TECHNIQUES) ......................... 17
    2.2.3.1 Prolog Programming Techniques ..................................................................................... 17
    2.2.3.2 Techniques editing ........................................................................................................... 19
    2.2.3.3 A Structural Synthesis System for LCC Protocols ............................................................ 20
    2.2.3.4 Comparing LCC-Argument Patterns with Logic Programming Techniques ......................... 22

2.3 VERIFICATION METHOD BASED ON SML AND COLOURED PETRI NET ................................. 23
  2.3.1 COLOURED PETRI NETS (CPNs) ........................................................................................ 24
    2.3.1.1 CPNs Model Elements .................................................................................................... 24
    2.3.1.2 CPNs Hierarchical Structure ........................................................................................... 28
    2.3.1.3 CPN Tool Components .................................................................................................. 30
  2.3.2 COMPARING OUR APPROACH WITH VERIFICATION APPROACHES BASED ON SML AND CPN MODEL ................................................................................................................ 36
    2.3.2.1 A Transformational Approach to CPN Model ................................................................. 366
    2.3.2.2 A Verification Method based on SML .............................................................................. 37
    2.3.2.3 LCC Verification Approaches based on Model Checking .................................................. 38

2.4 SUMMARY .................................................................................................................................... 39

CHAPTER 3: ARGUMENTATION, DIALOGUE GAMES AND MULTI-AGENT SYSTEMS .......................... 40
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>ARGUMENT AND ARGUMENTATION</td>
<td>40</td>
</tr>
<tr>
<td>3.2</td>
<td>DIALOGUE GAMES (ARGUMENTATION-BASED DIALOGUES)</td>
<td>41</td>
</tr>
<tr>
<td>3.3</td>
<td>ARGUMENTATION FOR AGENT COMMUNICATION</td>
<td>45</td>
</tr>
<tr>
<td>3.4</td>
<td>DIALOGUES GAMES TERMINOLOGY</td>
<td>46</td>
</tr>
<tr>
<td>3.5</td>
<td>TYPES OF DIALOGUES</td>
<td>49</td>
</tr>
<tr>
<td>3.6</td>
<td>EMBEDDED DIALOGUES</td>
<td>53</td>
</tr>
<tr>
<td>3.7</td>
<td>ARGUMENTATION SHARING PROBLEM AND ARGUMENT INTERCHANGE FORMAT</td>
<td>54</td>
</tr>
<tr>
<td>3.8</td>
<td>SUMMARY</td>
<td>64</td>
</tr>
<tr>
<td>4.1</td>
<td>AGENT PROTOCOL CONCEPTS FOR ARGUMENTATION BETWEEN TWO AGENTS</td>
<td>66</td>
</tr>
<tr>
<td>4.2</td>
<td>DIALOGUE INTERACTION DIAGRAM (AN EXTENSION OF AIF)</td>
<td>67</td>
</tr>
<tr>
<td>4.3</td>
<td>DIALOGUE INTERACTION DIAGRAM FOR EMBEDDING DIALOGUE</td>
<td>88</td>
</tr>
<tr>
<td>4.4</td>
<td>DIALOGUE INTERACTION DIAGRAM FOR ARGUMENTATION BETWEEN N-AGENT</td>
<td>102</td>
</tr>
</tbody>
</table>

**Chapter 4: Argument Specification Language**

4.1 AGENT PROTOCOL CONCEPTS FOR ARGUMENTATION BETWEEN TWO AGENTS

4.2 DIALOGUE INTERACTION DIAGRAM (AN EXTENSION OF AIF)

4.3 DIALOGUE INTERACTION DIAGRAM FOR EMBEDDING DIALOGUE

4.4 DIALOGUE INTERACTION DIAGRAM FOR ARGUMENTATION BETWEEN N-AGENT
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 AUTOMATED SYNTHESIS STEPS FOR GENERATING AGENT PROTOCOL BETWEEN TWO AGENTS ...........144
  5.2.2 AUTOMATED SYNTHESIS STEPS FOR GENERATING AGENT PROTOCOL FOR N-AGENT .........................144

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 DECLARATION OF COLOUR SETS AND FUNCTIONS ................................152
  6.1.2 GENERATION OF A CPN SUBPAGE ..........................................................152
  6.1.3 GENERATION OF A CPN SUPERPAGE ......................................................169

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

CHAPTER 7: DESIGN AND IMPLEMENTAION .......................................................189

7.1 ARCHITECTURE .............................................................................................189
  7.1.1 PART ONE: SYNTHESIS OF CONCRETE PROTOCOLS ARCHITECTURE ..................................................189
  7.1.2 PART TWO: VERIFICATION MODEL ARCHITECTURE ........................................189

7.2 AN EXAMPLE SCENARIO ................................................................................191

7.3 SUMMARY.......................................................................................................199

CHAPTER 8: EVALUATION AND DISCUSSION .....................................................202

8.1 SYNTHESIS OF CONCRETE PROTOCOLS ......................................................202
  8.1.1 RELATION BETWEEN DID AND AIF .......................................................202
  8.1.2 THE DIFFERENCE BETWEEN DID AND AIF EXTENSION .........................208
  8.1.3 DID LIMITATION .......................................................................................215
  8.1.4 LCC-ARGUMENT PATTERNS LIMITATIONS ........................................217

8.2 VERIFICATION METHOD BASED ON COLOURED PETRI NET AND SML ..........220
# List of Figures

1.1: SYSTEM ARCHITECTURE ........................................................................................................... 5  
2.1: THE SLAVE CLASS .................................................................................................................. 16  
2.2: CPNS MODEL ELEMENTS EXAMPLE .................................................................................... 25  
2.3: A HIERARCHICAL 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 (PROPOSALSENDERPROPOSAL AND PROPSALRECEIVERRECEIVER ROLES) ................................................................. 131
5.6: STRUCTURE (REPLYTOPROPSALSENDER AND REPLYPROMPROPSALRECEIVERPROPOSAL ROLES) ................................................................. 132
5.7: STRUCTURE (RESULTSENDERPROPOSAL / SENDREACHAGREEMENTPROPOSALDIVIDEROUPROPOSAL AND RESULTRECEIVERROLES) ................................................................. 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 GENERATELCCPROTOCOL TOOL .................................... 192
7.4: CREATE NEW DIALOGUE INTERACTION DIAGRAM EXAMPLE (CLAIM LOCUTION ICON) .... 193
A.5(B): GENERATED LCC PROTOCOL FOR NEGOTIATION DIALOGUE ................................................. 248
A.6: THE REQUESTSENDER CPN SUBPAGE .................................................................................. 251
A.7: THE REQUESTRECEIVER CPN SUBPAGE ................................................................................. 251
A.8: THE REPLYTOREQUESTSENDER CPN SUBPAGE ................................................................. 252
A.9: THE REPLYTOREQUESTSENDER CPN SUBPAGE ................................................................. 253
A.10: THE REPLYTOCHALLENGESENDER CPN SUBPAGE .......................................................... 248
A.11: THE REPLYTOCHALLENGERECEIVER CPN SUBPAGE ....................................................... 253
A.12: THE REPLYTOJUSTIFYSENDER CPN SUBPAGE .................................................................. 254
A.13: THE REPLYTOJUSTIFYRECEIVER CPN SUBPAGE .............................................................. 254
A.14: THE REPLYTOPROMISESENDER CPN .................................................................................. 255
A.15: THE REPLYTOPROMISERECEIVER 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: TERMINATION 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
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.8(A): GENERATED LCC PROTOCOL FOR PERSUSSION DIALOGUE (PART 1)</td>
<td>331</td>
</tr>
<tr>
<td>C.8(B): GENERATED LCC PROTOCOL FOR PERSUSSION DIALOGUE (PART 2)</td>
<td>332</td>
</tr>
<tr>
<td>C.8(C): GENERATED LCC PROTOCOL FOR PERSUSSION DIALOGUE (PART 3)</td>
<td>333</td>
</tr>
<tr>
<td>C.8(D): GENERATED LCC PROTOCOL FOR PERSUSSION DIALOGUE (PART 4)</td>
<td>334</td>
</tr>
<tr>
<td>C.8(E): GENERATED LCC PROTOCOL FOR PERSUSSION DIALOGUE (PART 5)</td>
<td>335</td>
</tr>
<tr>
<td>C.8(F): GENERATED LCC PROTOCOL FOR PERSUSSION DIALOGUE (PART 6)</td>
<td>336</td>
</tr>
<tr>
<td>C.9 (A): STEP 2 OF PROTOCOL GENERATION (MATCHING THE MOVE-TO-DIALOGUE PATTERN)</td>
<td>337</td>
</tr>
<tr>
<td>C.9 (B): STEP 2 OF PROTOCOL GENERATION (MATCHING THE MOVE-TO-DIALOGUE PATTERN)</td>
<td>338</td>
</tr>
<tr>
<td>C.9 (C): STEP 3 (PART 1) OF PROTOCOL GENERATION (MATCHING THE REWRITING METHODS OF THE RECURS-TO-N-DIALOGUE PATTERN)</td>
<td>339</td>
</tr>
<tr>
<td>C.9 (D): STEP 3 (PART 2) OF PROTOCOL GENERATION (MATCHING THE REWRITING METHODS OF THE RECURS-TO-N-DIALOGUE PATTERN)</td>
<td>340</td>
</tr>
<tr>
<td>C.9 (E): STEP 3 (PART 3) OF PROTOCOL GENERATION (MATCHING THE REWRITING METHODS OF THE RECURS-TO-N-DIALOGUE PATTERN)</td>
<td>341</td>
</tr>
<tr>
<td>C.10: THE CLAIMSENDER&lt;sub&gt;r&lt;/sub&gt; CPN SUBPAGE</td>
<td>342</td>
</tr>
<tr>
<td>C.11: THE CLAIMRECEIVER&lt;sub&gt;0&lt;/sub&gt; CPN SUBPAGE</td>
<td>343</td>
</tr>
<tr>
<td>C.12: THE REPLYTOCLAIMSENDER&lt;sub&gt;r&lt;/sub&gt; CPN SUBPAGE</td>
<td>343</td>
</tr>
<tr>
<td>C.13: THE REPLYTOCLAIMRECEIVER&lt;sub&gt;r&lt;/sub&gt; CPN SUBPAGE</td>
<td>344</td>
</tr>
<tr>
<td>C.14: THE REPLYTOWHYSENTER&lt;sub&gt;r&lt;/sub&gt; CPN SUBPAGE</td>
<td>344</td>
</tr>
<tr>
<td>C.15: THE REPLYTOWHYRECEIVER&lt;sub&gt;r&lt;/sub&gt; CPN SUBPAGE</td>
<td>345</td>
</tr>
<tr>
<td>C.16: THE REPLYTOARGUESENDER&lt;sub&gt;r&lt;/sub&gt; CPN SUBPAGE</td>
<td>345</td>
</tr>
<tr>
<td>C.17: THE REPLYTOARGUERECEIVER&lt;sub&gt;r&lt;/sub&gt; CPN SUBPAGE</td>
<td>346</td>
</tr>
<tr>
<td>C.18: THE PROTOCOL CPN SUPERPAGE</td>
<td>347</td>
</tr>
<tr>
<td>C.19: THE STATE SPACE GRAPH</td>
<td>348</td>
</tr>
<tr>
<td>C.20: DIALOGUE OPENING PROPERTY PAGE</td>
<td>351</td>
</tr>
<tr>
<td>C.21: TERMINATION OF A DIALOGUE PROPERTY PAGE</td>
<td>352</td>
</tr>
<tr>
<td>C.22: TURN TAKING BETWEEN AGENTS PROPERTY PAGE</td>
<td>352</td>
</tr>
<tr>
<td>C.23: MESSAGE SEQUENCING PROPERTY PAGE</td>
<td>352</td>
</tr>
<tr>
<td>C.24: RECURSIVE MESSAGE PROPERTY PAGE</td>
<td>353</td>
</tr>
</tbody>
</table>
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 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...
Bridging the Specification Protocol Gap in Argumentation

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 synthesize 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
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 is 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 process\(^1\) and declarative\(^2\) 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 choreography 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."
Bridging the Specification Protocol Gap in Argumentation

<table>
<thead>
<tr>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework ::= {Clause,…}</td>
</tr>
<tr>
<td>Clause ::= Agent ::= Dn</td>
</tr>
<tr>
<td>Agent ::= a(Role, Id)</td>
</tr>
<tr>
<td>Dn ::= Agent</td>
</tr>
<tr>
<td>Message ::= M =&gt; Agent</td>
</tr>
<tr>
<td>Constraint ::= Term</td>
</tr>
<tr>
<td>Role ::= Term</td>
</tr>
<tr>
<td>M ::= Term</td>
</tr>
<tr>
<td>Term ::= Constant (Argument,……)</td>
</tr>
<tr>
<td>Id ::= Constant</td>
</tr>
<tr>
<td>Constant ::= Character sequence made up of letters or numbers beginning with a lowercase letter</td>
</tr>
<tr>
<td>Variable ::= Character sequence made up of letters or numbers beginning with an uppercase character</td>
</tr>
<tr>
<td>Argument ::= Term</td>
</tr>
</tbody>
</table>

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 $C$ (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 ($M \Rightarrow Agent$), and receive a message from another agent ($M \Leftarrow Agent$). There are two types of constraints over the messages exchanged: pre-condition and post-condition. Pre-conditions ($M \Rightarrow Agent \Leftarrow C$) specify the required conditions for an agent to send a message. Post-conditions ($C \Leftarrow M \Leftarrow Agent$) 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:
Bridging the Specification Protocol Gap in Argumentation

### Chapter 2 Background and Literature Review

This is read as: role $R_1$ of agent $P$ sends a claim message to the role $R_2$ of agent $O$ and then role $R_2$ of agent $O$ receives the claim message from role $R_1$ of agent $P$. Then $P$ changes its role to $R_3$ and $O$ changes its role to $R_4$.

**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 $R$.

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.

\[
\begin{align*}
\text{a(auctioneer(Product,Bidders), A) ::=} & \\
\text{a(caller(Product, Bidders),A) then} & \\
\text{a(waiter(Bidders, Bids, curwinner(nul, 0),A) then} & \\
\text{sold(Product,Price) => a(bidder,BW) \gets curwinner(WB, Price) = Winner.} & \\
\text{a(caller(Product,Bidders), A) ::=} & \\
\text{null \gets Bidders = [ ] \%no bidders left} & \\
\text{or} & \\
\text{invite\_bid(Product) => a(bidder, BH) \gets Bidders = [BH|BT]} & \\
\text{then} & \\
\text{a(caller(Product, BT), A). \%recursion} & \\
\text{a(waiter(Bidders, Bids, curwinner(WinBidder, WinBid), A) ::=} & \\
\text{null \gets allarrived(Bids, Bidders) and Winner = curwinner(WinBidder, WinBid) or} & \\
\text{null \gets timeout( ) and Winner = curwinner(WinBidder, WinBid) or} & \\
\text{bid(Product,Offer) <= a(bidder, B) then} & \\
\text{a(waiter(Bidders, [B|Bids], curwinner(B, Offer), A) \gets Offer > WinBid or} & \\
\text{a(waiter(Bidders, [B|Bids], curwinner(WinBidder, WinBid), A) or} & \\
\text{a(waiter(Bidders, Bids, curwinner(WinBidder, WinBid), A) \gets sleep(1000).} & \\
\text{a(bidder, B) ::=} & \\
\text{invite\_bid(Product) <= a(caller, A) then} & \\
\text{bid(Product, Offer) => a(waiter, A) \gets bid\_at(Product, Offer) then} & \\
\text{sold(Product, Price) <= a(auctioneer(Product,Bidders), A).} & \\
\end{align*}
\]

The auctioneer role of agent \( A \) has two input parameters: \textit{Product} to sell and the list of \textit{Bidders}. The auctioneer role starts by changing its role to caller. The caller role of agent \( A \) recurses over the \textit{Bidders} list. If the list is empty, it returns null, otherwise, it
Bridging the Specification Protocol Gap in Argumentation

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

Chapter 2 Background and Literature Review
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 Aglets\(^4\) and Voyager\(^5\) 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:


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 "coordinating 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:

\[
\begin{align*}
\text{rev}(\ [\] , R , R ). \\
\text{rev}(\ [H|T] , R0 , R) :- \\
\text{rev}(\ T, [H|R0] , R).
\end{align*}
\]

This predicate consists of two parts:

1. A part which performs the recursion down the list:

\[
\begin{align*}
\text{rev}([],...) \\
\text{rev}([H|T],...) :- \\
\text{rev}(T,...).
\end{align*}
\]

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:

\[
\begin{align*}
\text{rev}([],R,R) \\
\text{rev}(...,R0,R) :- \\
\text{rev}(...,[H|R0],R).
\end{align*}
\]

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 et 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: Roberson'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 predicates\(^6\) and doubly recursive clauses\(^7\) along 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 if 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:

\[
\begin{align*}
a(R,X) :: & \\
& \begin{cases}
\text{<def>} & \text{then } a(R,X) \\
\text{or} & \\
\text{null} & \text{<con>}
\end{cases}
\end{align*}
\]

This example represents a general recursive clause that can be applied to different clauses. \(R\) represents role name, \(X\) represents agent identifier, \(<\text{def}>\) represents unspecified definition, and \(<\text{con}>\) 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:

\[
\begin{align*}
a(F(A_1...A_n),X) :: & \\
& \begin{cases}
\text{<def>} & \text{then } a(F(A_1...A_{n-1},A_n'),X) \\
\text{or} & \\
\text{null} & \text{<con>}
\end{cases}
\end{align*}
\]
This example represents a recursive clause in more detail than the *Skeletal* example. $F$ represents the role name and $A1...An$ 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 $<\text{def}>$ represents an unspecified definition.

There are two main differences between Robertson's editor and Ted, and the SDE editor. Firstly, Robertson'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 language\(^8\) (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 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 Nets\(^9\) (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 \((Open \text{ and } P)\) and two output places \((claim1 \text{ and } ChangeRole1)\):

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 Tool\(^{10}\) "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/](http://cpntools.org/)

Chapter 2 Background and Literature Review
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 technique\textsuperscript{11} to analyse the behaviours of communication systems [Jensen \textit{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 \textit{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 \textit{CPN workspace}. For more information about the CPN Tool and the construction of the CPN model see [Jensen \textit{et al.}, 2007; Kristensen \textit{et al.}, 1998].

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

<workspaceElements>
  <generator tool="CPN Tools" version="2.9.11" format="6"/>
  <cpnet>
    <globbox>
      <color id="ID1424220943">
        <id>TOPIC</id> <string/>
      </color>
      <var id="ID1424221049">
        <type>
          <id>TOPIC</id>
        </type>
        <id> t </id>
      </var>
    </globbox>
    <page id="ID6">
      <pageattr name="Starting Page"/>
      <place id="ID1424211163">
        <text>open</text>
        <type id="ID1424211164">
          <text tool="CPN Tools version="2.9.11">TOPIC</text>
        </type>
        <initmark id="ID1424211165">
          <text tool="CPN Tools version="2.9.11">1"The car is safe"</text>
        </initmark>
        <port id="ID1424205036" type="In">
          <text>change</text>
        </port>
      </place>
    </page>
  </cpnet>
</workspaceElements>
Figure 2.5 (b): CPNXML File Structure Example
by the CPN tool. We removed some CPNXML tags which are related to the background colour, foreground colour and element position).

**State Space Techniques**

The state space method of the CPN tool allows to model check the correctness of CPN models (concurrent systems) [Jensen et al., 2006]. It is used to verify concurrent systems (in a mathematical way) by computing all reachable states and state changes of this system. By constructing the state space, it is possible to demonstrate that certain properties are satisfied or that certain undesired properties are absent by using a set of CPN SML functions. An example of such properties is the guarantee of terminating a specific service when reaching a given state and the possibility of constantly reaching a given state [Kristensen et al., 1998].

A state space is a directed graph with reachable marking nodes and binding element arcs. These arcs are used to connect two nodes together and demonstrate that the occurrence of binding specific elements leads to the occurrence of the next node. Figure 2.6 illustrates one example of a state space graph. This graph has:
Bridging the Specification Protocol Gap in Argumentation

1 (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 XML file) [Jensen, 1992; Jensen et al., 2007; Kristensen et al., 1998]. The tool was tested by running the Design/CPN tool simulator for analysing the dynamic behavior of two large–scale UML systems:

1 (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 (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)\textsuperscript{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:

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

Chapter 2 Background and Literature Review
Bridging the Specification Protocol Gap in Argumentation

(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:
Bridging the Specification Protocol Gap in Argumentation

(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.
Bridging the Specification Protocol Gap in Argumentation

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:**

<table>
<thead>
<tr>
<th>Locutions (speech acts)</th>
<th>Meaning of Locution</th>
</tr>
</thead>
<tbody>
<tr>
<td>claim(T)</td>
<td>Making a claim</td>
</tr>
<tr>
<td>why(T)</td>
<td>Asking grounds for a claim</td>
</tr>
<tr>
<td>concede(T)</td>
<td>Conceding (accepting ) a claim</td>
</tr>
<tr>
<td>argue(Pre, T)</td>
<td>Offering grounds for a claim</td>
</tr>
<tr>
<td>retract (T)</td>
<td>retracting (withdrawing) a claim</td>
</tr>
</tbody>
</table>

(2) **Commitment Store:** There is one CS for each participant: \{CSₚ, CSₒ\}
### Commitment rules:

<table>
<thead>
<tr>
<th>Locutions</th>
<th>Commitment rules</th>
<th>Meaning of Commitment rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>claim(T)</td>
<td>CS ∪ {T}</td>
<td>The effect of a 'claim' move is always to add topic 'T' to the mover's commitments 'CS'</td>
</tr>
<tr>
<td>why(T)</td>
<td>CS</td>
<td>The mover's commitments remain unchanged</td>
</tr>
<tr>
<td>concede(T)</td>
<td>CS ∪ {T}</td>
<td>The effect of a 'concede' move is always to add topic 'T' to the mover's commitments 'CS'</td>
</tr>
<tr>
<td>argue(Pre, T)</td>
<td>CS ∪ {T} ∪ {Pre}</td>
<td>The effect of an 'argue' move is always to add topic 'T' and premise 'Pre' to the mover's commitments 'CS'</td>
</tr>
<tr>
<td>retract(T)</td>
<td>CS - {T}</td>
<td>The effect of a 'retract' move is always to remove topic 'T' from the mover's commitments 'CS'</td>
</tr>
</tbody>
</table>

### Pre-conditions

<table>
<thead>
<tr>
<th>Locutions</th>
<th>Pre-conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>claim(T)</td>
<td>There are no special pre-conditions to starting a persuasion dialogue (for the utterance of 'claim' locution).</td>
</tr>
<tr>
<td>why(T)</td>
<td>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).</td>
</tr>
<tr>
<td>concede(T)</td>
<td>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 '~T'.</td>
</tr>
<tr>
<td>argue(Pre, T)</td>
<td>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 promises 'Pre' in his 'KB' or 'CS' to support a claim 'T'.</td>
</tr>
<tr>
<td>retract(T)</td>
<td>In order for the speaker to retract a claim, he must have committed to it. He also must not be able to find a promises 'Pre' to support a claim 'T'.</td>
</tr>
</tbody>
</table>

### Structural rules:

<table>
<thead>
<tr>
<th>Locutions</th>
<th>Structural rules</th>
<th>Meaning of Structural rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>claim(T)</td>
<td>why(T) or concede(T)</td>
<td>After a 'claim' move, the Next player can select either 'why' or 'concede' locutions</td>
</tr>
<tr>
<td>why(T)</td>
<td>argue(Pre) or retract(T)</td>
<td>After an 'why' move, the Next player can select either 'argue' or 'retract' locutions</td>
</tr>
<tr>
<td>concede(T)</td>
<td>No reply</td>
<td>After a 'concede' move, the Next player cannot make a move.</td>
</tr>
<tr>
<td>argue(Pre, T)</td>
<td>why(Pre), argue(Def) or concede(T)</td>
<td>After an 'argue' move, the Next player can select 'why', 'argue' or 'concede' locutions</td>
</tr>
<tr>
<td>retract(T)</td>
<td>No reply</td>
<td>After a 'retract' move, the Next player cannot make a move</td>
</tr>
</tbody>
</table>
(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].
## 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 a reasonable 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.
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:

![Diagram of determining the type of dialogue]
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
Bridging the Specification Protocol Gap in Argumentation

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,
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 ≥ 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 \( \neg \text{flies}(P) \) (I1-node) and one for \( \text{flies}(P) \) (I2-node);

2. The argument for \( \neg \text{flies}(P) \) 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 \( \text{flies}(P) \) is composed of one RA2-node, namely defeasible Modus Ponens and two child nodes (premises);
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 \( \neg \text{flies}(P) \) 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 \( \text{flies}(P) \) is weak because the premises (I5-node and I6-node) support it with only 0.8 degree (a low probability). So, \( \neg \text{flies}(P) \) is preferred to the argument for \( \text{flies}(P) \). That is why the intermediate Preference Application node (PA-node that defines the value judgments or preference orderings of argument), namely Logical attack, links \( \neg \text{flies}(P) \) (I1-node) to \( \text{flies}(P) \) (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 ~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 ~T.

(9) A1 concede 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 ASPIC\(^{15}\) (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](http://www.arg.dundee.ac.uk/?page_id=197)
Figure 3.5: Illustration of the Link between Argument (AIF Nodes) and Dialogue Games (AIF\textsuperscript{+} 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) $A1$ concede 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.
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.
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; Starting rules; Termination 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.
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.

```
claim(T)
```

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\text{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 syntheses).

```
claim(T)
\hline
claimSender
\hline
KB\text{Sender}, CS\text{Sender},
CS\text{Receiver}, T
\hline
ID\text{Sender}
```

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\text{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 syntheses).

```
claim(T)
\hline
claimSender & claimReceiver
\hline
KB\text{Sender}, CS\text{Sender},
CS\text{Receiver}, T & KB\text{Receiver}, CS\text{Receiver},
CS\text{Sender}
\hline
ID\text{Sender} & ID\text{Receiver}
```
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.

```
<table>
<thead>
<tr>
<th>claim(T)</th>
<th>claimSender</th>
<th>claimReceiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>KB_Sender,CS_Sender, CS_Receiver, T</td>
<td>KB_Receiver, CS_Receiver, CS_Sender</td>
<td></td>
</tr>
<tr>
<td>ID_Sender</td>
<td>ID_Receiver</td>
<td></td>
</tr>
</tbody>
</table>
```

Starting Locution

```
addToCS(T,CSSender)
```

Starting Locution

```
addToCS(T,CSSender)
```

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.
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...
Bridging the Specification Protocol Gap in Argumentation

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

In this example, a dialogue always starts with a \textit{claim} and ends with a \textit{concede} or \textit{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 \textit{claim}(T) locution if he is able to satisfy the \textit{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: \textit{why}(T) or \textit{concede}(T). \(O\) will make his choice using the pre-conditions which appear in the rhombus shape. In order to choose \textit{concede}(T), \(O\) must be able to satisfy the four pre-conditions which connect with concede:

1) \textit{findTopicInKB}(T, KB_{O}) which returns true if agent \(O\) is able to find \(T\) in its knowledge base \(KB_{O}\); 2) \textit{notFindTopicInCS}(T,CS_{O}) which returns true if agent \(O\) is not able to find \(T\) in its commitment store \(CS_{O}\); 3) \textit{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) \textit{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 \textit{concede}(T) because the explained pre-conditions are not satisfied, then \(O\) will send \textit{why}(T). After that, the turn switches to \(P\), and so on. The argument terminates once \(P\) or \(O\) sends \textit{concede or retract} locutions.

\textbf{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
Dialogue takes place between two agents, $P$ and $O$.

$P$ has $KB_P$ and $CS_P$, and $O$ has $KB_O$ and $CS_O$.

Initially the $CS_P$ and $CS_O$ are empty.

$P$ and $O$ can access both $CS_P$ and $CS_O$.

$P$ opens the discussion by sending $claim(\text{"My car is safe"})$.

$O$ 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, $O$ 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 $why(\text{"Why is your car safe"})$ locution.

$P$ 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 $argue(\text{"Since it has an airbag"})$ locution.

$O$ 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 $argue(\text{"Your car is not safe since the newspapers recently reported on airbags expanding without cause"})$ locution.

$P$ finds a counterargument for $O$'s argument and sends an $argue(\text{"Newspaper reports are very unreliable sources of technological information"})$ locution.

$O$ finds a counterargument for $P$'s argument and sends an $argue(\text{"Still your car is not safe, since its maximum speed is very high"})$ locution.

$P$ 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 $retract(\text{"My car is safe"})$ locution.
Figure 4.4: The Complex Car Safety Example

KB_P = {My car safe, My car has an airbag, car has an airbag → car is safe, newspapers are unreliable sources}

KB_O = {Your car is not safe, newspapers reported on airbags expanding → car is not safe, car has high maximum speed → car is not safe}
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:

$$(L, \text{Players}, CS, KB, Roles, Acts, ActType, Replies, Moves, LegalMoves)$$

**Definition 2: Topic**

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

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

**Definition 3: Players**

Players = \{player$_1$, player$_2$\}
Where,

- Each player player_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\(^{17}\).

- Each player player_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 4: Commitment Store**

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

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

Initially \( CS(\text{player}_i, M_1) = \emptyset \), where \( i = 1 \) or \( 2 \)

**Definition 5: Knowledge Base**

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

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

**Definition 6: Roles**

Roles = \{r_1, r_2, \ldots, 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)

\[^{17}\text{For any set } S:\]

\[
\wp(S) = \text{the powerset of any set } S
\]

\[
\subseteq = \text{a partial order on the set } \wp(S) \text{ of all subsets of } S.
\]
Definition 7: Acts

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

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

Definition 8: ActType

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

\[ \text{ActType} : \text{Acts} \rightarrow \varnothing(\text{Types}) \]

Where,

- Types = {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.

\[ \text{Replies} : \text{Acts} \rightarrow \varnothing(\text{Acts}) \]

For instance Replies(claim(T)) = \{why(T),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.

\[ \text{PreC} : \text{Acts} \times \varnothing(\text{args}(L)) \times \varnothing(\text{args}(L)) \rightarrow \text{Boolean} \]
For example:

\[ \text{PreC} (\text{claim(\(\alpha\)), CS(player_1, M_t), KB(player_1)}) = \exists \ \alpha \in \text{CS(player}_1, M_t) \cup \text{KB(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.

\[ \text{PostC}: \text{Acts} \times \varnothing(\text{args(L)}) \times \varnothing(\text{args(L)}) \rightarrow \text{Boolean} \]

**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**

'\text{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**: \( \text{MoveSeq} \times \phi(\text{args}(L)) \times \phi(\text{args}(L)) \rightarrow \phi(\text{Moves}) \)

**Rule 1: (Start a Dialogue)**

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

\[
\text{LegalMovesTwo}(\emptyset, \text{CS}1, \text{CS}2) = \{ \text{M}1 \}
\]

Where,

- \( \text{M}1= (\text{player}_1, \text{act}_1, \text{null}, \text{player}_2, \text{sRole}_1, \text{rRole}_1) \),
- \( \text{ActType}(\text{act}_1) = \text{Starting} \),
- \( \text{PreC}(\text{act}_1, \text{KB}_1, \text{CS}_1) = \text{true} \), 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:

\[
\text{LegalMovesTwo}(\text{[M}_1,\text{M}_2,\ldots,\text{M}_n], \text{CS}1, \text{CS}2) = \emptyset
\]

if

- \( \text{M}_n= (\text{player}_n, \text{act}_n, \text{M}_{n-1}, \text{null}, \text{sRole}_n, \text{rRole}_n) \),
- \( \text{ActType}(\text{act}_n) = \text{Termination} \),
- \( \text{PreC}(\text{act}_n, \text{KB}_n, \text{CS}_n) = \text{true} \), 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:
  - \( n \neq m \)
Rule 3: (Reply to an Agent’s Move)

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

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

if

- player_i ≠ player_j
- M_t = (player_i, act_t, M_{t-1}, player_j, sRole_t, rRole_t),
- ActTypes(act_t) ∈ \{Starting, Intermediate\},
- PreC(\text{act}_{t+1}, KB_i, CS_i) = \text{true}, where:
  - KB(player_i) = KB_i
  - CS(player_i) = CS_i
- M_{t+1} = (player_j, act_{t+1}, M_t, player_i, sRole_{t+1}, rRole_{t+1}),
- act_{t+1} ∈ \text{Replies}(act_t) \quad (M_{t+1} \text{ replies to } M_t),
- PostC(\text{act}_{t+1}, KB_j, CS_j) = \text{true}, where:
  - KB(player_j) = KB_j
  - CS(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}.
Bridging the Specification Protocol Gap in Argumentation

Note that in order to send $M_{t+1}$, player, must satisfy $\text{PreC}$ and after $M_{t+1}$, player $j$ must satisfy $\text{PostC}$.

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):**

<table>
<thead>
<tr>
<th>Act</th>
<th>ActType (Act)</th>
</tr>
</thead>
<tbody>
<tr>
<td>claim</td>
<td>{Starting}</td>
</tr>
<tr>
<td>why</td>
<td>{Intermediate}</td>
</tr>
<tr>
<td>concede</td>
<td>{Termination}</td>
</tr>
<tr>
<td>argue</td>
<td>{Intermediate}</td>
</tr>
<tr>
<td>retract</td>
<td>{Termination}</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Act</th>
<th>Replies(Act)</th>
</tr>
</thead>
<tbody>
<tr>
<td>claim(T)</td>
<td>{why(T), concede(T)}</td>
</tr>
<tr>
<td>why(T)</td>
<td>{argue(Pre), retract(T)}</td>
</tr>
<tr>
<td>concede(T)</td>
<td>Ø</td>
</tr>
<tr>
<td>argue(Pre,T)</td>
<td>{why(Pre), argue(Def,T'), concede(T)}</td>
</tr>
<tr>
<td>retract(T)</td>
<td>Ø</td>
</tr>
</tbody>
</table>
(5) PreC(Act, KB, CS):

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

<table>
<thead>
<tr>
<th>Act</th>
<th>PreC(Act, KB, CS)</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>claim(T)</td>
<td>addTopicToCS(T, CS_P) = true</td>
<td>• addTopicToCS function always returns true and results in agent P adding T to its commitment store CS_P.</td>
</tr>
<tr>
<td>why(T)</td>
<td>notFindTopicInKB(T, KB_P) = true and notFindTopicInCS(T, CS_P) = true</td>
<td>• notFindTopicInKB function returns true if agent P is not able to find T in its Knowledge Base KB_P. • notFindTopicInCS function returns true if agent P is not able to find T in its Commitment Store CS_P.</td>
</tr>
<tr>
<td>concede(T)</td>
<td>findTopicInKB(T, KB_P) = true and notFindTopicInCS(T, CS_P) = true and notFindOppTopicInCS(not(T), CS_P) = true and addTopicToCS(T, CS_P) = true</td>
<td>• findTopicInKB function returns true if agent P is able to find T in its Knowledge Base KB_P. • notFindTopicInCS function returns true if agent P is not able to find T in its Commitment Store CS_P. • notFindOppTopicInCS function returns true if agent P is not able to find the opposite of T (not(T)) in its commitment store CS_P. • addTopicToCS function always returns true and results in agent P adding T to its commitment store CS_P.</td>
</tr>
<tr>
<td>argue(Pre, T)</td>
<td>Pre = findPremise(T, KB_P, CS_P) = true and addPreToCS(T, Pre, CS_P) = true</td>
<td>• Where T = topic and Pre = Promises which is used to support a claim (Topic) • findPremise function returns true if agent P is able to find Pre either in its knowledge base KB_P or its commitment store CS_P. • addPreToCS function always returns true and results in agent P adding T and Pre to its commitment store CS_P.</td>
</tr>
</tbody>
</table>
Bridging the Specification Protocol Gap in Argumentation

Chapter 4: Argument Specification Language

<table>
<thead>
<tr>
<th>Act</th>
<th>PreC(Act, KB, CS)</th>
<th>Note</th>
</tr>
</thead>
</table>
| argue(Def, T')       | Def = findDefeats(T, Pre, KB\textsubscript{p}, CS\textsubscript{p}) = true and addDefeatToCS (T', Def, CS\textsubscript{p}) = true | - Where Def = Defeat an argument which is used to attacking an argument (T or Pre) with a counterarguments (Def)  
- findDefeats function returns true if agent \textit{P} is able to find Def either in its knowledge base KB\textsubscript{p} or its commitment store CS\textsubscript{p}.  
- addDefeatToCS function always returns true and results in agent \textit{P} adding Def and T'(T or Pre) to its commitment store CS\textsubscript{p}. |
| retract(T)           | cannotFindPreInKB(T, KB\textsubscript{p}) = true and findTopicInCS (T, CS\textsubscript{p}) = true and subtractFromCS(T, CS\textsubscript{p}) = true | - cannotFindPreInKB function returns true if agent \textit{P} is not able to find any promises (pre) in its knowledge base KB\textsubscript{p} to support a claim (T).  
- findTopicInCS function returns true if agent \textit{P} is able to find T in its Commitment Store CS\textsubscript{p}.  
- subtractFromCS function always returns true and results in agent \textit{P} subtract T from its commitment store CS\textsubscript{p}. |

(6) \textbf{LegalMovesTwo} (M\textsubscript{1}, CS\textsubscript{p}, CS\textsubscript{O})

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

- Dialogue begins by making a \textit{claim} move

    \quad M\textsubscript{1} = initial move, ActType(Act(M\textsubscript{1})) = \textit{Starting} and Act(M\textsubscript{1}) = \{claim\}

- In the persuasion dialogue, the argument terminates once agents send a \textit{concede} or \textit{retract} locution. In other words, both \textit{concede} and \textit{retract} \epsilon 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 \textit{why} and \textit{argue} $\in \{\text{Intermediate}\}$. 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 \( \text{Player} = P \),
- else \( \text{NextPlayer} = O \) \text{ iff } \( \text{Player} = P \)
  and \( \text{NextPlayer} = P \) \text{ iff } \( \text{Player} = O \)

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

\textbf{4.3 Dialogue Interaction Diagram for Embedding Dialogue}

\textbf{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).

\textbf{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.

\textbf{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
Bridging the Specification Protocol Gap in Argumentation

- **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={γ}). Then, if an agent wants to open a sub-dialogue, he can make a move with δ where δ= β₁,β₂,β₃,...,βₙ → γ.

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 $AS_{P''} = \{KB_{P''}, CS_{P''}\}$

• $P$ argumentation system $AS_P = \{KB_P, CS_P\}$

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

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

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

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

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

In the inquiry dialogue the Replies rules are as follows:

<table>
<thead>
<tr>
<th>Act</th>
<th>Replies(Act)</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>open(γ)</td>
<td>{assert (Φ,γ), open(δ), close(γ)}</td>
<td>• (Φ,γ) is an argument, Φ is the argument support and γ is the argument claim</td>
</tr>
<tr>
<td>close(γ)</td>
<td>{assert (Φ2,X), open(δ2), close(γ)}</td>
<td>• When ActType (close) = {Intermediate}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• X variable in assert(Φ2,X) represents either γ or β_n ∈ δ.</td>
</tr>
<tr>
<td>close(γ)</td>
<td>Ø</td>
<td>• when ActType (close) = {Termination}</td>
</tr>
</tbody>
</table>
Bridging the Specification Protocol Gap in Argumentation

### Act

<table>
<thead>
<tr>
<th>Act</th>
<th>Replies(Act)</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert(Φ,γ)</td>
<td>{assert (Φ2, X2), open(δ2), close(γ)}</td>
<td>• X2 variable in assert (Φ2, X2) represents either γ or δ</td>
</tr>
<tr>
<td>open(δ)</td>
<td>{assert (Φ2, X2), open(δ2), close(γ), subclose(δ)}</td>
<td></td>
</tr>
<tr>
<td>assert(Φ2, X2)</td>
<td>{assert (Φ3, X3), open(δ3), close(γ), subclose(δ), subclose(Z)}</td>
<td>• Z variable in subclose(Z) represents either δ or δ2.</td>
</tr>
<tr>
<td>open(δ2)</td>
<td>{assert (Φ2, X2), open(δ2), close(γ), subclose(Z)}</td>
<td>• Z variable in subclose(Z) represents either δ or δ2.</td>
</tr>
<tr>
<td>subclose(δ)</td>
<td>{assert (Φ2, X2), open(δ2), close(γ), subclose(Z), subclose(δ)}</td>
<td>• subclose(δ) after subclose(δ) ends sub-dialogue δ</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Act</th>
<th>PreC(Act, KB, CS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open(γ)</td>
<td>findInKB(γ, KBP&quot;) = true and emptyCS(CSP&quot;) = true and addToQueryStore (QS, γ) = true and addToOpenDialogue (γ, OpenD) = true</td>
</tr>
</tbody>
</table>

when ActType (open) = {Starting}, four functions must return true:

- findInKB function returns true if agent P" is able to find γ in its Knowledge Base KBP".
- emptyCS function returns true if agent P" Commitment Store CSP" is empty.
- addToQueryStore function always returns true and results in agent P" adding γ to dialogue Question Store QS.
- addToOpenDialogue function always returns true and results in agent P" adding γ to Open Dialogue list OpenD.
<table>
<thead>
<tr>
<th>Act</th>
<th>PreC(Act,KB,CS)</th>
<th>Note</th>
</tr>
</thead>
</table>
| 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(δ,OpenD) = true and addToSubD(δ,γ ,SubD) = true | when ActType (open) = \{Intermediate\}, seven functions must return true:  
• `isRelationship` function returns true if agent $P''$ is able to find a relation between $δ$ and $γ$.  
• `findInQS(QS,γ)` function returns true if agent $P''$ is able to find $γ$ in the dialogue Question Store $QS$.  
• `findInKB(δ,KB)` function returns true if agent $P''$ is able to find $δ$ in its Knowledge Base $KB_{P'}$.  
• `notFindInQS` function returns true if agent $P$ is not able to find $δ$ in escape $QS$.  
• `addToQueryStore` function always returns true and results in agent $P''$ adding $δ$ to Open Dialogue list $OpenD$.  
• `addToSubD` function always returns true and results in agent $P''$ adding $δ$ to SubDialogue list $SubD$. |
| assert(Φ,γ) | findInQS(QS,γ) = true and notFindInCS(Φ,CS_{P'}) = true and findInK BorCS((Φ,γ),KB_{P'},CS_{P'}) = true and addToCS (Φ,CS_{P'}) = true |  
• `findInQS` function returns true if agent $P''$ is able to find $γ$ in the dialogue Question Store $QS$.  
• `notFindInCS` function returns true if agent $P''$ is not able to find $Φ$ in its Commitment Store $CS_{P'}$.  
• `findInK BorCS((Φ,γ),KB_{P'},CS_{P'})` function returns true if agent $P''$ is able to find $(Φ,γ)$ in either its knowledge base $KB_{P'}$ or its commitment store $CS_{P'}$.  
• `addToCS (Φ,CS_{P'})` function always returns true and results in agent $P$ adding $Φ$ to its commitment store $CS_{P''}$. |
<table>
<thead>
<tr>
<th>Act</th>
<th>PreC(Act,KB,CS)</th>
<th>Note</th>
</tr>
</thead>
</table>
| Assert (Φ₂,X) | setInitialValueForX(X) = true and findInQS(QS₂,X) = true and notFindInCS(Φ₂,CSₚ) = true and findInKBorCS((Φ₂,X),KBₚ',CSₚₜ) = true and addToCS (Φ₂,CSₚ') = true | • setInitialValueForX function always returns true and results in set initial value for X. Note that X can be either γ or βₙ ∈ δ.  
• (See assert(Φ,γ) for more information about functions definition) |
| close(γ)     | findInOpenDialogue(γ,OpenD) = true and allSubDialogueClosed(γ,SubD,ClosedD) = true and (notFindInKBandCS(Φ,γ),KBₚ',CSₚₜ) = true or findInCS(Φ,CSₚ') = true ) and (noRelationship(δ,γ) = true or notFindInKB(δ,KBₚ') = true or findInQS(QS, δ) = true) | when ActType (close) = {Intermediate}, at last four functions of seven functions must return true:  
• 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 KBₚ' or other agent P commitment store CSₚ.  
• findInCS function returns true if agent P'' is able to find Φ in its commitment store CSₚ'.  
• 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 KBₚ'.  
• findInQS function returns true if agent P'' is able to find δ= β₁,β₂,β₃...βₙ in the dialogue Question Store QS. |
close(γ)
(
notFindInKbandCS((Φ, γ), KBp, CSp) = true
or
findInCS(Φ, CSp) = true
)
and
(
noRelationship(δ, γ) = true
or
notfindInKB(δ, KBp) = true
or
findInQS(QS, δ) = true
)
and
addToClosedDialogue (γ, ClosedD) = true

when ActType (close) = \{Termination\}, at last three functions of seven functions must return true:

- cannotFindInKB and CS function returns true if agent $P''$ is not able to find $(Φ, γ)$ in either in its knowledge base $KBp$ or other agent $P$ commitment store $CSp$.
- findInCS function returns true if agent $P''$ is able to find $Φ$ in its commitment store $CSp$.
- 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 $KBp$.
- findInQS function returns true if agent $P''$ is able to find $δ = 1/2, 2/2, 3/2, ..., n$ in the dialogue Question Store $QS$.
- addToClosedDialogue function always returns true and results in agent $P''$ adding $γ$ to closed Dialogue list $ClosedD$.

subclose(δ)
findInOpenDialogue (δ, OpenD) = true and
allSubDialogueClosed(δ, Sub D, ClosedD) = true
and
(notFindInKBandCS ((Φ, δ), KBp, CSp) = true
or
findInCS(Φ, CSp) = true)
and
(noRelationship(δ2, δ) = true
or
notfindInKB(δ2, KBp) = true
or
findInQS(QS, δ2) = true)

when agent sends subclose(δ) after open, assert or subclose(Z), at last five functions of eight functions must return true:

- FindInOpenDialogue
- allSubDialogueClosed
- notFindInKB and CS
- findInCS
- noRelationship
- notFindInKB
- findInQS

(See close(γ), ActType (close) = \{Intermediate\}, for more information about functions definition)
Act | $\text{PreC(Act,KB,CS)}$ | Note
--- | --- | ---
subclose($\delta$) | ({
notFindInKbandCS((\Phi,\delta),KB_{p},CS_{p}) = \text{true}
or
findInCS(\Phi,CS_{p}) = \text{true}
}) and ({
noRelationship(\delta_{2},\delta) = \text{true}
or
notfindInKB(\delta_{2},KB_{p}) = \text{true}
or
findInQS(QS,\delta_{2}) = \text{true}
}) and
addToClosedDialogue ($\delta$,ClosedD) = \text{true} | when agent sends subclose($\delta$) after subclosoe($\delta$), at last three functions of seven functions must return true:

- cannotFindInKBandCS
- findInCS
- noRelationship
- notfindInKB
- findInQS
- addToClosedDialogue

(See close($\gamma$), where ActType (close) = {Termination}, for more information about functions definition)

(6) LegalMovesTwo( $M_{1}$, CS_{A1}, CS_{A2})

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

- Dialogues begin by making an *open* move.

$M_{1}$ = 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
assert move, by either $P$ or $P''$, could be followed by $\text{Assert}$, $\text{close}$ and $\text{open}$.

subclose move, by either $P$ or $P''$, could be followed by $\text{Assert}$, $\text{close}$, $\text{subclose}$ and $\text{open}$.

if the dialogue has not terminate yet, $\text{close}$ move, by either $P$ or $P''$, could be followed by $\text{Assert}$, $\text{close}$ and $\text{open}$

open move, by either $P$ or $P''$, could be followed by $\text{Assert}$, $\text{close}$ and $\text{open}$.

- 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: $\text{open}$, $\text{assert}$, $\text{subclose}$ and $\text{close}$. There are three types of locutions: starting ($\text{open}$), termination ($\text{close}$), and intermediate ($\text{assert}$, $\text{close}$, $\text{subclose}$ and $\text{open}$).

The dialogue always starts with an $\text{open}$ and ends with a $\text{close}$ locution. $P''$ can open the discussion by sending an $\text{open}(\gamma)$ 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: $\text{assert}(\Phi, \gamma)$, $\text{open}(\delta)$ or $\text{close}(\gamma)$. $P$ will make his choice using the pre-conditions that appear in the rhombus shape. For example, in order to choose $\text{assert}(\Phi, \gamma)$, $P$ must be
Figure 4.9: DID Structure of an Inquiry Dialogue
Bridging the Specification Protocol Gap in Argumentation

Figure 4.10(a): Inquiry Dialogue Locutions Pre-conditions and Post-conditions
Figure 4.10 (b): Inquiry Dialogue Locations 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((Φ,γ),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 (Φ,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′

Figure 4.10 (C): Inquiry Dialogue Locutions Pre-conditions and Post-conditions
and so forth. The argument terminates when two close moves appear next to each other. Note that $X$ variable in $\text{assert}(\Phi 2, X)$ represents either $\gamma$ 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 ($\text{CS}''_{P''} = \{d\}$) and move 16 ($\text{CS}''_{P''} = \{d,e \land e \rightarrow b , b \rightarrow c\}$). The commitment store of agent $P$ is changed at move 9 ($\text{CS}_P = \{e\}$) and 13 ($\text{CS}_P = \{e, d, d \land e \rightarrow b\}$). 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

KBₚ = \{c, e, d \land e \rightarrow b, a \rightarrow b\}  
KBₚ⁺⁺ = \{d, b \rightarrow c\}  

1: open(c)  
2: open(b \rightarrow c)  
3: open(a \rightarrow b)  
4: subcover(a \rightarrow b)  
5: subcover(a \rightarrow b)  
6: subcover(b \rightarrow c)  
7: open(d \land e \rightarrow b)  
8: assert(\{d\},d)  
9: assert(\{e\},e)  
10: subcover(d \land e \rightarrow b)  
11: subcover(d \land e \rightarrow b)  
12: subcover(b \rightarrow c)  
13: assert(\{d,e,d \land e \rightarrow b\},b)  
14: subcover(b \rightarrow c)  
15: subcover(b \rightarrow c)  
16: assert(\{d,e,d \land e \rightarrow b,b \rightarrow C\},C)  
17: close(c)  
18: close(c)  

QS1:  
QS2:  
QS3:  
QS4:  
P''  
P
(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):
(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).
(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.
(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 achieved.\(^{18}\) 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-conditions) 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.
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).

**Pattern 1:**

**Name**: Starting pattern (SP).

**Problem**: How to start an argument (dialogue).

**Solution**:

![Diagram of Pattern 1](image-url)
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(R_P1(KB_P, CS_P, Topic, ID_A), ID_P) ::= 
    SL(Topic) => a(R_A1(KB_A, CS_A, ID_P), ID_A) \leftarrow C1
    then
    a(R_P2(KB_P, CS_P, Topic, ID_A), ID_P).

a(R_A1(KB_A, CS_A, ID_P), ID_A) ::= 
    C2 \leftarrow SL(Topic) <= a(R_P1(KB_P, CS_P, Topic, ID_A), ID_P)
    then
    a(R_A2(KB_A, CS_A, Topic, ID_P), ID_A)
```
Where $SL$ represents the *Starting Locution* and $C1$ represents a condition that must be satisfied in order for a proposal agent $ID_P$ to send the *Starting Locution SL*. Usually, $C1$ 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 $C1$ condition. If the $C1$ 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{align*}
  a(R_{Sender1}(KB_{Sender}, CS_{Sender}, CS_{Receiver}, Topic, ID_{Receiver}), ID_{Sender})::= \\
  R_{Sender1} & \ TL \approx> R_{Receiver1} \\
  \text{or} \\
  R_{Sender1} & \ IL \approx> R_{Receiver1} \\

  a(R_{Receiver1}(KB_{Receiver}, CS_{Receiver}, CS_{Sender}, Topic, ID_{Sender}), ID_{Receiver})::= \\
  R_{Receiver1} & \ TL \lessapprox R_{Sender1} \\
  \text{or} \\
  R_{Receiver1} & \ IL \lessapprox R_{Sender1}
\end{align*}
\]

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. \( \approx> \) represents outgoing messages from a role, and \( \lessapprox \) represents incoming messages.

In this LCC pattern, there are two roles: \( R_{Sender1} \) and \( R_{Receiver1} \). The \( R_{Sender1} \) 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 \( R_{Sender1} \) role begins by sending either a Termination Locution \( TL \) to the \( R_{Receiver1} \) role or an Intermediate Locution \( IL \). The \( \approx> \) symbol indicates that the \( R_{Sender1} \) role may send one or more different \( TLs \) (or \( ILs \)) to the \( R_{Receiver1} \) role.

The \( R_{Receiver1} \) 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 \( R_{Receiver1} \) role begins by receiving either a Termination Locution \( TL \) from the \( R_{Sender1} \) role or an Intermediate Locution \( IL \). The \( \lessapprox \) symbol indicates that the \( R_{Receiver1} \) role may receive one or more different \( TLs \) (or \( ILs \)) from the \( R_{Sender1} \) role.
Rewriting methods:

First (Sending Termination Method): Rewriting of the "\( R_{Sender1}^{TL} \approx> R_{Receiver1} \)"

If there is a general relation of "\( R_{Sender1}^{TL} \approx> R_{Receiver1} \)" then it is possible to specialise it within two different statements:

**Rewrite 1: (one termination location)**

We might specialise "\( R_{Sender1}^{TL} \approx> R_{Receiver1} \)" 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 \( CS \)).

\[
TL \ (Topic) \Rightarrow a(R_{Receiver1}(KB_{Receiver},CS_{Receiver},CS_{Sender},Topic,ID_{Sender}),ID_{Receiver}) \leftarrow C1
\]

**Rewrite 2: (multiple termination location)**

We might specialise "\( R_{Sender1}^{TL} \approx> R_{Receiver1} \)" 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 \( R_{Receiver1} \).

\[
TL \ (Topic) \Rightarrow a(R_{Receiver1}(KB_{Receiver},CS_{Receiver},CS_{Sender},Topic,ID_{Sender}),ID_{Receiver}) \leftarrow C1
\]

**Second (Receiving Termination Method): Rewriting of the "\( R_{Receiver1}^{<\approx TL} R_{Sender1} \)"

If there is a general relation of "\( R_{Receiver1}^{<\approx TL} R_{Sender1} \)" then it is possible to specialise it within two different statements:

**Rewrite 1: (one termination location)**

We might specialise "\( R_{Receiver1}^{<\approx TL} R_{Sender1} \)" 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, $C_2$ may represent more than one condition that is connected by *or* and *and* operators. Usually, $C_2$ is a condition over the role arguments (e.g. $KB$ and $CS$).

\[
C_2 \leftarrow TL (\text{Topic}) \leq a(R_{\text{Sender1}}(KB_{\text{Sender}}, CS_{\text{Sender}}, CS_{\text{Receiver}}, \text{Topic}, ID_{\text{Receiver}}, ID_{\text{Sender}}))
\]

**Rewrite 2: (multiple termination location)**

We might specialise "$R_{\text{Receiver1}} \approx TL R_{\text{Sender1}}" to an interaction statement that receives a $TL(Topic)$ Termination message from agent $ID_{\text{Sender}}$. $C_2$ represents a condition that must be satisfied after receiver agent receives the *Termination Locution TL*. Then, there is another termination relation between $R_{\text{Sender1}}$ and $R_{\text{Receiver1}}$.

\[
C_2 \leftarrow TL(\text{Topic}) \leq a(R_{\text{Sender1}}(KB_{\text{Sender}}, CS_{\text{Sender}}, CS_{\text{Receiver}}, \text{Topic}, ID_{\text{Receiver}}, ID_{\text{Sender}}))
\]

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

**Rewrite 1: (One intermediate location)**

We might specialise "$R_{\text{Sender1}} IL \approx R_{\text{Receiver1}}" to an interaction statement that sends message $IL(Topic)$ to agent $ID_{\text{Receiver}}$ which is achieved by the constraint $C_3$. Following this, it changes its role. In practice, $C_3$ may represent more than one condition, which is connected by *or* and *and* operators. Usually, $C_3$ is a condition over the role arguments (e.g. $KB$ and $CS$).

\[
IL(\text{Topic}) \Rightarrow a(R_{\text{Receiver1}}(KB_{\text{Receiver}}, CS_{\text{Receiver}}, CS_{\text{Sender}}, \text{Topic}, ID_{\text{Sender}}, ID_{\text{Receiver}}) \leftarrow C_3 \text{ then } a(R_{\text{Sender2}}(KB_{\text{Sender}}, CS_{\text{Sender}}, CS_{\text{Receiver}}, \text{Topic}, ID_{\text{Receiver}}, ID_{\text{Sender}}))
\]
Rewrite 2: (multiple Intermediate locutions):

We might specialize \( R_{Sender1} \overset{IL}{\Rightarrow} R_{Receiver1} \) to an interaction statement that sends message \( IL(Topic) \) to agent \( ID_{Receiver} \) which is achieved by the constraint \( C_3 \), after that it recurses. Then, there is then another recursive relation between \( R_{Sender1} \) and \( R_{Receiver1} \).

\[
IL(Topic) \Rightarrow a(R_{Receiver1}(KB_{Receiver}, CS_{Receiver}, CS_{Sender}, Topic, ID_{Sender}), ID_{Receiver}) \\
\leftarrow C_3 \\
\text{then} \\
a(R_{Sender2}(KB_{Sender}, CS_{Sender}, CS_{Receiver}, Topic, ID_{Receiver}), ID_{Sender})
\]

or

\[
R_{Sender1} \overset{IL_2}{\Rightarrow} R_{Receiver1}
\]

Fourth (Receiving Intermediate method): Rewriting of \( "R_{Receiver1} \overset{IL}{\Leftarrow} R_{Sender1}" \)

Rewrite 1: (One intermediate locution)

We might specialise \( R_{Receiver1} \overset{IL}{\Leftarrow} R_{Sender1} \) to an interaction statement that receives message \( IL(Topic) \) from agent \( ID_{Sender} \). Following this, it changes its role. \( C_3 \) represents a condition that must be satisfied after receiver agent receives the Intermediate Location \( IL \). In practice, \( C_4 \) may represent more than one condition, which is connected by \( or \) and \( and \) operators. Usually, \( C_4 \) is a condition over the role arguments (e.g. \( KB \) and \( CS \)).

\[
C_4 \leftarrow IL(Topic) \\
<= a(R_{Sender1}(KB_{Sender}, CS_{Sender}, CS_{Receiver}, Topic, ID_{Receiver}), ID_{Sender}) \\
\text{then} \\
a(R_{Receiver2}(KB_{Receiver}, CS_{Receiver}, CS_{Sender}, Topic, ID_{Sender}), ID_{Receiver})
\]

Rewrite 2: (multiple Intermediate locutions):

We might specialize \( R_{Receiver1} \overset{IL}{\Leftarrow} R_{Sender1} \) 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 \( R_{Receiver1} \). \( C_4 \) represents a
condition that must be satisfied after receiver agent receives the Intermediate Locution $IL$.

\[
\begin{align*}
C4 & \leftarrow IL(Topic) \\
\leq & \ a(R_{Sender1}(KB_{Sender}, CS_{Sender}, CS_{Receiver}, Topic, ID_{Receiver}), ID_{Sender}) \\
\text{then} & \ a(R_{Receiver2}(KB_{Receiver}, CS_{Receiver}, CS_{Sender}, Topic, ID_{Sender}), ID_{Receiver}) \\
\text{or} & \ R_{Receiver1} \preceq_{IL2} R_{Sender1}
\end{align*}
\]

**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. (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 $replyToProposalReceiver_{prop}$ (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,AgentP,AgentO) location for a pair of agents (step i.1): AgentP and AgentO (telling them to interact together) and then recourses (step i.2) or changes its role (step i.3).

ii. Both AgentP and AgentO receive argueWith(Topic,AgentP,AgentO) 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>= 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 Topic $\epsilon C_{Proposal}$ (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):

  $$(1) \text{proposalSender}_{prop} \quad (2)\text{proposalReceiver}_{ID}$$

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

  $$(1) \text{ReplyToProposalSender}_{ID} \quad (2)\text{replyToProposalReceiver}_{prop}$$
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. $\text{resultSender}_{\text{proposal}}$
2. $\text{sendReachAgreement}_{\text{Proposal}}$
3. $\text{divideGroup}_{\text{Proposal}}$
4. $\text{resultReceiver}_{\text{ID}}$

Where $\text{DivideC2}$ represents a condition that must be satisfied in order for a proposal agent to divide agents into groups composed of two agents. By default, $\text{DivideC2}$ is \\
"lessThan(NAccepting,NSupporters) and isNotEmpty(RejectionList) and isNotEmpty(AcceptingList)"

$\text{TerminationC1}$ represents a condition that must be satisfied in order for a proposal agent to terminate the dialogue between $N$-agent. By default, $\text{TerminationC1}$ is \\
"greaterThanOrEqual(NAccepting,NSupporters)" a function which returns true if $\text{NAccepting}$ is greater than or equal to $\text{NSupporters}$.

$\text{AgentGroupC3}$ represents a function which divides agents into groups composed of two agents.
Figure 5.6: Structure (replyToProposalSender and replyToProposalReceiver roles)
Figure 5.7: Structure (resultSenderProposal, sendReachAgreementProposal, divideGroupProposal and resultReceiver roles)
RecursC4 represents a condition that must be satisfied in order for a proposal agent to recur. By default, RecursC4 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 proposalSenderProposal role of proposal agent IDProposal has four input parameters: AgentList, NAgent, NSupporters and Topic. The proposalSenderProposal 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 replyToProposalReceiverProposal role, otherwise, it recurses over the remaining agents (recourses over the OtherAgents list. Note that OtherAgents = AgentList - {the head of the AgentList}). The proposalReceiverID role begins by receiving the proposal(Topic) message from the proosalSenderProposal 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: KBID, CSID, Topic and IDProposal. 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 replyToProposalReceiverProposal 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 replyToProposalReceiverProposal role. Then it changes its role to the resultReceiverID role.
Table 5.1: Broadcasting Pattern Roles Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AgentList</td>
<td>Agents ID list</td>
</tr>
<tr>
<td>NAgent</td>
<td>The number of agents (note that the number of agents ( \geq 3 ))</td>
</tr>
<tr>
<td>NSupporters</td>
<td>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)</td>
</tr>
<tr>
<td>Topic</td>
<td>Main dialogue topic</td>
</tr>
<tr>
<td>IDProposal</td>
<td>Proposal agent ID</td>
</tr>
<tr>
<td>OtherAgents</td>
<td>Agents ID list</td>
</tr>
<tr>
<td></td>
<td>Where, OtherAgents = AgentList ( \setminus ) {The head of the AgentList}</td>
</tr>
<tr>
<td>KBID</td>
<td>Agent Knowledge Base</td>
</tr>
<tr>
<td>CSID</td>
<td>Agent Commitment Store</td>
</tr>
<tr>
<td>AcceptanceList</td>
<td>The list of the accepting agents ID (note that when replyToProposalReceiverProposal role is called AcceptanceList is empty)</td>
</tr>
<tr>
<td>RejectionList</td>
<td>The list of the rejected agents ID (note that when replyToProposalReceiverProposal role is called RejectionList is empty)</td>
</tr>
<tr>
<td>NAccepting</td>
<td>The number of accepted agents (note that when replyToProposalReceiverProposal role is called NAccepting equal 0)</td>
</tr>
<tr>
<td>NRejecting</td>
<td>The number of rejected agents (note that when replyToProposalReceiverProposal role is called NRejecting equal 0)</td>
</tr>
<tr>
<td>NReply</td>
<td>The number of reply agents (note that when replyToProposalReceiverProposal role is called NReply equal 0)</td>
</tr>
<tr>
<td>AgentGroup</td>
<td>Agent group list. Each element of the agent group list is composed of two agents ID ( (P, O) )</td>
</tr>
<tr>
<td>P</td>
<td>Agent ID</td>
</tr>
<tr>
<td>O</td>
<td>Agent ID</td>
</tr>
</tbody>
</table>

The `replyToProposalReceiverProposal` role of the proposal agent has four input parameters: `AgentList`, `NAgent`, `NSupporters` and `Topic`. It also has five output parameters: `NReply`, `AcceptingList`, `RejectingList`, `NAccepting` and `NRejecting`. The values of the output parameters when the role begins are as follows: \( NReply=0, AcceptingList=[] \), \( RejectingList=[] \), \( NAccepting=0 \) and \( NRejecting=0 \).

The `replyToProposalReceiverProposal` 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

Chapter 5: Synthesis of Concrete Protocols

135
### Table 5.2 (a): Broadcasting Pattern Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getAgentIDFromList</code></td>
<td>The <code>getAgentIDFromList</code> function gets one agent ID from the <code>AgentsList</code> and puts the remainder agents in the <code>otherAgents</code> list.</td>
</tr>
<tr>
<td><code>(AgentList,otherAgents,ID)</code></td>
<td></td>
</tr>
<tr>
<td><code>addTopicToCS</code></td>
<td>The <code>addTopicToCS</code> function always returns true and results in the agent adding <code>Topic</code> to its commitment store <code>CS</code>.</td>
</tr>
<tr>
<td><code>(Topic,CS)</code></td>
<td></td>
</tr>
<tr>
<td><code>agentListEmpty</code></td>
<td>The <code>agentListEmpty</code> function returns true if <code>AgentList</code> is empty (which means that proposal agent broadcasts the <code>Topic</code> to all agents).</td>
</tr>
<tr>
<td><code>(AgentList)</code></td>
<td></td>
</tr>
<tr>
<td><code>findTopicInKB</code></td>
<td>The <code>findTopicInKB</code> function returns true if the agent is able to find <code>Topic</code> in its Knowledge Base <code>KB</code>.</td>
</tr>
<tr>
<td><code>(Topic, KB)</code></td>
<td></td>
</tr>
<tr>
<td><code>notFindTopicInCS</code></td>
<td>The <code>notFindTopicInCS</code> function returns true if the agent is not able to find <code>Topic</code> in its Commitment Store <code>CS</code>.</td>
</tr>
<tr>
<td><code>(Topic,CS)</code></td>
<td></td>
</tr>
<tr>
<td><code>notFindOppTopicInCS</code></td>
<td>The <code>notFindOppTopicInCS</code> which returns true if the agent is not able to find the opposite of <code>Topic</code> (<code>not(Topic)</code>) in its commitment store <code>CS</code>.</td>
</tr>
<tr>
<td><code>(not(Topic),CS)</code></td>
<td></td>
</tr>
<tr>
<td><code>notFindTopicInKB</code></td>
<td>The <code>notFindTopicInKB</code> function returns true if the agent is not able to find <code>Topic</code> in its Knowledge Base <code>KB</code>.</td>
</tr>
<tr>
<td><code>(Topic,KB)</code></td>
<td></td>
</tr>
<tr>
<td><code>addToAcceptingList</code></td>
<td>The <code>addToAcceptingList</code> function always returns true and results in proposal agent adding the accepting agent ID to the <code>AcceptingList</code> (<code>AccList = AcceptingList ∪ {ID}</code>).</td>
</tr>
<tr>
<td><code>(AcceptingList,AccList,ID)</code></td>
<td></td>
</tr>
<tr>
<td><code>addToRejectingList</code></td>
<td>The <code>addToRejectingList</code> function always returns true and results in proposal agent adding the rejecting agent ID to the <code>RejectingList</code> (<code>RejList = RejectingList ∪ {ID}</code>).</td>
</tr>
<tr>
<td><code>(RejectingList,RejList,ID)</code></td>
<td></td>
</tr>
<tr>
<td><code>increaseRejecting</code></td>
<td>The <code>increaseRejecting</code> function increases the number of rejecting agents by adding one to <code>NRejecting</code> (<code>NRej = NRejecting +1</code>).</td>
</tr>
<tr>
<td><code>(NRejecting,NRej)</code></td>
<td></td>
</tr>
<tr>
<td><code>increaseAccepting</code></td>
<td>The <code>increaseAccepting</code> function increases the number of accepting agents (<code>NAcc = NAccepting +1</code>).</td>
</tr>
<tr>
<td><code>(NAccepting,NAcc)</code></td>
<td></td>
</tr>
<tr>
<td><code>increaseReply</code></td>
<td>The <code>increaseReply</code> function increases the number of replying agents by adding one to <code>NReply</code> (<code>NRep = NReply +1</code>).</td>
</tr>
<tr>
<td><code>(NReply,NRep)</code></td>
<td></td>
</tr>
<tr>
<td><code>RejList = RejList</code></td>
<td>Assigns the value of <code>RejList</code> argument to the <code>RejList</code> variable.</td>
</tr>
<tr>
<td><code>(RejList)</code></td>
<td></td>
</tr>
<tr>
<td><code>NRej is NRejecting</code></td>
<td>Assigns the value of <code>NRejecting</code> argument to the <code>NRej</code> variable.</td>
</tr>
<tr>
<td><code>(NRej)</code></td>
<td></td>
</tr>
</tbody>
</table>
### Table 5.2 (b): Broadcasting Pattern Roles Functions

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

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

The proposal agent then changes the \textit{replyToProposalReceiver} role to the \textit{resultSender} role. The \textit{resultSender} role has nine input parameters: \textit{AgentList}, \textit{N Agent}, \textit{NSupporters}, \textit{Topic}, \textit{NReply}, \textit{AcceptingList}, \textit{RejectionList}, \textit{NAccepting} and \textit{NRejection}. The \textit{replyToProposalReceiver} role begins by checking \textit{TerminationC1} condition. If this condition is true, then the proposal agent changes its role to the \textit{sendReachAgreement} role. Otherwise, the \textit{replyToProposalReceiver} role checks \textit{DivideC2} condition. If this condition is true, then the proposal agent changes its role to the \textit{divideGroup} role.

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

The \textit{divideGroup} role has six input parameters: \textit{AgentList}, \textit{N Agent}, \textit{NSupporters}, \textit{Topic}, \textit{AcceptingList} and \textit{RejectionList}. It also has one output parameter: \textit{AgentGroup}. This role is responsible for dividing the agents in the \textit{AgentList} list into a group composed of two agents. It begins by checking \textit{AgentGroupC3}. If this condition is true, then this role creates the first agent group by taking one agent from the head of the \textit{AcceptingList} and one agent from the head of the \textit{RejectionList}. It then sends the \textit{argueWith} message to the first group (agent \textit{P} and agent \textit{O}) and asks them to start arguing together about the dialogue \textit{Topic}. Then, if the \textit{RecursC4} condition is true, the proposal agent changes its role to the \textit{recurs} role, otherwise, it recurses.
Finally, the control changes to the $\text{resultReceiver}_{ID}$ role. The $\text{resultReceiver}_{ID}$ 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 $\text{resultReceiver}_{ID}$ role ends once it has received the $reachAgreement(Topic)$ message. Otherwise, agent $ID$ changes its role to $\text{startDID}$ 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.
Bridging the Specification Protocol Gap in Argumentation

Chapter 5: Synthesis of Concrete Protocols

Structure:

\[
a(\text{startDID}_{ID}(\text{KB}_{ID}, \text{CS}_{ID}, \text{CS}_{PartnerID}, \text{Topic}, \text{ID}_{Proposal}, \text{PartnerID}), \text{ID}) ::= \\
a(R_{Sender1} (\text{KB}_{ID}, \text{CS}_{ID}, \text{CS}_{PartnerID}, \text{Topic}, \text{ID}_{Proposal}, \text{PartnerID}), \text{ID}) \leftarrow C1 \\
or \quad a(R_{Receiver1} (\text{KB}_{ID}, \text{CS}_{ID}, \text{CS}_{PartnerID}, \text{Topic}, \text{ID}_{Proposal}, \text{PartnerID}), \text{ID}) \leftarrow C2.
\]

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-Discussion 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 proposalSenderProposal, 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-Discourse Pattern
Structure:

\[
a(\text{recurs}_{\text{Proposal}} (\text{AgentList}, \text{NAgent}, \text{NSupporters}, \text{NReply}, \text{Topic}), \text{ID}_{\text{Proposal}}) ::= \\
\text{N} = \text{NReply} + 1 \leftarrow \text{end(Topic)} \\
\leq a( \text{R}_{\text{sender}} (_, _, _, \text{Topic}, \text{ID}_{\text{Proposal}}, _), \text{ID}_{\text{sender}} ) \\
or \\
\text{N} = \text{NReply} + 1 \leftarrow \text{end(Topic)} \\
\leq a( \text{R}_{\text{receiver}} (_, _, _, \text{Topic}, \text{ID}_{\text{Proposal}}, _), \text{ID}_{\text{receiver}} ) \\
or \\
\text{recurs}_{\text{Proposal}} \leftarrow \text{end R}_{\text{Sender2}} \\
\text{then} \\
( \\
\text{a(\text{proposalSender}_{\text{Proposal}} (\text{AgentList}, \text{NAgent}, \text{NSupporters}, \text{Topic}), \text{ID}_{\text{Proposal}}) \leftarrow \text{isEqual(N, NAgent)} \\
or \\
\text{a( recurs}_{\text{Proposal}} (\text{AgentList}, \text{NAgent}, \text{NSupporters}, \text{N}, \text{Topic}), \text{ID}_{\text{Proposal}}) ) \\
).
\]

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

Rewriting methods:

**Rewriting of the "recurs}_{\text{Proposal}} \leftarrow \text{R}_{\text{Sender2}}"**

If there is a general relation of "\text{recurs}_{\text{Proposal}} \leftarrow \text{R}_{\text{Sender2}}" then it is possible to specialise within two different statements:
**Rewrite 1: (one end locution)**

We might specialise " recurs\textsubscript{Proposal} \leftarrow R\textsubscript{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.

\[
N = N\text{Reply} + 1 \leftarrow \text{end(Topic)} \leq a( R\textsubscript{sender2} (_ ,_,_,Topic,ID\text{Proposal}_, ), ID\text{sender2} ) \text{ or } N = N\text{Reply} + 1 \leftarrow \text{end(Topic)} \leq a( R\textsubscript{receiver2} (_ ,_,_,Topic,ID\text{Proposal}_, ), ID\text{receiver2} )
\]

**Rewrite 2: (multiple end locutions)**

We might specialise " recurs\textsubscript{Proposal} \Leftrightarrow R\textsubscript{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\textsubscript{Proposal} \Leftrightarrow R\textsubscript{Sender3}).

\[
N = N\text{Reply} + 1 \leftarrow \text{end(Topic)} \leq a( R\textsubscript{sender2} (_ ,_,_,Topic,ID\text{Proposal}_, ), ID\text{sender2} ) \text{ or } N = N\text{Reply} + 1 \leftarrow \text{end(Topic)} \leq a( R\textsubscript{receiver2} (_ ,_,_,Topic,ID\text{Proposal}_, ), ID\text{receiver2} ) \text{ or } \text{recurs}_{\text{Proposal}} \Leftrightarrow R\text{Sender3}
\]

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 "\textit{GenerateLCCProtocol}" (see chapter 7 for more details), summarised pictorially in Figure 5.9, can generate agent protocols...
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):
Bridging the Specification Protocol Gap in Argumentation

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, DivideC2, 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 \hspace{1cm} (Step1)  
3. **Pattern Input** (TerminationC1, DivivdeC2, AgentGroupC3, RecursC4)  
4. Use Move-To-Dialogue Pattern \hspace{1cm} (Step2)  
5. **Match** (Starting locution icon in the DID for two agents, Move-To-Dialogue Pattern)  
6. Use Recurs-To-N-Dialogue Pattern \hspace{1cm} (Step3)  
7. recursNumber = number of Termination locution icon in the DID for two agents -1  
8. If (reuseNumber = 0) then \hspace{1cm} //one Termination Locution  
9. Select&Save RewriteMethod=**Rewrite 1**  
10. **Match** (Termination Icon, Recurs-To-N-Dialogue Pattern, RewriteMethod)  
11. Else \hspace{1cm} //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 = reuseNumber)  
17. Go To two agents algorithm \hspace{1cm} (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](image)

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 proposalReceiverID (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.
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. Automated Transformation**
- **2. Construction**
- **3. Automated Creation**
- **4. Applying the verification process**

Resulting LCC Protocol → CPNXML → State space → Verification model → Result

Verification model (General behaviour property checking code in CPN SML)

Figure 6.1: Verification Process
(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 \textit{TOPIC}. This type is used to model the main dialogue topic. It is defined as a string.

\begin{verbatim}
colset TOPIC = string;
\end{verbatim}

(2) Type \textit{Message}. This type is used to model messages. It is defined as the product of the types \textit{Locution}, \textit{TOPIC}, \textit{Premise}, \textit{ID} and \textit{ID}. The types \textit{Locution}, \textit{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.
Bridging the Specification Protocol Gap in Argumentation

<table>
<thead>
<tr>
<th>LCC CPNs Model</th>
<th>CPNXML Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>non</td>
<td>&lt;page id=&quot;ID6&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;pageattr name=&quot;Role Name&quot; /&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/page&gt;</td>
</tr>
</tbody>
</table>

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).
Bridging the Specification Protocol Gap in Argumentation

Chapter 6: Verification Method based Coloured Petri Nets and SML

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

<table>
<thead>
<tr>
<th>LCC Code (Send a Message)</th>
<th>CPN Model</th>
<th>CPNXML Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send message symbol</td>
<td>[Boolean conditions]</td>
<td>1. <code>&lt;trans id=&quot;ID1423689023&quot;&gt;</code></td>
</tr>
<tr>
<td>&quot;Send&quot; + Message Name</td>
<td></td>
<td>2. <code>&lt;text&gt; &quot;Send&quot;+ message name&lt;/text&gt;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. <code>&lt;cond&gt;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. <code>&lt;text tool=&quot;CPN Tools version=&quot;2.9.11&quot;&gt; LCC Boolean conditions&lt;/text&gt;</code></td>
</tr>
</tbody>
</table>
|                           |           | 5. `<condition>
|                           |           |   `<text tool="CPN Tools version="2.9.11"> LCC Boolean conditions</text>` |
|                           |           |   `<cond>` |
|                           |           | 6. `<text tool="CPN Tools version="2.9.11"> LCC Boolean conditions</text>` |
|                           |           | 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. `<text tool="CPN Tools version="2.9.11"> 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. `<text tool="CPN Tools version="2.9.11"> Message arguments</text>` |
|                           |           | 27. `</annot>` |
|                           |           | 28. `</arc>` |

### 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);
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).
then
a(NewRoleName(Arguments),AgentID)
\leftarrow\text{ChangeRoleConditions}

<table>
<thead>
<tr>
<th>LCC CPNs Model</th>
<th>CPNXML Structure</th>
</tr>
</thead>
</table>
| \textit{LCC} (LCC Then keyword followed by Changing Role statement) | 1. <place id="ID1423689035">
2. <text> \textit{New Role name} </text>
3. <type id="ID1423689036">
4. <text tool="CPN Tools" version="2.9.11">
5. \textit{Role} </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" orientation="TtoP" order="1">
14. <transend idref="Last Message transition ID"/>
15. <placeend idref="New place ID"/>
16. <annot id="ID1423689050">
17. <text tool="CPN Tools version="2.9.11">
18. \textit{Role Arguments} </text>
19. </annot>
20. </text>
21. </annot>
22. </arc> |

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

\textbf{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= "OpenDialogue", 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
<table>
<thead>
<tr>
<th>LCC (LCC or keyword)</th>
<th>LCC CPNs Model</th>
<th>CPNXML Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Or symbol</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. `<place id="ID1423689035">`
2. `<text>` **Main role name** `<text>`
3. `<type id="ID1423689036">`
4. `<text tool="CPN Tools" version="2.9.11">`
5. `<Role>` `<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" orientation="PtoT" order="1">`
14. `<transend idref="First Message transition ID"/>`
15. `<placeend idref="New place ID"/>`
16. `<annot id="ID1423689050">`
17. `<text tool="CPN Tools version="2.9.11">`
18. `<Role arguments>`
19. `</text>`
20. `</annot>`
21. `</arc>`
22. `</arcr`
23. `<arc id="ID1423689049" orientation="PtoT" order="1">`
24. `<transend idref="Second Message transition ID"/>`
25. `<placeend idref="New place ID"/>`
26. `<annot id="ID1423689050">`
27. `<text tool="CPN Tools version="2.9.11">`
28. `<Role arguments>`
29. `</text>`
30. `</annot>`
31. `</arc>`
32. `</arcr`

Table 6.5: LCC-CPNXML Transformation Table (Or keyword)
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):
**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));
LCC
(\(\cdot\) After sending or receiving message statement)

Message(Topic) \(\Rightarrow\) a(RoleName(Arguments),AgentID)

\[\text{Conditions} \quad \Rightarrow\]

Message(Topic) \(\Leftarrow\) a(RoleName(Arguments),AgentID)

<table>
<thead>
<tr>
<th>LCC CPNs Model</th>
<th>CPNXML Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>End symbol</strong></td>
<td>1. (&lt;\text{id=&quot;ID1423689035&quot;}&gt;)</td>
</tr>
<tr>
<td>(\text{Out})</td>
<td>2. (&lt;\text&gt;{End})&lt;/text&gt;</td>
</tr>
<tr>
<td>Role</td>
<td>3. (&lt;\text{id=&quot;ID1423689036&quot;}&gt;)</td>
</tr>
<tr>
<td>(\text{role arguments})</td>
<td>4. (&lt;\text{tool=&quot;CPN Tools&quot; version=&quot;2.9.11&quot;}&gt;)</td>
</tr>
<tr>
<td></td>
<td>5. (&lt;\text{Role})&lt;/text&gt;</td>
</tr>
<tr>
<td></td>
<td>6. (&lt;\text/&gt;)</td>
</tr>
<tr>
<td></td>
<td>7. (&lt;\text{id=&quot;ID1423689037&quot;}&gt;)</td>
</tr>
<tr>
<td></td>
<td>8. (&lt;\text{tool=&quot;CPN Tools&quot; version=&quot;2.9.11&quot;/&gt;)</td>
</tr>
<tr>
<td></td>
<td>9. (&lt;\text/&gt;)</td>
</tr>
<tr>
<td></td>
<td>10. (&lt;\text{id=&quot;ID1424205036&quot; type=&quot;Out&quot;&gt;})</td>
</tr>
<tr>
<td></td>
<td>11. (&lt;\text/&gt;)</td>
</tr>
<tr>
<td></td>
<td>12. (&lt;\text{place})</td>
</tr>
<tr>
<td></td>
<td>13. (&lt;\text{id=&quot;ID1423689049&quot;}&gt;)</td>
</tr>
<tr>
<td></td>
<td>14. (&lt;\text{orientation=&quot;TtoP&quot; order=&quot;1&quot;/&gt;)</td>
</tr>
<tr>
<td></td>
<td>15. (&lt;\text{idref=&quot;Message transition ID&quot;/&gt;)</td>
</tr>
<tr>
<td></td>
<td>16. (&lt;\text{idref=&quot;New place ID&quot;/&gt;)</td>
</tr>
<tr>
<td></td>
<td>17. (&lt;\text{id=&quot;ID1423689050&quot;}&gt;)</td>
</tr>
<tr>
<td></td>
<td>18. (&lt;\text{tool=&quot;CPN Tools version=&quot;2.9.11&quot;/&gt;)</td>
</tr>
<tr>
<td></td>
<td>19. (&lt;\text{Role arguments}&gt;)</td>
</tr>
<tr>
<td></td>
<td>20. (&lt;\text/&gt;)</td>
</tr>
<tr>
<td></td>
<td>21. (&lt;\text/&gt;)</td>
</tr>
<tr>
<td></td>
<td>22. (&lt;\text{/arc}&gt;)</td>
</tr>
</tbody>
</table>

Table 6.8: LCC-CPNXML Transformation table (End Statement)
Bridging the Specification Protocol Gap in Argumentation

<table>
<thead>
<tr>
<th>LCC Code (Get condition)</th>
<th>CPNXML Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get Conditions</td>
<td>CPNs Model</td>
</tr>
</tbody>
</table>

1. `<trans id="ID1423689023">`  
2. `<text> "getConditionTransition" </text>`  
3. `<cond>`  
4. `<text tool="CPN Tools" version="2.9.11">true</text>`  
5. `"true"`  
6. `</cond>`  
7. `</trans>`  
8. `<place id="ID1423689035">`  
9. `<text> Item name </text>`  
10. `<type id="ID1423689036">`  
11. `<text tool="CPN Tools" version="2.9.11">Item Type</text>`  
12. `</type>`  
13. `<initmark id="ID1423689037">`  
14. `<text tool="CPN Tools" version="2.9.11">/Item Type</text>`  
15. `</initmark>`  
16. `</place>`  
17. `<place id="ID1423689036">`  
18. `<text> flow </text>`  
19. `<type id="ID1423689036">`  
20. `<text tool="CPN Tools" version="2.9.11">/flow</text>`  
21. `</type>`  
22. `<initmark id="ID1423689039">`  
23. `<text tool="CPN Tools" version="2.9.11">Role</text>`  
24. `</initmark>`  
25. `</place>`  
26. `<arc id="ID1424199627" orientation="PtoT" order="1">`  
27. `<transend idref="getConditionTransition ID"/>`  
28. `<placeend idref="Item Place ID"/>`  
29. `<annot id="ID1424199628">`  
30. `<text tool="CPN Tools" version="2.9.11">/getConditionTransition</text>`

Table 6.9 (a): LCC-CPNXML Transformation Table (Get an Argument Condition)
<table>
<thead>
<tr>
<th>LCC Code</th>
<th>CPNs Model</th>
<th>CPNXML Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Get condition)</td>
<td><strong>GetConditions</strong></td>
<td></td>
</tr>
</tbody>
</table>

34. **Item arguments**
35. `</text>`
36. `</annot>`
37. `</arc>`
38. `<arc id="ID1424199687" orientation="PtoT" order="1">`
39. `<transend idref="getConditionTransition ID"/>
40. `<placeend idref="flow Place ID"/>
41. `<annot id="ID1424199618">
42. `<text tool="CPN Tools" version="2.9.11">
43. **Role arguments**
44. `</text>`
45. `</annot>`
46. `</arc>`
47. `<arc id="ID1424199684" orientation="TtoP" order="1">`
48. `<transend idref="getConditionTransition ID"/>
49. `<placeend idref="Message Place ID"/>
50. `<annot id="ID1424199638">
51. `<text tool="CPN Tools" version="2.9.11">
52. **Message arguments**
53. `</text>`
54. `</annot>`
55. `</arc>`
56. `<arc id="ID1424199664" orientation="TtoP" order="1">`
57. `<transend idref="Message transition ID"/>
58. `<placeend idref="Item Place ID"/>
59. `<annot id="ID1424199638">
60. `<text tool="CPN Tools" version="2.9.11">
61. **Get Condition**
62. `</text>`
63. `</annot>`
64. `</arc>`

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

(Get condition)

CPNs Model

← GetConditions

CPNXML Structure

| 68. <arc id="ID1424129684" |
| 69. orientation="TtoP" order="1"> |
| 70. <transend idref="Message transition ID"/> |
| 71. <placeend idref="flow Place ID"/> |
| 72. <annot id="ID1424299638"> |
| 73. <text tool="CPN Tools" version="2.9.11"> |
| a. Role Arguments |
| 74. </text> |
| 75. </annot> |
| 76. </arc> |

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,
Table 6.10: LCC-CPNXML Transformation table (Role in the CPN Superpage)

Port ID). Note that port socket relation (portsock) is used to represent the hierarchical relation among CPN pages. The socket ID represents the place ID in the superpages and the Port ID represents the place ID in the corresponding subpage. (see chapter 2, section 2.3.2.1) (line 1 to 9 of Table 6.10);

(2) One or more arcs. The number of arcs is dependent upon the number of related sockets. These arcs are used to connect the new substitution to the related sockets. Each arc has an arc ID = unique identifier, the arc type = PtoT or TtoP (depends on the relation between the transition and the socket), the transition ID reference = the new substitution transition ID, the place ID reference = the related socket ID, the arc inscription depends on the socket colour set type (line 10 to 20 of Table 6.10);

(3) If this role is the primary role (the first role in the LCC code which is responsible for opening the dialogue), then:

a) Create one new place where the place ID = unique identifier, the place name = "OpenDialogue", the place colour set type = Topic and place (port) type = In (line 1 to 12 of Table 6.11);
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
Bridging the Specification Protocol Gap in Argumentation

<table>
<thead>
<tr>
<th>LCC Code (Starter Role Arguments)</th>
<th>CPNXML Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(RoleName(Arguments, Topic),AgentID)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LCC CPNs Model</th>
<th>CPNXML Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Starter Role argument symbol</strong></td>
<td>1. <code>&lt;place id=&quot;ID1423689035&quot;&gt;</code></td>
</tr>
<tr>
<td>Arguments initial values</td>
<td>2. <code>&lt;text&gt;</code> <em>Agent ID</em> <code>&lt;text&gt;</code></td>
</tr>
<tr>
<td>Agent ID</td>
<td>3. <code>&lt;type id=&quot;ID1423689036&quot;&gt;</code></td>
</tr>
<tr>
<td>In</td>
<td>4. <code>&lt;text tool=&quot;CPN Tools&quot; version=&quot;2.9.11&quot;&gt;</code></td>
</tr>
<tr>
<td>Role</td>
<td>5. <em>Role</em> <code>&lt;text&gt;</code></td>
</tr>
<tr>
<td>(Role Arguments)</td>
<td>6. <code>&lt;type&gt;</code> <code>&lt;text&gt;</code></td>
</tr>
<tr>
<td></td>
<td>7. <code>&lt;initmark id=&quot;ID1423689037&quot;&gt;</code></td>
</tr>
<tr>
<td></td>
<td>8. <code>&lt;text tool=&quot;CPN Tools&quot; version=&quot;2.9.11&quot;&gt;</code></td>
</tr>
<tr>
<td></td>
<td>9. <em>Arguments initial values</em></td>
</tr>
<tr>
<td></td>
<td>10. <code>&lt;initmark&gt;</code></td>
</tr>
<tr>
<td></td>
<td>11. <code>&lt;port id=&quot;ID1424205036&quot; type=&quot;In&quot;&gt;</code></td>
</tr>
<tr>
<td></td>
<td>12. <code>&lt;port&gt;</code></td>
</tr>
<tr>
<td></td>
<td>13. <code>&lt;place&gt;</code></td>
</tr>
<tr>
<td></td>
<td>14. <code>&lt;arc id=&quot;ID1423689049&quot;</code></td>
</tr>
<tr>
<td></td>
<td>15. <code>orientation=&quot;PtoT&quot; order=&quot;1&quot;&gt;</code></td>
</tr>
<tr>
<td></td>
<td>16. <code>&lt;transend idref=&quot;New substitution transition ID&quot; /&gt;</code></td>
</tr>
<tr>
<td></td>
<td>17. <code>&lt;placeend idref=&quot;New place ID&quot;</code></td>
</tr>
<tr>
<td></td>
<td>18. <code>&lt;annot id=&quot;ID1423689050&quot;&gt;</code></td>
</tr>
<tr>
<td></td>
<td>19. <code>&lt;text tool=&quot;CPN Tools version=&quot;2.9.11&quot;&gt;</code></td>
</tr>
<tr>
<td>20. <em>Role arguments</em></td>
<td>21. <code>&lt;text&gt;</code></td>
</tr>
<tr>
<td></td>
<td>22. <code>&lt;annot&gt;</code></td>
</tr>
<tr>
<td></td>
<td>23. <code>&lt;arc&gt;</code></td>
</tr>
</tbody>
</table>

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();
   CalculateSccGraph();
   ```
(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,
Termination Locutions file: contains message names which are used to terminate the dialogue;

Termination Locutions Effect CS and Effective CS files: contain the effect of the termination message to the sender commitment store CS;

Player Types file: contains dialogue game player types (e.g. opponent or proponent);

Player IDs file: contains dialogue game player IDs;

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** \(DID\)OpenDialogueMessages \(\Rightarrow\)DID information
3. **Call** \(\)CheckProperty1
4. **Input** \((SS,DID\)OpenDialogueMessages\)
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 \(DID\)OpenDialogueMessages variable.

(3) Line 3: Call CheckProperty1 function.

(4) Line 4: CheckProperty1 function inputs are \(SS\) and \(DID\)OpenDialogueMessages.

(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**  
   \[ \text{TNodes} = \text{state space termination nodes information} \]

2. **Read**  
   \[ \text{TDID} = \text{DID termination nodes information} \]

3. **Call**  
   \[ \text{CheckProperty2} \]

4. **Input**  
   \[ (\text{TNodes}, \text{TDID}) \]

5. **Extract**  
   \[ (\text{message}, \text{topic}, \text{premise}, \text{sender}, \text{receiver}, \text{sCS}, \text{rCS}) \]

6. **Extract**  
   \[ (\text{DIDTL}, \text{DIDEf}, \text{DIDAID}, \text{DIDS}) \]

7. \[ \text{val checkSR} = \text{checkSenderReceiver}(\text{message}, \text{sender}, \text{receiver}, \) \]

8. \[ \text{opponent, proponent, DIDAID, DIDS} \]

9. \[ \text{val csContant} = \text{checkTheContantofCS}(\text{role}, \text{message}, \text{rCS}, \text{topic}, \text{premise}, \text{rCSSize}, \) \]

10. \[ \text{topicSize, premiseSize, DIDTL, DIDEf}) \]

11. \[ \text{val lengthofRest} = \text{length restStateSpace} \]

12. \[ \text{if (lengthofRest} \geq 4) \text{ andalso (csContant} = \text{true} \text{ andalso (checkSR} = \text{true}) \text{ then} \]

13. \[ \text{CheckProperty2(restStateSpace, DID)} \]

14. \[ \text{else} \]

15. \[ \text{if (csContant) andalso (checkSR) then} \]

16. \[ \"\text{Property 2(Termination of a Dialogue) is Satisfied}\" \]

17. \[ \text{else} \]

18. \[ \text{if not (csContant) then} \]

19. \[ \"\text{Property 2(Termination of a Dialogue) is not Satisfied: There is a}\" \]

20. \[ \text{problem in the agent's commitment store}\" \]

21. \[ \text{else} \]

22. \[ \"\text{Property 2(Termination of a Dialogue) is not Satisfied: There is a}\" \]

23. \[ \text{problem in the who to terminated the dialogue}\" \]

24. **End**  
   \[ \text{CheckProperty2} \]

25. **Create&Save**  
   \[ \text{Property2 result file} \]

---

**Figure 6.4: Property 2 as an SML Function**

h) \text**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) \text**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 \text**TDID** where:
a. \textit{DIDL} represents the expected termination message for the specific role;

b. \textit{DIDEf} represents the effect of the termination message to the sender commitment store \textit{CS} (e.g. \textit{DIDEf} = "Add Topic");

c. \textit{DIDAID} represents the expected agent ID of the termination message sender;

d. \textit{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. \textit{checkSenderReceiver} function is used to compare the sender and receiver of the termination message;

b. \textit{proponent} and \textit{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. \textit{checkSR} variable represents the \textit{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 \textit{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. \textit{checkTheContantofCS} 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 location (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))
   andalso (senderM1 = receiverM2) andalso (receiverM1 = senderM2)
   then checkProperty3Part1(restArcsList)
10. else
11.   if (restLength >= 3) andalso ((role1 = role2))
12.     andalso ((senderM1 = senderM2) andalso (receiverM1 = receiverM2))
13.     then checkProperty3Part1(restArcsList)
14.   else
15.     if (not (role1 = role2)) andalso (senderM1 = receiverM2)
16.       andalso (receiverM1 = senderM2)
17.       then true
18.     else
19.       false
20.   End checkProperty3Part1
21. **Return Back to** CheckPropert3AllTN
22. if (turnTaking= true) then
23.   "Property 3(Turn Taking) is Satisfied"
24. else
25.   "Property 2(Turn Taking) is not Satisfied"
26. **end CheckPropert3AllTN**
27. **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:
Bridging the Specification Protocol Gap in Argumentation

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 \( \text{restArcsList} \) and save it in \( \text{restLength} \).

(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 \( \text{restArcsList} \), then recall the \( \text{checkProperty3Part1} \) function (recurs).

(10) Line 23: Return the control back to \( \text{CheckProperty3AllTN} \) function.

(11) Lines 24 to 27: Check the result of the comparison (\( \text{turnTaking} \) variable). A positive (negative) result indicates that Property3 is satisfied (unsatisfied).

(12) Line 29: Create the Property3 result file and write the result of \( \text{CheckProperty3AllTN} \).
**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.
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 locotion 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, Openingmessage)
19. val checkTerminationM = checkAllTeminatedMessages(DIDRecursiveMessages, TNodes)
20. if (not (checkopeningDM)) andalso (not (checkTerminationM)) then
21.  "Property 5(Recursive Message) is Satisfied"
22. else
23.  "Property 5(Recursive Message) is not Satisfied"
24. end CheckProperty5
25. **Create&Save**  Property5 result file

**Figure 6.7: Property 5 as an Standard ML Function**

a) The target location is not the starting or opening location in the state space;

b) The target location is not the terminating location in the state space.

(4) Line 4: Function inputs are SS and DIDRecursiveMessages.

(5) Line 5: Extract the starting locations information from SS and save this information in Openingmessage. Then extract the termination locations information from SS and save this information in TNodes.

(6) Lines 6 and 7: Check if the recursive location in the DID is a Starting Location in the state space, where:

a) findElementInTheList function is used to check if the recursive location in the DID is a Startin Location in the state space;

b) checkopeningDM represents the findElementInTheList function result. It is considered true if the recursive location in the DID is a Starting Location in the state space.

Chapter 6: Verification Method based Coloured Petri Nets and SML
Bridging the Specification Protocol Gap in Argumentation

(7) Lines 8 and 9: Check if the recursive locution in the DID is a Termination Locution in the state space, where:

   a) \textit{checkAllTerminatedMessages} function is used to check if the recursive locution in the DID is a Termination Locution in the state space;

   b) \textit{checkTerminationM} represents the \textit{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(\textit{checkopeningDM} and \textit{checkTerminationM}). A positive (negative) result indicates that Property5 is satisfied (unsatisfied).

(9) Line 15: Create Property5 result file and write the result of \textit{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.

\section*{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.
7.1 Architecture

The synthesis protocol tool \((\text{GenerateLCCProtocol})\) tool has been designed and implemented in the Java programing 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);
By using Design Patterns (LCC-Argument Patterns)

Part 1 (Synthesis of Concrete Protocols)

1. DID (Argument Specification Language)

Chapter 4

2. By using Design Patterns (LCC-Argument Patterns)

Chapter 5

Resulting LCC Protocol (LCC = Multi-agents Development Language)

Part 2 (Verification Model)

3. Automatic Creation

Chapter 6

1. Automatic transformation

DID Properties files

Automatic verification

Verification model

(General behaviour property checking code in CPN SML)

Result

True /False

State space

CPNXML

Construction

2

Chapter 7: Design and Implementation 190
(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)`;
# Creating a DID Diagram Process

1. Creates the DID diagram for a persuasion dialogue using 'create new DID diagram' screen in the GenerateLCCProtocol tool

# Synthesising Concrete LCC Protocols Process

2. Synthesises the LCC protocol of the persuasion dialogue using 'Generate concrete LCC protocols' button in the GenerateLCCProtocol tool

# Verification Process

3. Specifies agents Knowledge Base (KB) using 'Agents KB' button in the GenerateLCCProtocol tool

4. Transforms the LCC protocol of the persuasion dialogue into an equivalent CPN model as well as creates the DID properties files using 'Create CPN File' button in the GenerateLCCProtocol tool

5. Constructs the state space of the CPN model of the LCC protocol of the persuasion dialogue using 'the CPN state space tool' palette in the CPN Tool

6. Applies the verification model using 'test a Text as ML Code(ML!)' icon in the simulation tool palette in the CPN Tool

7. Displays the verification model result of the five basic properties using 'Verification Model Result' button in the GenerateLCCProtocol tool

Figure 7.3: An Example Scenario of GenerateLCCProtocol Tool
Figure 7.4: Create New Dialogue Interaction Diagram Example (Claim Locution Icon)

(2)-b) Locution Name = \textit{claim}(T)

(2)-a) Locution Type = \textit{Starting}

(2)-c) Sender information

(3) Level Number = 1

(2)-d) Receiver information

(4) ’Add locution to level (level 1)

Figure 7.5: Create New Dialogue Interaction Diagram Example (Add Locution Formal Definition to DID)

Locution Formal definition (DID textual representation)

Reply to locution (structural rules)
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 DIDs 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).
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).
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 agent'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.
Figure 7.9: Specifying Agents Knowledge Base Screens

2- Click on ‘Ok’ button

3- Add the Knowledge Base for both agents

2- Click on ‘Agent KB’ button

4- Click on ‘Add Agent1 and Agent 2 KB’
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.
Figure 7.10: Transforming LCC Protocol into an Equivalent CPN Model Screens
(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 Verification Model Result

To display the verification 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 is 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
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 \(I1 \, ^\text{flies}(P)\) with its premises \(I2\) and \(I3\)).

The \(RA2\) node links \(L4\) and its propositional content \(I5\) and \(I6\) (the \(RA2\) node connects \(I4 \, ^\text{~flies}(P)\) 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 \(\text{claim}(I1)\) in \(L1\) locution.

2. \(DID\) diagram (in Figure 4.3 in chapter 4) specifies that \(A2\) can reply with \(\text{why}(T)\) or \(\text{concede}(T)\).

3. \(A2\) sends \(\text{why}(I1)\) in \(L2\).

4. \(DID\) diagram (in Figure 4.3 in chapter 4) specifies the legal replies \(\text{argue}(\beta)\) where \(\beta\)'s conclusion is \(T\), or \(\text{retract}(T)\).

5. \(A1\) responds to the challenge by declaring the supporting premises \(I2\) and \(I3\) for \(I1\) [sends \(\text{argue}(I2 \text{ and } I3)\) in \(L3\) node]. Note that \(A1\) satisfies the \(\text{argue}\) message constraint \(\text{Pre}=\text{findPremise}(T, KB_\beta, CS_\beta)\) using AIF which describes the relation between \(A1\)'s argument \((T=\text{Tweety flies in } I1)\) and 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(β), argue(β) 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,KBO,CSO) 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(β), argue(β) 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);
Represent Locutions

<table>
<thead>
<tr>
<th>DID</th>
<th>Modgil and McGinnis [Modgil and McGinnis, 2007]</th>
<th>Reed et al. [Reed et al., 2008]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locutions are represented in the form of Locution icon.</td>
<td>Expands I-nodes content to represent locution</td>
<td>Locutions are represented in the form of L-nodes</td>
</tr>
</tbody>
</table>

Dialogue Games Concepts

<table>
<thead>
<tr>
<th>DID</th>
<th>Modgil and McGinnis work has the following dialogue games concepts: locutions, pre-conditions and structural rules.</th>
<th>AIF+ has the following dialogue games concepts: locutions, pre and post conditions and structural rules.</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td>Modgil and McGinnis work has the following dialogue games concepts: locutions, pre-conditions and structural rules.</td>
<td>AIF+ has the following dialogue games concepts: locutions, pre and post conditions and structural rules.</td>
</tr>
</tbody>
</table>

Protocol automated synthesis

<table>
<thead>
<tr>
<th>DID</th>
<th>Modgil and McGinnis work has the following dialogue games concepts: locutions, pre-conditions and structural rules.</th>
<th>AIF+ has the following dialogue games concepts: locutions, pre and post conditions and structural rules.</th>
</tr>
</thead>
<tbody>
<tr>
<td>The user can perform fully automated synthesis of multi-agent protocols using LCC–Argument patterns</td>
<td>The user cannot perform fully automated synthesis of multi-agent protocols</td>
<td>The user cannot perform fully automated synthesis of multi-agent protocols</td>
</tr>
</tbody>
</table>

Argument Format

<table>
<thead>
<tr>
<th>DID</th>
<th>Modgil and McGinnis work has the following dialogue games concepts: locutions, pre-conditions and structural rules.</th>
<th>AIF+ has the following dialogue games concepts: locutions, pre and post conditions and structural rules.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DID can work with any argument format</td>
<td>Only AIF</td>
<td>Only AIF</td>
</tr>
</tbody>
</table>

Table 8.1: Differences between Modgil and McGinnis [Modgil and McGinnis, 2007], Reed et al. [Reed et al., 2008] and DID

![DID Location icon](image)

![Modgil and McGinnis I-node](image)

![Reed et al. L-node](image)

Figure 8.3: Locution Concepts

Chapter 8: Evaluation and Discussion
• 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;
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
Bridging the Specification Protocol Gap in Argumentation

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...
level 2 (Figures 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 experiences 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 itself\(^{20}\).

(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.
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 is 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...
A 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 representaion of the DID in order to be able to edit the DID digram. 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 form 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 synthesis\(^\text{21}\) 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".
Bridging the Specification Protocol Gap in Argumentation

- **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 participants: '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):**

<table>
<thead>
<tr>
<th>Act</th>
<th>ActType (Act)</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>{Starting}</td>
</tr>
<tr>
<td>challenge</td>
<td>{ Intermediate }</td>
</tr>
<tr>
<td>accept</td>
<td>{Termination}</td>
</tr>
<tr>
<td>refuse</td>
<td>{Termination}</td>
</tr>
<tr>
<td>justify</td>
<td>{Intermediate}</td>
</tr>
<tr>
<td>promise</td>
<td>{Intermediate}</td>
</tr>
</tbody>
</table>

(10) **Replies(Act):**

In the persuasion dialogue the Replies rules are as follows:

<table>
<thead>
<tr>
<th>Act</th>
<th>Replies(Act)</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>request(R)</td>
<td>{challenge(R), accept(R), refuse(R)}</td>
<td>R= missing resource for the speaker</td>
</tr>
<tr>
<td>challenge(R)</td>
<td>{justify(R,S)}</td>
<td>S= support for R</td>
</tr>
<tr>
<td>accept(R)</td>
<td>Ø</td>
<td></td>
</tr>
<tr>
<td>refuse(R)</td>
<td>Ø</td>
<td></td>
</tr>
<tr>
<td>justify(R,S)</td>
<td>{refuse(R), promise(R'',R')}</td>
<td>R''= missing resource for the speaker and R''= new resource for new plan for the audience</td>
</tr>
<tr>
<td>promise(R'',R')</td>
<td>{accept(R'',R'), refuse(R'',R')}</td>
<td></td>
</tr>
</tbody>
</table>
(11) **PreC(Act,KB,CS):**

Let Player = A

In the negotiation dialogue the Pre-conditions are as follows:

<table>
<thead>
<tr>
<th>Act</th>
<th>PreC(Act,KB,CS)</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>request(R)</td>
<td>miss(KB(<em>A</em>),R) = true</td>
<td>• <em>miss</em> function returns true if agent A misses a resource R for a plan P to achieve a goal G.</td>
</tr>
</tbody>
</table>
| challenge(R) | \( (\text{have}(KB\(_A_\),R) \land \text{need}(KB\(_A_\),R) = \text{true}) \lor \text{notHave}(KB\(_A_\),R) = \text{true}) \lor \text{missResource}(KB\(_A_\),P,G) = \text{true} \) | • *have* function returns true if agent A has a resource R.  
  • *need* function returns true if agent A has a resource R needed for a plan P to achieve a goal G.  
  • *notHave* function returns true if agent A does not have a resource R.  
  • *missResource* function returns true if agent A needs R's resource for a plan P to achieve a goal G. |
| accept(R)    | have(KB\(_A_\),R) = true and \text{notNeed}(KB\(_A_\),R) = \text{true} and \text{notmissResource}(KB\(_A_\),P,G) = \text{true} and gaveAway(CS\(_A_\),R) = \text{true} | • *have* function returns true if agent A has a resource R.  
  • *notNeed* function returns true if agent A has a resource R which is not needed for a plan P to achieve a goal G.  
  • *notmissResource* function returns true if agent A does not miss a resource R for a plan P to achieve a goal G.  
  • *gaveAway* function always returns true and results in agent A giving away a resource R (agent A subtract R from its commitment store CS\(_A_\)). |
| refuse(R)    | \( (\text{notHave}(KB\(_A_\),R) \lor \text{need}(KB\(_A_\),R) = \text{true}) \land \text{notmissResource}(KB\(_A_\),P,G) = \text{true} \) | These pre-conditions must be satisfied in order for A to move refuse after request move where,  
  • *notHave* function returns true if agent A does not have a resource R.  
  • *need* function returns true if agent A has a resource R needed for a plan P to achieve a goal G.  
  • *notmissResource* function returns true if agent A does not miss a resource R for a plan P to achieve a goal G. |
<table>
<thead>
<tr>
<th>Act</th>
<th>PreC(Act,KB,CS)</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>refuse(R)</td>
<td>missResource(KB_A,P,G) = true and notExistAlternativePlane(G, without(R,R')) = true</td>
<td>These pre-conditions must be satisfied in order for A to move refuse after justify move where,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• missResource function returns true if agent A needs R’ resource for a plan P to achieve a goal G.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• notExistAlternativePlane function returns true if agent A cannot find an alternative plan for agent B’s goal without R and R’.</td>
</tr>
<tr>
<td>justify(R,S)</td>
<td>miss(KB_A,R) = true and getPlan(KB_A,P) = true and getGoal(KB_A,G) = true</td>
<td>• miss function returns true if agent A needs R resource for a plan P to achieve a goal G.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• getPlan function returns true if agent A is able to find a plan P in its Knowledge Base KB_A (A needs R resource for a plan P to achieve a goal G).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• getGoal function returns true if agent A is able to find a goal G in its Knowledge Base KB_A (A needs R resource for a plan P to achieve a goal G).</td>
</tr>
<tr>
<td>promise(R”,R”)</td>
<td>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</td>
<td>• R”= missing resource for the speaker A and R”= new resource for new plan for the audience B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• missResource function returns true if agent A needs R’ resource for a plan P to achieve a goal G.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• have function returns true if agent A has a resource R&quot;.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• notNeed function returns true if agent A has a resource R” which is not needed for a plan P to achieve a goal G.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• choosealternativeplane function returns true if agent A finds a new and different plan NewPlan for other agent B’s goal that requires neither of R and R’ and needs R”.</td>
</tr>
<tr>
<td>refuse(R”,R’)</td>
<td>miss(KB_A,R) = true and notChooseBetterPlan(KB_A,G, NewPlan, oldPlan, without(R,R’), with(R”)) =true</td>
<td>These pre-conditions must be satisfied in order for A to move refuse after promise move where,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• miss function returns true if agent A has a resource R needed for a plan OldPlan (P) to achieve a goal G.</td>
</tr>
</tbody>
</table>
### Appendix A: Negotiation Dialogue

#### PostC(Act, KB, CS):

Let Player(M₀) = A and NextPlayer = B.

In a negotiation dialogue the Post-Conditions (conditions for receiver player B of M₀) are as follows:

<table>
<thead>
<tr>
<th>Act</th>
<th>PostC(Act, KB, CS)</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>request(R)</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>challenge(R)</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>accept(R)</td>
<td>obtained(CSₐ,R) = true</td>
<td>• obtained function always returns true and results in agent B obtaining a resource R (agent B adding R to its commitment store CSₐ).</td>
</tr>
</tbody>
</table>

- **notChooseBetterPlan** function compares the `OldPlan` and the `NewPlan` and returns true if `NewPlan` is not acceptable.
- `accept(R'', R')` These pre-conditions must be satisfied in order for A to move accept after promise move where,
  - miss function returns true if agent A has a resource R needed for a plan `OldPlan (P)` to achieve a goal G.
  - have function returns true if agent A has a resource R'.
  - notNeed function returns true if agent A has a resource R' which is not needed for a plan P to achieve a goal G.
  - chooseBetterPlan function compare the `OldPlan` and the `NewPlan` and returns true if agent the `NewPlan` (that requires neither of R and R' and needs R'') is acceptable.
  - gaveAway function always returns true and results in agent A giving away a resource R' (agent A subtract R' from its commitment store CSₐ).
  - obtained function always returns true and results in agent A obtaining a resource R'' (agent A adding R'' to its commitment store CSₐ).
Table 1: Legal Moves

<table>
<thead>
<tr>
<th>Act</th>
<th>PostC(Act,KB,CS)</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>refuse(R)</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>justify(R,S)</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>promise(R'',R')</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>
| accept(R'',R')         | obtained(CS_B, R') = true and giveaway(CS_B,R'') = true | These post-conditions must be satisfied in order for A to move accept after promise move where,  
  - giveaway function always returns true and results in agent B giving away a resource R'' (agent B subtract R'' from its commitment store CS_B).  
  - obtained function always returns true and results in agent B obtaining a resource R' (agent B adding R' to its commitment store CS_B). |

(13) **LegalMoves( M, CS_A, CS_B)**

From Figure A.1 the negotiation dialogue, we can see that:

- Dialogues open by making a request move
  
  \[
  M_1 = \text{initial move, ActType(Act(M_1))} = \text{Starting and Act(M_1)} = \{\text{request}\}
  \]

- In the negotiation dialogue, the argument terminates once the agents send accept or refuse. In other words, both accept and refuse \(\in\{\text{Termination}\}\). There is no reply move to these moves (there are no arrows coming out from these moves).

- Challenge, justify and promise \(\in\{\text{Intermediate}\}\). 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 \(M_1\) then Player = A,

  b) else NextPlayer = B  iff  Player = A

  and  NextPlayer = A  iff  Player = B
Figure A.1: The Negotiation Dialogue Legal Moves

- **M1**: A, request
- **M2**: B, challenge
- **M3**: B, accept
- **M4**: B, refuse
- **M5**: A, justify
- **M6**: B, refuse
- **M7**: B, promise
- **M8**: A, accept
- **M9**: A, refuse
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]). 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 follows the Commitment rules in Amogud et al [Amogud et al. 2000] work.)
Appendix A: Negotiation Dialogue

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
(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.
Appendix A: Negotiation Dialogue

**KB_A** = { ("have","picture"), ("have","hammer"), ("plan-Obtain","nail"), ("plan-Goal","hang a picture"), ("goal","hung a picture"), ("missing","nail"), ("better-Plan-Obtain", "screw and screwdriver"), ("better-Plan-Goal","hang a picture") }

**KB_B** = { ("have","mirror"), ("have","nail"), ("have","screw"), ("have","screwdriver"), ("plan-Obtain","hammer"), ("plan-Goal","hang a mirror"), ("goal","hung a mirror"), ("missing","hammer"), ("promise-Plane-Obtain", "screw and screwdriver"), ("promise-Plan-Goal", "hang a picture") }

---

**Figure A.4: The Picture Hanging Example**

A

1. request(Can you please give me a nail?)

2. challenge(why do you need a nail?)

3. justify(Because I want to hang a picture and for that I need a nail)

4. promise(But you can 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 the hammer.)

5. accept(Really, I guess in that case, I do not need the hammer. Here you go.)
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 \texttt{justify("Because I want to hang a picture and for that I need a nail")} locution.

\( B \) 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 \texttt{promise("But you can 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.

\( A \) 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 \texttt{accept("Really, I guess in that case, I do not need the nail. Here you go")} locution.

The commitment stores of \( A \) and \( B \) at the end of the dialogue are:

\[
\begin{align*}
\text{CS}_A &= \{("\text{gaveAway","hammer"}), ("\text{obtained","screw and screwdriver"})\} \\
\text{CS}_B &= \{("\text{obtained","hammer"}), ("\text{gaveAway","screw and screwdriver"})\}
\end{align*}
\]

**A.5 LCC Synthesis Protocol of the Negotiation Dialogue**

This section represents the generated LCC protocol from the automated agent protocol synthesis tool "\texttt{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 \texttt{request(R)}. 

Bridging the Specification Protocol Gap in Argumentation

<table>
<thead>
<tr>
<th>Agent A</th>
<th>Agent B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(requestSender&lt;sub&gt;A&lt;/sub&gt;(KB&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,ID&lt;sub&gt;B&lt;/sub&gt;,ID&lt;sub&gt;A&lt;/sub&gt;)) ::=</td>
<td>a(requestReceiver&lt;sub&gt;B&lt;/sub&gt;(KB&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,ID&lt;sub&gt;A&lt;/sub&gt;,ID&lt;sub&gt;B&lt;/sub&gt;)) ::=</td>
</tr>
<tr>
<td>request(R) =&gt; a(requestReceiver&lt;sub&gt;B&lt;/sub&gt;(KB&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,ID&lt;sub&gt;A&lt;/sub&gt;),ID&lt;sub&gt;B&lt;/sub&gt;)</td>
<td>request(R) &lt;= a(requestSender&lt;sub&gt;A&lt;/sub&gt;(KB&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,ID&lt;sub&gt;B&lt;/sub&gt;,ID&lt;sub&gt;A&lt;/sub&gt;),ID&lt;sub&gt;B&lt;/sub&gt;)</td>
</tr>
<tr>
<td>then</td>
<td>then</td>
</tr>
<tr>
<td>a(replyToRequestReceiver&lt;sub&gt;A&lt;/sub&gt;(KB&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,R,ID&lt;sub&gt;B&lt;/sub&gt;,ID&lt;sub&gt;A&lt;/sub&gt;)) ::=</td>
<td>a(replyToRequestReceiver&lt;sub&gt;B&lt;/sub&gt;(KB&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,ID&lt;sub&gt;A&lt;/sub&gt;,ID&lt;sub&gt;B&lt;/sub&gt;)) ::=</td>
</tr>
<tr>
<td>obtained(CS&lt;sub&gt;A&lt;/sub&gt;,R) &lt;= accept(R) &lt;=</td>
<td>accept(R) =&gt;</td>
</tr>
<tr>
<td>a(replyToRequestSender&lt;sub&gt;B&lt;/sub&gt;(KB&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,R,ID&lt;sub&gt;A&lt;/sub&gt;),ID&lt;sub&gt;B&lt;/sub&gt;)</td>
<td>a(replyToRequestReceiver&lt;sub&gt;A&lt;/sub&gt;(KB&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,R,ID&lt;sub&gt;B&lt;/sub&gt;,ID&lt;sub&gt;A&lt;/sub&gt;),ID&lt;sub&gt;B&lt;/sub&gt;),ID&lt;sub&gt;A&lt;/sub&gt;)</td>
</tr>
<tr>
<td>or</td>
<td>or</td>
</tr>
<tr>
<td>refuse(R) &lt;=</td>
<td>refuse(R) &lt;=</td>
</tr>
<tr>
<td>a(replyToRequestSender&lt;sub&gt;B&lt;/sub&gt;(KB&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,R,ID&lt;sub&gt;A&lt;/sub&gt;),ID&lt;sub&gt;B&lt;/sub&gt;)</td>
<td>a(replyToRequestReceiver&lt;sub&gt;A&lt;/sub&gt;(KB&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,R,ID&lt;sub&gt;B&lt;/sub&gt;,ID&lt;sub&gt;A&lt;/sub&gt;),ID&lt;sub&gt;B&lt;/sub&gt;),ID&lt;sub&gt;A&lt;/sub&gt;)</td>
</tr>
<tr>
<td>(</td>
<td></td>
</tr>
<tr>
<td>challenge(R) &lt;=</td>
<td>challenge(R) &lt;=</td>
</tr>
<tr>
<td>a(replyToRequestSender&lt;sub&gt;B&lt;/sub&gt;(KB&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,R,ID&lt;sub&gt;A&lt;/sub&gt;),ID&lt;sub&gt;B&lt;/sub&gt;)</td>
<td>a(replyToRequestReceiver&lt;sub&gt;A&lt;/sub&gt;(KB&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,R,ID&lt;sub&gt;B&lt;/sub&gt;,ID&lt;sub&gt;A&lt;/sub&gt;),ID&lt;sub&gt;B&lt;/sub&gt;),ID&lt;sub&gt;A&lt;/sub&gt;)</td>
</tr>
<tr>
<td>then</td>
<td>then</td>
</tr>
<tr>
<td>a(replyToChallengeSender&lt;sub&gt;A&lt;/sub&gt;(KB&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,R,ID&lt;sub&gt;B&lt;/sub&gt;),ID&lt;sub&gt;A&lt;/sub&gt;)) ::=</td>
<td>a(replyToChallengeReceiver&lt;sub&gt;B&lt;/sub&gt;(KB&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,R,ID&lt;sub&gt;A&lt;/sub&gt;),ID&lt;sub&gt;B&lt;/sub&gt;)) ::=</td>
</tr>
<tr>
<td>justify(R,P,G) =&gt;</td>
<td>justify(R,P,G) &lt;=</td>
</tr>
<tr>
<td>a(replyToChallengeReceiver&lt;sub&gt;B&lt;/sub&gt;(KB&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,R,ID&lt;sub&gt;A&lt;/sub&gt;),ID&lt;sub&gt;B&lt;/sub&gt;),ID&lt;sub&gt;B&lt;/sub&gt;)) &lt;= miss(KB&lt;sub&gt;B&lt;/sub&gt;,R) and getGoal(KB&lt;sub&gt;A&lt;/sub&gt;,G) and getPlan(KB&lt;sub&gt;A&lt;/sub&gt;,P)</td>
<td>a(replyToChallengeSender&lt;sub&gt;A&lt;/sub&gt;(KB&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,R,ID&lt;sub&gt;B&lt;/sub&gt;),ID&lt;sub&gt;A&lt;/sub&gt;),ID&lt;sub&gt;B&lt;/sub&gt;)) &lt;= ( (have(KB&lt;sub&gt;B&lt;/sub&gt;,R) and notNeed(KB&lt;sub&gt;B&lt;/sub&gt;,R)) and notmissResource(KB&lt;sub&gt;B&lt;/sub&gt;,P,G) and gaveAway(CS&lt;sub&gt;B&lt;/sub&gt;,R) ) and</td>
</tr>
<tr>
<td>then</td>
<td></td>
</tr>
<tr>
<td>a(replyToJustifyReceiver&lt;sub&gt;A&lt;/sub&gt;(KB&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,R,ID&lt;sub&gt;B&lt;/sub&gt;))</td>
<td>a(replyToJustifySender&lt;sub&gt;B&lt;/sub&gt;(KB&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,R,ID&lt;sub&gt;A&lt;/sub&gt;),ID&lt;sub&gt;B&lt;/sub&gt;))</td>
</tr>
<tr>
<td>a(replyToChallengeReceiver&lt;sub&gt;B&lt;/sub&gt;(KB&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,R,ID&lt;sub&gt;A&lt;/sub&gt;),ID&lt;sub&gt;B&lt;/sub&gt;)) ::=</td>
<td>a(replyToChallengeSender&lt;sub&gt;A&lt;/sub&gt;(KB&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;A&lt;/sub&gt;,CS&lt;sub&gt;B&lt;/sub&gt;,R,ID&lt;sub&gt;B&lt;/sub&gt;),ID&lt;sub&gt;A&lt;/sub&gt;),ID&lt;sub&gt;B&lt;/sub&gt;)) ::=</td>
</tr>
</tbody>
</table>

**Figure A.5(a): Generated LCC Protocol**

Appendix A: Negotiation Dialogue 247
### Bridging the Specification Protocol Gap in Argumentation

#### Appendix A: Negotiation Dialogue

<table>
<thead>
<tr>
<th>Agent A</th>
<th>Agent B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a(replyToJustifyReceiver, A(KB_A, CS_A, CS_B, R, ID_B), ID_A) := )</td>
<td>( a(replyToJustifySender, A(KB_B, CS_B, CS_A, R, ID_A, ID_B) := )</td>
</tr>
</tbody>
</table>
| refuse(R) \(\Leftarrow\) \( a(replyToJustifySender, A(KB_B, CS_B, CS_A, R, ID_A, ID_B) \) | refuse(R) \(\Rightarrow\) \( a(replyToJustifyReceiver, A(KB_A, CS_A, CS_B, R, ID_B), ID_A) \)
| or \((\) | or \((\)
| \( \text{addToCS}(CS_A, R') \leftarrow \) promise(R', R') \(\Leftarrow\) \( a(replyToJustifySender, B(KB_B, CS_B, CS_A, R, ID_B), ID_A) \) | missResource(KB_B, P, G) and notExistAlternativePlane(KB_B, R', Without(R, R')) \)
| then \( a(replyToPromiseSender, A(KB_A, CS_A, CS_B, R, R', R'', ID_B), ID_A) \) | then \( a(replyToPromiseReceiver, B(KB_B, CS_B, CS_A, R, R', R'', ID_B), ID_A) \)
| \( a(replyToPromiseSender, A(KB_A, CS_A, CS_B, R, R', R'', ID_B), ID_A) := \) | \( a(replyToPromiseReceiver, B(KB_B, CS_B, CS_A, R, R', R'', ID_B), ID_A) := \)
| ( | (|
| accept(R', R') \(\Rightarrow\) \( a(replyToPromiseReceiver, B(KB_B, CS_B, CS_A, R, R', R'', ID_B), ID_A) \) | ( | (|
| miss(KB_A, R) and | obtained(CS_B, R') and gaveaway(CS_B, R') \)
| have(KB_A, R') and notNeed(KB_A, R') and | chooseBetterPlan(KB_B, G, NewPlan, OldPlan, Without(R, R'), With(R')) and gaveaway(CS_A, R') \) and obtained(CS_A, R') \)
| chooseBetterPlan(KB_A, G, NewPlan, OldPlan, Without(R, R'), With(R')) \) and gaveaway(CS_A, R') and obtained(CS_A, R') \) | \) or \( |
| or \( ( | or \( ( |
| refuse(R', R') \(\Rightarrow\) \( a(replyToPromiseReceiver, B(KB_B, CS_B, CS_A, R, R', R'', ID_B), ID_A) \) | refuse(R', R') \(\Rightarrow\) \( a(replyToPromiseReceiver, B(KB_B, CS_B, CS_A, R, R', R'', ID_B), ID_A) \)
| \) ). | )).

Figure A.5(b): Generated LCC Protocol
(2) The tool then 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 request(R) icon, starting from the top-down and moving left to right.

(4) Moves to the next level (level two of the DID of the negotiation dialogue).

(5) Following this, the tool selects the *Termination-Intermediate Pattern*.

(6) Applies the *Termination-Intermediate Pattern*.

(7) Moves to the next level in the DID and repeats steps 4 and 6. Note that the automated synthesis process finishes when the tool matches the last level (level five) in the DID of the negotiation dialogue with the *Termination-Intermediate Pattern*.

### A.6 Verification Model of the LCC Synthesis Protocol of the Negotiation Dialogue

In this section, we will give a brief description of how to verify the semantics of the DID of a negotiation dialogue (shown in Figure A.2) against the semantics of the synthesised LCC protocol (shown in Figures A.5(a) and A.5(b)). In this example, the initial marking of:

1. **OpenDialogue place** = "request a nail". This place represents dialogue game topic.

2. **A place** = ("IDA", [ ], [ ("have", "picture"), ("have", "hammer"), ("planObtain", "nail"), ("planGoal", "hang picture"), ("goal", "hung picture"), ("missing", "nail"), ("betterPlanObtain", "screw"), ("betterPlanGoal", "hang picture") ], "requestSenderA", "",", "", [ ], "IDB", "", "", ""). This place represents agent A arguments.

3. **B place** = ("IDB", [ ], [ ("have", "mirror"), ("have", "nail"), ("have", "screw"), ("have", "screwdriver"), ("planObtain", "hammer"), ("planGoal", "hang mirror"), ("planGoal", "hang mirror"),...
Bridging the Specification Protocol Gap in Argumentation

Appendix A: Negotiation Dialogue

("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, replyToJustifySenderA, 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).
Figure A.6: The requestSenderA CPN Subpage

Figure A.7: The requestReceiverB CPN Subpage
Figure A.8: The replyToRequestSenderB CPN Subpage
Appendix A: Negotiation Dialogue

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
Appendix A: Negotiation Dialogue
Figure A.16: The protocol CPN Superpage
Step Three: Automated Creation of DID Properties

Bridging the Specification Protocol Gap in Argumentation

Figure A.21: Intermediate Locutions File

Figure A.22: Termination Locutions File

Termination Locutions Effect CS File

Effective CS Files

Figure A.23: Termination Locutions Effect CS and Effective CS Files

Figure A.24: Player Types File

Figure A.25: Player Ids File

Appendix A: Negotiation Dialogue
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 considered to be 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
Bridging the Specification Protocol Gap in Argumentation

Appendix A: Negotiation Dialogue

Figure 2.9: C-successful Dialogue Example

The commitment stores of agents $A$ and $B$ at the end of the dialogue are:

$CS_A = \{("obtained", R"), ("gaveAway", R")\}$

$CS_B = \{("obtained", R"), ("gaveAway", R")\}$

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) Line11: 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.
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, ... A_n\},

Where,

- \( A_i \in \text{Players}, \) where \( i=1,2,3, ..., n \)

- 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-agente Act Type**

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

\[ \text{ActType: Acts } \rightarrow \varphi(\text{Types}) \]

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.

\[ \text{ReC: Acts } \times \varphi(\text{args(L)}) \rightarrow \text{Boolean} \]

**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.

\[ \text{DC: Acts } \times \varphi(\text{Players}) \times (\text{args(L)})^n \times (\text{args(L)})^n \rightarrow \text{Boolean} \]

**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 $\rightarrow \varnothing$(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,
- $\text{SetM}_{t-1} \in \varnothing(\text{Moves}) \cup \{\text{null}\}$ represents the previous moves ($M_t$ is a reply to $\text{SetM}_{t-1}$),
- $\text{setPlayer}_j \in \varnothing(\text{Players})$ represents the next players in the dialogue,
- $\text{sender}_t \in \text{Roles}$ represents the role identifier of player (sender agent),
- $\text{rSetRole}_e \in \varnothing(\text{Roles})$ represents the role identifiers of the $\text{setPlayer}_j$ (receiver agents),

(2) One sender agent and one receiver agent:

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

$$M_t = (\text{player}_t, \text{act}_t, \text{SetM}_{t-1}, \text{player}_j, \text{sRole}_t, \text{rRole}_t),$$

Where,

- $\text{player}_t \in \text{Players}$ represents the player of the move,
• act_t ∈ Acts represents the speech act performed in the move,

• player_t ≠ player_j,

• player_j ∈ Players represents the next player in the dialogue,

• SetM_{t-1} ∈ ϕ (Moves) ∪ {null} represents the previous moves (M_t is a reply to SetM_{t-1}),

• sRole_s ∈ Roles represents the role identifiers of the player_t (sender agent),

• rRole_{player} ∈ 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 ∈ Moves, t >= 1, is defined as:

M_t = (setPlayer_t, act_t, SetM_{t-1}, player_j, sSetRole_s, rRole_t),

Where,

• setPlayer_t ∈ ϕ(Players) represents the players of the move,

• act_t ∈ Acts represents the speech act performed in the move,

• SetM_{t-1} ∈ ϕ (Moves) ∪ {null} represents the previous moves (M_t is a reply to SetM_{t-1}),

• player_t ∈ Players represents the next player of the move,

• player_t ∉ setPlayer_j

• sSetRole_s ∈ ϕ(Roles) represents the role identifiers of the player_t sender agents,

• rRole_t ∈ 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:

\[ \text{LegalMovesNAgent: } \text{MoveSeq} \times (\varnothing(\text{args}(L)))^n \rightarrow \varnothing(\text{Moves}) \]

Rule 4: (Start N-agent Dialogue)

This rule says that a N-agent dialogue always starts with a RecursiveStarting act by proposal agent:

\[ \text{LegalMovesNAgent( } \text{[ ] , CS}_{1}, \text{CS}_{2}, ....... \text{CS}_{n}) = \{ \text{M}_1 \} \]

Where,

- \( \text{M}_1 = (\text{proposal, act}_1, \text{null, setPlayer}_j, \text{sRole}_\text{proposal1, rSetRole}_1) \),
- \( \text{proposal} \notin \text{setPlayer}_j \)
- \( \text{ActType} (\text{act}_1) = \{ \text{RecursiveStarting} \} \),
- \( \text{PreC} (\text{act}_1, \text{KB}_\text{proposal, CS}_\text{proposal}) = \text{true} \), 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 player \( \in \text{setPlayer}_j \)), where \( \text{KB}_j \) represents agent knowledge base and \( \text{CS}_j \) 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:

\[ \text{LegalMovesNAgent ( } \text{[M}_1, \text{M}_2, ....... \text{M}_t \} , \text{CS}_1, \text{CS}_2, ....... \text{CS}_n) = \text{SetM}_{t+1} \]

if

- \( \text{M}_t = (\text{proposal, act}_t, \text{SetM}_{t-1}, \text{setPlayer}_j, \text{sRole}_\text{proposal}, rSetRole_t) \),

Appendix B: N-Agent Dialogue 270
Bridging the Specification Protocol Gap in Argumentation

- \( \text{proposal} \notin \text{setPlayer}_j \)

- \( \text{PreC}(\text{act}_t, \text{KB}_{\text{proposal}}, \text{CS}_{\text{proposal}}) = \text{true} \), 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 player \( \in \text{setPlayer}_j \)), where \( \text{KB}_j \) represents agent knowledge base and \( \text{CS}_j \) 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 player \( \in \text{setPlayer}_j \)), where \( \text{KB}_j \) represents agent knowledge base and \( \text{CS}_j \) 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 \( \text{SetM}_{t+1} \) and the receiver of move \( M_t \) is the sender of all the move from the \( \text{SetM}_{t+1} \).

**Rule 6: (N-agent Dialogue Termination)**

This rule says that a N-agent dialogue always terminates with a RecursiveTermination act by the proposal agent:

\[
\text{LegalMovesNAgent}((M_1, M_2, \ldots, M_t), \text{CS}_1, \text{CS}_2, \ldots, \text{CS}_n) = \emptyset
\]
if

- \( M_t = (\text{proposal}, \text{act}_t, M_{t-1}, \text{null}, \text{sRole}_{\text{proposal}}, \text{rSetRole}_t) \),

- \( \text{ActType}(\text{act}_t) = \{\text{RecursiveTermination}\} \),

- \( \text{PreC}(\text{act}_t, KB_{\text{proposal}}, CS_{\text{proposal}}) = \text{true} \), where \( KB_{\text{proposal}} \) represents proposal agent's knowledge base and \( CS_{\text{proposal}} \) represents proposal agent's commitment store.

- \( \text{PostC}(\text{act}_t, KB_j, CS_j) = \text{true} \) (for each player \( \in \text{setPlayer}_j \), \( \text{setPlayer}_j \) represents the previous players and 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):

\[
\text{LegalMovesNAgent} ( [M_1, M_2, \ldots, M_t] , CS_1, CS_2, \ldots, CS_n) = \{M_{t+1}\}
\]

- \( M_t = (\text{proposal}, \text{act}_t, M_{t-1}, \text{setPlayer}_j, \text{sRole}_{\text{proposal}}, \text{rSetRole}_t), \)

- \( \text{ActTypes}(\text{act}_t) = \{\text{Divided}\}, \)

- 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*),

- \( \text{PreC}(\text{act}_t, KB_{\text{proposal}}, CS_{\text{proposal}}) = \text{true} \), where \( KB_{\text{proposal}} \) represents proposal agent's knowledge base and \( CS_{\text{proposal}} \) represents proposal agent's commitment store,
Bridging the Specification Protocol Gap in Argumentation

- PostC act_i, KB_j, CS_j = true (for each player ∈ setPlayer_j), where KB_j represents agent knowledge base and CS_j represents agent commitment store,

- DC act_i, Players, SetKB, SetCS = true, where
  - each player_i ∈ Players has KB_i ∈ SetKB and has CS_i ∈ SetCS
  - KB_i represents agent knowledge base
  - and CS_i represents agent commitment store
  - i = 1, 2, .....n

**Rule 8: (Return Back to Dialogue Between N-agent)**

This rule says that:

LegalMovesNAgent ([M_1, M_2, ....... M_{t+1}], CS_1, CS_2, ....... CS_n) = {M_{t+2}}

If

- M_{t+1} is a move in dialogue between two agents
- M_{t+1} = (player_i, act_{t+1}, M_{t+1}, null, sRole_{t+1}, rRole_{t+1}),
- ActType(act_{t+1}) = {Termination},
- PreC act_{t+1}, KB_i, CS_i = true, where KB_i represents agent I's knowledge base and CS_i represents agent I's commitment store.
- PostC act_{t+1}, KB_k, CS_k = true, where KB_k represents agent K's knowledge base and CS_k represents agent K's commitment store.
- M_{t+2} = (proposal, act_{t+2}, M_{t+1}, setPlayer_j, sRole_proposal, rSetRole_{t+2}),
- M_{t+2} is a move in dialogue between N-agent
- ActTypes(act_{t+2}) = {RecursiveStarting}
Bridging the Specification Protocol Gap in Argumentation

- proposal \in \text{setPlayer}_j

- \text{PreC}(\text{act}_{t+2}, \text{KB}_{\text{prop}}; \text{CS}_{\text{prop}}) = \text{true}, \text{ where } \text{KB}_{\text{prop}} \text{ represents proposal agent's knowledge base and } \text{CS}_{\text{prop}} \text{ represents proposal agent's commitment store.}

- \text{PostC}(\text{act}_{t+2}, \text{KB}_j; \text{CS}_j) = \text{true} \quad (\text{for each player } \in \text{setPlayer}_j), \text{ where } \text{KB}_j \text{ represents agent knowledge base and } \text{CS}_j \text{ 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 \textit{proposal(Topic)} locution to all other agents.

- Each agent who receives the \textit{proposal(Topic)} reports acceptance of the \textit{proposal(Topic)} by sending an \textit{accept(Topic)} locution or rejection of the \textit{proposal(Topic)} by sending a \textit{reject(Topic)} locution.

- If the agents reach an agreement (if Acceptance number \geq The number of supporter agents), the proposal sends \textit{reachAgreement(Topic)} to all other agents.

- If the agents could not reach an agreement on the \textit{proposal(Topic)}, the proposal divides agents into groups composed of two agents and sends \textit{argueWith} locution to all other agents to inform them about the groups.

**DID formal definition for a persuasion dialogue between N-agent**

1. **Players:** Players=\{Agent_1, Agent_2,..........Agent_n\}

Each player has its own KB and CS such that:
Appendix B: N-Agent Dialogue
Agent_1 argumentation system \( AS_{Agent_1} = \{ KB_{Agent_1}, CS_{Agent_1} \} \)

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

\[
\text{Acts} = \{ \text{proposal(Topic), accept(Topic), reject(Topic), reachAgreement(Topic), argueWith(Topic,Agent_P,Agent_O)} \}
\]

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

<table>
<thead>
<tr>
<th>Act</th>
<th>ActType (Act)</th>
</tr>
</thead>
<tbody>
<tr>
<td>proposal(Topic)</td>
<td>{ RecursiveStarting }</td>
</tr>
<tr>
<td>accept(Topic)</td>
<td>{ Intermediate }</td>
</tr>
<tr>
<td>reject(Topic)</td>
<td>{ Intermediate }</td>
</tr>
<tr>
<td>reachAgreement</td>
<td>{ RecursiveTermination }</td>
</tr>
<tr>
<td>argueWith(Topic)</td>
<td>{ Divided }</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Act</th>
<th>Replies(Act)</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>proposal(Topic)</td>
<td>{ accept(Topic), reject(Topic) }</td>
<td></td>
</tr>
<tr>
<td>accept(Topic)</td>
<td>{ reachAgreement(Topic), argueWith(Topic) }</td>
<td></td>
</tr>
<tr>
<td>reject(Topic)</td>
<td>{ reachAgreement(Topic), argueWith(Topic) }</td>
<td></td>
</tr>
<tr>
<td>reachAgreement(Topic)</td>
<td>( \emptyset )</td>
<td></td>
</tr>
<tr>
<td>argueWith(Topic,Agent_P,Agent_O)</td>
<td>{ claim(Topic) }</td>
<td>Replies(Act) for argueWith locution represents the Starting Locuution icon in the DID for two agents (e.g. Replies(Act)= claim(Topic) which represents the Starting Locuution icon in the persuasion dialogue between two agents in section 4.2.1). In other words, we need to connect argueWith with the Starting Locuution icon in the DID for two agents.</td>
</tr>
</tbody>
</table>
(5) PreC(Act, KB, CS):

Let’s Player = Proposal

<table>
<thead>
<tr>
<th>Act</th>
<th>PreC(Act, KB, CS)</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>proposal(Topic)</td>
<td>addTopicToCS(Topic, CSProposal) = true</td>
<td></td>
</tr>
<tr>
<td>accept(Topic)</td>
<td>findTopicInKB(Topic, KBProposal) = true and notFindTopicInCS(Topic, CSProposal) = true and notFindOppTopicInCS(not(Topic), CSProposal) = true and addTopicToCS(Topic, CSProposal) = true</td>
<td></td>
</tr>
<tr>
<td>reject(Topic)</td>
<td>notFindTopicInKB(Topic, KBProposal) = true and notFindTopicInCS(Topic, CSProposal) = true</td>
<td></td>
</tr>
<tr>
<td>reachAgreement(Topic)</td>
<td>greaterThanOrEqual(NAccepting, NSupporters) = true</td>
<td>greaterThanOrEqual function returns true if the number of accepting agents NAccepting is greater than or equal to the number of supporter agents NSupporters. (NAccepting &gt;= NSupporters)</td>
</tr>
<tr>
<td>argueWith(Topic, AgentP, AgentO)</td>
<td>lessThan(NAccepting, NSupporters) = true and IsNotEmpty(RejectionList) = true and IsNotEmpty(AcceptingList) = true</td>
<td>• lessThan function returns true if the number of accepting agents NAccepting is less than the number of supporter agents NSupporters. (NAccepting &lt; NSupporters) • IsNotEmpty function returns true if the list is not empty.</td>
</tr>
</tbody>
</table>
### (6) PostC(Act, KB, CS):

<table>
<thead>
<tr>
<th>Action</th>
<th>PostC(Act, KB, CS)</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>proposal(Topic)</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>
| accept(Topic)   | addToAcceptingList (AcceptingList, AccList, ID) = true  
|                 | and                                  | increaseAccepting (NAccepting, NAcc) = true  
|                 | and                                  | addIDToList(AgentList, SendingList, ID) = true                       | • *addToAcceptingList* function always returns true and results in proposal agent adding the accepting agent ID to the *AcceptingList*  
|                 |                                      |                                                                      |
| reject(Topic)   | addToRejectingList (RejectingList, RejList, ID) = true  
|                 | and                                  | increaseRejecting (NRejecting, NRej) = true  
|                 | and                                  | addIDToList(AgentList, SendingList, ID) = true                       | • *addToRejectingList* function always returns true and results in proposal agent adding the rejecting agent ID to the *RejectingList*  
| reachAgreement  | true                                |                                                                      |
| argueWith(Topic, AgentP, AgentO) | true |                                                                      |

### (7) ReC(Act, KB, CS):

<table>
<thead>
<tr>
<th>Action</th>
<th>PostC(Act, KB, CS)</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Player(Mt) = Proposal</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Bridging the Specification Protocol Gap in Argumentation

#### Appendix B: N-Agent Dialogue

<table>
<thead>
<tr>
<th>Act</th>
<th>ReC(Act, KB, CS)</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>proposal(Topic)</td>
<td>getAgentIDFromList (AgentList, otherAgents, ID) = true</td>
<td>getAgentIDFromList function gets agent ID from the AgentList and puts the remaining agents in the otherAgents list (OtherAgents = AgentList – {ID})</td>
</tr>
<tr>
<td>accept(Topic)</td>
<td>notEqual(AgentList, SendingList)</td>
<td>notEqual function compare the AgentList with the Sending and returns true if these two lists are equal</td>
</tr>
<tr>
<td>reject(Topic)</td>
<td>notEqual(AgentList, SendingList)</td>
<td></td>
</tr>
<tr>
<td>reachAgreement(Topic)</td>
<td>getAgentIDFromList (AgentList, otherAgents) = true</td>
<td></td>
</tr>
<tr>
<td>argueWith(Topic, AgentP, AgentO)</td>
<td>creatOneAgentGroups (RejectingList, Re, AcceptingList, Ac, AgentGroup, AGroup, P, O) = true</td>
<td>creatOneAgentGroups function: (4) creates one agent group by getting one agent O from the RejectingList and one agent P from the AcceptingList. (5) adds the new agents groups to AGroup list (AGroup = AgentGroup + {(P, O)}). (6) Saves the remained rejection agent in Re list and saves the remained accepting agents in Ac.</td>
</tr>
</tbody>
</table>

### (8) LegalMovesNAgent(Mt, CSAgent1, CSAgent2, ..., CSAgentN)

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 ∈ {Intermediate}. There are several moves to these moves. (there are arrows coming out from these moves).
- After argueWith ∈ {Divided}, the dialogue between two agents begins.
The turn-taking between participants switches after each move (the agents take it in turns to make moves):

- if \( M_1 \) then \( \text{Player} = \text{Proposal} \),
- else \( \text{NextPlayer} = \text{All other agents} \) iff \( \text{Player} = \text{Proposal} \) and \( \text{NextPlayer} = \text{Proposal} \) iff \( \text{Player} = \text{All other agents} \)
**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
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:
Bridging the Specification Protocol Gap in Argumentation

- If an agent finds that "A's car is safe", it sends \textit{accept("My car is safe")} to \(A\),

- If an agent does not find "A's car is safe", it sends \textit{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 \textit{a reachAgreement("My car is safe")} locution to all other agents.

- If the number of rejections is equal or greater than one (\textit{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 \textit{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 \textit{accept("My car is safe")} or \textit{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>= 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>= 3 agents;
- Or, use this pattern when one agent wants to send a message to more than one agent.

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:

\[
\text{a(R1}_{\text{Sender}}(\text{AgentList}, \text{NAgent}, \text{Topic}), \text{ID}_{\text{Sender}}) := \]

\[
\begin{align*}
\text{R}_{\text{Sender}} & \quad \text{RSL}_{\text{>>}} \quad \text{R}_{\text{Receiver}} \\
\text{then} \\
\quad ( & \text{a(R2}_{\text{sender}}(\text{OtherAgents}, \text{NAgent}, \text{Topic}), \text{ID}_{\text{sender}}) \leftarrow \text{FailureRecursiveC} \\
& \text{or} \quad \text{a(R1}_{\text{sender}}(\text{OtherAgents}, \text{NAgent}, \text{Topic}), \text{ID}_{\text{sender}}) \\
) .
\end{align*}
\]

\[
\text{a(R1}_{\text{Receiver}}(\text{KBId}, \text{CSId}, \text{ID}_{\text{Sender}}), \text{ID}) := \]

\[
\begin{align*}
\text{R}_{\text{Receiver}} & \quad <_{\text{RSL}} \quad \text{R}_{\text{Sender}} \\
\text{then} \\
\quad \text{a(R2}_{\text{Receiver}}(\text{KBId}, \text{CSId}, \text{ID}_{\text{Sender}}), \text{ID}).
\end{align*}
\]
Figure B.6: Recursive Starting(Sending) Pattern Solution
Bridging the Specification Protocol Gap in Argumentation

This pattern represents a generic recursive clause. In this pattern, and in the rest of the patterns, \textit{RRL} represents Recursive Receiving Locution, '\textless\Rightarrow\textgreater' represents outgoing messages from a role, and '\textless\Leftarrow\textgreater' represents incoming messages. \textit{FailureRecursiveC} represent a condition when it is true the recursive end (usually, \textit{FailureRecursiveC} is a condition over \textit{AgentList}).

In this LCC code, there are two roles: \textit{R1}\textsubscript{Sender} and \textit{R1}\textsubscript{Receiver}. The \textit{R1}\textsubscript{Sender} role of the sender agent \textit{ID}_{\text{sender}} has three input parameters: (1) \textit{AgentList} which represents the agents ID list; (2) \textit{NAgent} which represents the number of agents (note that the number of agents is \(\geq 3\)). (3) \textit{Topic} to open dialogue. The \textit{R1}\textsubscript{Sender} role begins by sending a Recursive Starting locution \textit{RSL} to the \textit{R1}\textsubscript{Receiver} role (the '\textless\Rightarrow\textgreater' symbol indicates that the \textit{R1}\textsubscript{Sender} role may send one or more different \textit{RSL}s to the \textit{R1}\textsubscript{Receiver} role.). Then, the \textit{R1}\textsubscript{Sender} role check \textit{FailureRecursiveC}. If this condition is true, the \textit{R1}\textsubscript{Sender} changes its role to the \textit{R2}\textsubscript{Sender}, otherwise, it recurse.

The \textit{R1}\textsubscript{Receiver} role of receiver agent \textit{ID}_{\text{Receiver}} has three input parameters: (1) \textit{KB}_{\text{Receiver}} which represents the agent knowledge base list; (2) \textit{CS}_{\text{Receiver}} which represents the agent commitment store list. Note that \textit{CS}_{A} is empty, since \textit{R1}\textsubscript{Receiver} represents the first role of the audience agent in the LCC protocol; (3) \textit{ID}_{\text{Sender}} which represents the sender agent identifier. The \textit{R1}\textsubscript{Receiver} role begins by receiving a Recursive Starting locution \textit{RSL} from \textit{R1}\textsubscript{Sender}. Then, it changes its role to the \textit{R2}\textsubscript{Receiver}.

\textbf{Rewriting methods:}

\textit{First (Sending method): Rewriting of the "\textit{R}\textsubscript{Sender} \textit{RSL}\Rightarrow \textit{R}\textsubscript{Receiver}"}

The main function of rewriting is to allow generic relations between the \textit{R}\textsubscript{Sender} and the \textit{R}\textsubscript{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 "\textit{R}\textsubscript{Sender} \textit{RSL}\Rightarrow \textit{R}\textsubscript{Receiver}", then it is possible to specialise it within two different statements:

\textit{Rewrite 1: (one location)}

We might specialise "\textit{R}\textsubscript{Sender} \textit{RSL}\Rightarrow \textit{R}\textsubscript{Receiver}" to an interaction statement that sends a \textit{RRL(Topic)} message to agent \textit{ID}_{\text{Receiver}}, which is achieved by the \textit{RecursiveC} and \textit{C1}.
constraints. In practice, \textit{RecursiveC} represents a recursive condition (usually, \textit{RecursiveC} is a condition over \textit{AgentList}), \textit{C1} represents a condition (\textit{C1} may represent more than one condition that is connected by \textit{or} and \textit{and} operators) that must be satisfied in order for a sender agent to send the Recursive Starting locution (usually, \textit{C1} is a condition over \textit{Topic}).

\[
\text{RSL(Topic)} \Rightarrow a(\text{Receiver}(\text{KBID,CSID,ID}_{\text{Sender}}), \text{ID}) \leftarrow \text{RecursiveC and C1}
\]

\textbf{Rewrite 2: (multiple location)}

We might specialise "\textit{R}_{\text{Sender}}^{\text{RSL}} \approx \textit{R}_{\text{Receiver}}" to an interaction statement that sends a \textit{RSL(Topic)} message to agent \textit{ID}_{\text{Receiver}} which is achieved by the constraints \textit{RecursiveC} and \textit{C1}. Then, there is another relation between \textit{R}_{\text{Sender1}} and \textit{R}_{\text{Receiver1}}.

\[
\text{RSL(Topic)} \Rightarrow a(\text{receiver}(\text{KBID,CSID,ID}_{\text{Sender}}), \text{ID}) \leftarrow \text{RecursiveC and C1}
\]

or

\[
\text{R}_{\text{Sender}}^{\text{RSL}} \approx \text{R}_{\text{Receiver}}
\]

\textbf{Second (Receiving method): Rewriting of the " \textit{R}_{\text{Receiver}} ^{\text{RSL}} \approx \textit{R}_{\text{Sender}} "

If there is a general relation "\textit{R}_{\text{Receiver}} ^{\text{RSL}} \approx \textit{R}_{\text{Sender}} ", then it is possible to specialise it within two different statements:

\textbf{Rewrite 1: (one location)}

We might specialise " \textit{R}_{\text{Receiver}} ^{\text{RSL}} \approx \textit{R}_{\text{Sender}} " to an interaction statement that receive a \textit{RSL(Topic)} message from agent \textit{ID}_{\text{Sender}}. \textit{C2} represents a condition that must be satisfied after receiver agent receives the Recursive Sending locution. In practice, \textit{C2} may represent more than one condition that is connected by \textit{or} and \textit{and} operators. Usually, \textit{C2} is a condition over the role arguments (e.g. \textit{KB} and \textit{CS}).

\[
\text{C2} \leftarrow \text{RSL(Topic)} \leq a(\text{Sender}(\text{KB}_{\text{Sender}}, \text{CS}_{\text{Sender}}, \text{ID}_{\text{Receiver}}),\text{ID}_{\text{Sender}})
\]

\textbf{Rewrite 2: (multiple location)}

We might specialise " \textit{R}_{\text{Receiver}} ^{\text{RSL}} \approx \textit{R}_{\text{Sender}} " to an interaction statement that receive a \textit{RSL(Topic)} message from agent \textit{ID}_{\text{Sender}}. Then, there is another relation between \textit{R}_{\text{Sender1}} and \textit{R}_{\text{Receiver1}}.
Pattern 7:

**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:**

\[
a(R1_{\text{Sender}}(\text{KB,CS,Topic,ID}_{\text{Receiver}}), \text{ID}_{\text{Sender}}) := \]

\[
R_{\text{Sender}} \overset{\text{RRL} \Rightarrow}{\rightarrow} R_{\text{Receiver}}
\]

then

\[
a(R2_{\text{Sender}}(\text{KB,CS,Topic,ID}_{\text{Receiver}}), \text{ID}_{\text{Sender}}).
\]

\[
a(R1_{\text{Receiver}}(\text{AgentList, SendingList, NAgent,Topic,ID}_{\text{Receiver}}) :=
\]

\[
R_{\text{Receiver}} \overset{\text{RRL}}{\Leftrightarrow} R_{\text{Sender}}
\]

then

(\[
\]

\[
a(R1_{\text{Receiver}}(\text{AgentList, OtherSendingLists, NAgent,Topic,ID}_{\text{Receiver}})
\]

\[
\text{RecursiveC}
\]

\[
\text{or}
\]

\[
a(R2_{\text{Receiver}}(\text{AgentList,OtherASendingLists, NAgent,Topic,ID}_{\text{Receiver}})
\]

).
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: R1Sender and R1Receiver. The R1Sender 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) IDReceiver which represents the receiver agent identifier.

The R1Receiver 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 \( \geq 3 \)). (4) Topic which represents the dialogue game topic. The R1Receiver role begins by receiving RRL from R1Sender.

The R1Receiver role begins by receiving a RRL message from the R1Sender role (the '<<' symbol indicates that the R1Receiver role may receive one or more different RRLs from the R1Sender role). Then, the R1Receiver role check RecursiveC. If this condition is true, the R1Receiver recurse, otherwise, it changes its role to the R2Receiver.

**Rewriting methods:**

**First (Sending method): Rewriting of the "RSender RRL ≻ RReceiver"**

If there is a general relation "RSender RRL ≻ RReceiver", then it is possible to specialise it within two different statements:

**Rewrite 1: (one locution)**

We might specialise "RSender RRL ≻ RReceiver" to an interaction statement that sends a RRL(Topic) message to agent IDReceiver, 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).

\[
\text{RRL(Topic) } \Rightarrow \text{a(R_Receiver(AgentList, NAgent, N, Topic), IDReceiver) } \leftarrow \text{C1}
\]
Rewrite 2: (multiple locution)

We might specialise "\(R_{\text{Sender}} \text{RRL} \approx> R_{\text{Receiver}}\)" to an interaction statement that sends a \(RRL(Topic)\) message to agent \(ID_{\text{Receiver}}\), which is achieved by the constraint \(C1\). Then, there is another relation between \(R_{\text{Sender1}}\) and \(R_{\text{Receiver1}}\).

\[
\begin{align*}
RRL(Topic) & \Rightarrow a( R_{\text{Receiver}}(\text{AgentList}, \text{NAgent}, \text{N}, \text{Topic}), ID_{\text{Receiver}}) \leftarrow C1 \\
\text{or} \\
R_{\text{Sender}} \text{RRL} \approx> & R_{\text{Receiver}}
\end{align*}
\]

Second (Receiving method): Rewriting of the "\(R_{\text{Receiver}} \approx< \text{RRL} R_{\text{Sender}}\)"

If there is a general relation "\(R_{\text{Receiver}} \approx< \text{RRL} R_{\text{Sender}}\)" then it is possible to specialise it within two different statements:

Rewrite 1: (one location)

We might specialise "\(R_{\text{Receiver}} \approx< \text{RRL} R_{\text{Sender}}\)" to an interaction statement that receive a \(RRL(Topic)\) message from agent \(ID_{\text{Sender}}\). \(C2\) 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).

\[
C2 \leftarrow \text{RRL(Topic)} \leq a(R_{\text{Sender1}}(\text{KB}_{ID}, \text{CS}_{ID}, \text{ID}_{\text{Receiver}}), \text{ID}_{\text{Sender1}})
\]

Rewrite 2: (multiple locution)

We might specialise "\(R_{\text{Receiver}} \approx< \text{RRL} R_{\text{Sender}}\)" to an interaction statement that receive a \(RRL(Topic)\) message from agent \(ID_{\text{Sender}}\). Then, there is another relation between \(R_{\text{Sender1}}\) and \(R_{\text{Receiver1}}\).

\[
\begin{align*}
C2 & \leftarrow \text{RRL(Topic)} \leq a(R_{\text{Sender1}}(\text{KB}_{ID}, \text{CS}_{ID}, \text{ID}_{\text{Receiver}}), \text{ID}_{\text{Sender1}}) \\
\text{or} \\
R_{\text{Receiver1}} & \approx< \text{RRL} R_{\text{Sender1}}
\end{align*}
\]
**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:**

\[ a(R_{\text{sender}}(\text{AgentList}, \text{Topic}), ID_{\text{sender}}) ::= \]

\[
\begin{align*}
& (R_{\text{Sender}} \, \text{RTL} \rightarrow \, R_{\text{Receiver}}) \\
& \text{then} \\
& (\text{null} \leftarrow \text{FailureRecursiveC1} \\
& \text{or} \\
& a(R_{\text{sender}}(\text{OtherAgents}, \text{Topic}), ID_{\text{sender}})) \\
& ) \\
& \text{or} \\
& (R_{\text{Sender}} \, \text{RSL} \rightarrow \, R_{\text{Receiver}}) \\
& \text{then} \\
& (a(R_{2_{\text{sender}}}(\text{OtherAgents}, \text{Topic}), ID_{\text{sender}}) \\
& \leftarrow \text{FailureRecursiveC2} \\
& \text{or} \\
& a(R_{\text{sender}}(\text{OtherAgents}, \text{Topic}), ID_{\text{sender}})) \\
& ). \\
\]

\[ a(R_{\text{Receiver}}(ID_{\text{KB}}, ID_{\text{CS}}, ID_{\text{Sender}}), ID) ::= \]

\[
\begin{align*}
R_{\text{Receiver}} & \equiv RTL \, \, R_{\text{Sender}} \\
\text{or} \\
(R_{\text{Receiver}} & \equiv RSL \, \, R_{\text{Sender}} \\
\text{then} \\
& a(R_{2_{\text{Receiver}}}(ID_{\text{KB}}, ID_{\text{CS}}, ID_{\text{Sender}}), ID) ) \\
). \\
\]

This pattern represents a generic recursive clause. \textit{RTL} represents the Recursive Termination locution and \textit{FailureRecursiveC} represents a condition that when it is true forces the recursion to end (usually, \textit{FailureRecursiveC} is a condition over \textit{AgentList}).

In this LCC code, there are two roles: \textit{R}_{\text{Sender}} and \textit{R}_{\text{Receiver}}. The \textit{R}_{\text{Sender}} role of the sender agent \textit{ID}_{\text{sender}} has two input parameters: \textit{AgentList} and \textit{Topic}. It begins by
either: (1) sending a Recursive Termination locution. Then, the $R_{Sender}$ role checks $FailureRecursiveC_1$. 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 '≐' symbol indicates that the $R_{Sender}$ role may send one or more different $RSL$s to the $R_{Receiver}$ role.). Then, the $R_{Sender}$ role check $FailureRecursiveC_2$. If this condition is true, the $R_{Sender}$ changes its role to the $R_{2Sender}$, 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 location from $R_{Sender}$ (the '≐' symbol indicates that the $R_{Receiver}$ role may receive one or more different RTLs from the $R_{Sender}$ role); or (2) a Recursive Starting location $RSL$ from $R_{Sender}$. Then, it changes its role to the $R_{2Receiver}$.

**Rewriting methods:**

**First (Sending Termination method): Rewriting of the "$R_{Sender}^{RTL} ≐> R_{Receiver}$"**

If there is a general relation "$R_{Sender}^{RTL} ≐> R_{Receiver}$", then it is possible to specialise it within two different statements:

**Rewrite 1: (one termination location)**

We might specialise "$R_{Sender}^{RTL} ≐> R_{Receiver}$" 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. 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 location)

We might specialise "\( R_{\text{Sender}}^{\text{RTL} \Rightarrow} R_{\text{Receiver}} \)" to an interaction statement that sends a \( \text{RTL}(\text{Topic}) \) Recursive Termination message to agent \( ID_{\text{Receiver}} \) which is achieved by the \( \text{RecursiveC} \) and \( C1 \) constraints. Then, there is another termination relation between \( R_{\text{Sender}} \) and \( R_{\text{Receiver}} \).

\[
\begin{align*}
\text{RTL}(\text{Topic}) & \Rightarrow a(R_{\text{Receiver}}(KB_{\text{ID}_{\text{Sender}}}, CS_{\text{ID}_{\text{Sender}}}, ID_{\text{ID}}), ID) \leftarrow \text{RecursiveC and C1} \\
\text{Or} \\
R_{\text{Sender}}^{\text{RTL} \Rightarrow} R_{\text{Receiver}}
\end{align*}
\]

Second (Receiving Termination method): Rewriting of the \( R_{\text{Receiver}} \Leftarrow^{\text{RTL}} R_{\text{Sender}} \)

If there is a general relation \( R_{\text{Receiver}} \Leftarrow^{\text{RTL}} R_{\text{Sender}} \), then it is possible to specialise it within two different statements:

Rewrite 1: (one location)

We might specialise " \( R_{\text{Receiver}} \Leftarrow^{\text{RTL}} R_{\text{Sender}} \)" to an interaction statement that receive a \( \text{RTL}(\text{Topic}) \) message from agent \( ID_{\text{Sender}} \). \( C2 \) represents a condition that must be satisfied after receiver agent receives the Recursive Termination location. In practice, \( C2 \) may represent more than one condition that is connected by \( \text{or} \) and \( \text{and} \) operators. Usually, \( C2 \) is a condition over the role arguments (e.g. \( KB \) and \( CS \)).

\[
C2 \leftarrow \text{RTL}(\text{Topic}) \Leftarrow a(R1_{\text{Sender}}(KB_{\text{Sender}}, CS_{\text{Sender}}, ID_{\text{Receiver}}, ID_{\text{Sender}}))
\]

Rewrite 2: (multiple location)

We might specialise " \( R_{\text{Receiver}} \Leftarrow^{\text{RTL}} R_{\text{Sender}} \)" to an interaction statement that receive a \( \text{RTL}(\text{Topic}) \) message from agent \( ID_{\text{Sender}} \). Then, there is another relation between \( R_{\text{Sender1}} \) and \( R_{\text{Receiver1}} \).

\[
\begin{align*}
C2 & \leftarrow \text{RTL}(\text{Topic}) \Leftarrow a(R1_{\text{Sender}}(KB_{\text{Sender}}, CS_{\text{Sender}}, ID_{\text{Receiver}}, ID_{\text{Sender}})) \\
\text{Or} \\
R_{\text{Receiver}} \Leftarrow^{\text{RSL}2} R_{\text{Sender}}
\end{align*}
\]

Third (Sending method): Rewriting of the \( R_{\text{Sender}}^{\text{RSL} \Rightarrow} R_{\text{Receiver}} \)

See rewriting method of Recursive Sending Pattern (Rewriting of the \( R_{\text{Sender}}^{\text{RSL} \Rightarrow} R_{\text{Receiver}} \)).
Fourth(Receiving method): Rewriting of the " $R_{Receiver} \preccurlyeq^{RSL} R_{Sender}$ "

See rewriting method of Recursive Sending Pattern (Rewriting of the " $R_{Receiver} \preccurlyeq^{RSL} R_{Sender}$ ").

**Pattern9:**

**Name:** Recursive Termination-Divided Pattern (RTDP)

**Problem:** How to divide agents into groups of two or terminate an argument (dialogue) for N>=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: \(R_{Sender, Termination}\), \(R_{Sender, divideGroup}\) 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 \(divideGroup\) 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 ‘\(\Rightarrow\)’ 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 \(FailureRecursiveC\). If this condition is true, the \(TerminationR_{Sender}\) terminates, otherwise, it recurse;

The \(divideGroup\) 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
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-Discourse Pattern), otherwise, it recurses.

The RReceiver role of audience agent ID has three input parameters: (1) KBReceiver which represents the agent knowledge base list; (2) CSReceiver which represents the agent commitment store list.; (3) IDSender which represents the sender agent identifier. The RReceiver role begins by either: (1) receiving a Recursive Termination locution from TerminationSender (the '≈' symbol indicates that the RReceiver role may receive one or more different RTLs from the TerminationSender role); (2) receiving an argueWith message from divideGroupSender. Then, it changes its role to the startDID role(see chapter 5, Move-To-Discourse Pattern).

Rewriting methods:

First (Sending Termination method): Rewriting of the "TerminationSender RTL ≈> RReceiver"

See rewriting method of Recursive Termination-Sending Pattern (Rewriting of the "Sender RTL ≈> RReceiver").

Second (Receiving Termination method): Rewriting of the "RReceiver ≈> RTL TerminationSender"

See rewriting method of Recursive Termination-Sending Pattern (Rewriting of the "RReceiver ≈> RTL 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).

\[
\begin{align*}
\text{a(R1}_{\text{Sender}} (& \text{AgentList, NAgent, Topic), ID}_{\text{Sender}}) ::= & \\
R_{\text{Sender}} & \xrightarrow{RSL} \quad R_{\text{Receiver}} \\
\text{then} & \\
\text{a(R2}_{\text{Sender}} (& \text{OtherAgents, NAgent, Topic), ID}_{\text{Sender}}) & \leftarrow \text{FailureRecursiveC} \\
\text{or} & \\
\text{a(R1}_{\text{Sender}} (& \text{OtherAgents, NAgent, Topic), ID}_{\text{Sender}}) & \
\text{a(R1}_{\text{Receiver}} (& \text{AgentList, SendingList, NAgent, Topic), ID}_{\text{Receiver}}) ::= & \\
R_{\text{Receiver}} & \xleftarrow{RRL} \quad R_{\text{Sender}} \\
\text{then} & \\
\text{a(R1}_{\text{Receiver}} (& \text{AgentList, OtherSendingLists, NAgent, Topic), ID}_{\text{Receiver}}) & \leftarrow \text{RecursiveC} \\
\text{or} & \\
\text{a(R2}_{\text{Receiver}} (& \text{AgentList, OtherASendingLists, NAgent, Topic), ID}_{\text{Receiver}}) & 
\end{align*}
\]
Rewriting methods:

First (Sending method): Rewriting of the "\( R_{Sender}^{\text{RSL}} \approx > R_{Receiver} \)"

See rewriting method of Recursive Sending Pattern (Rewriting of the "\( R_{Sender}^{\text{RSL}} \approx > R_{Receiver} \)").

Second (Receiving method): Rewriting of the "\( R_{Receiver} \approx <= \text{RRL} R_{Sender} \)"

See rewriting method of Recursive Receiving Pattern (Rewriting of the "\( R_{Receiver} \approx <= \text{RRL} R_{Sender} \)").

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
3. **Select&Save** Pattern= one pattern from the LCC-Argument patterns for general N-agent protocol library
4. **If** (Pattern has rewriting methods) then
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)
10. **If** (Pattern = *Recursive Termination-Divided*) then
11. **Use** Recurs-To-N-Discussion Pattern
12. recursNumber = number of Termination locution icon in the DID for two agents
13. **If** (reurseNumber = 0) then
14. **Select&Save** RewriteMethod2= *Rewrite 1*
15. **Match** (Termination Icon, Recurs-To-N-Discussion Pattern, RewriteMethod2)
16. **Else**
17. **Loop begin** (if i=1)
18. **Select&Save** RewriteMethod2= *Rewrite 2*
19. **Match** (Termination Icon, Recurs-To-N-Discussion 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
25. **Output** LCC protocol

---

**Table B.1 Relationship Between Locution Type and Patterns**

<table>
<thead>
<tr>
<th>Locution Type</th>
<th>Pattern Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recursive Starting Locution</td>
<td>Recursive Starting (Sending) Pattern</td>
</tr>
<tr>
<td>Recursive Receiving Locution</td>
<td>Recursive Receiving Pattern</td>
</tr>
<tr>
<td>Recursive Termination Locution and Divided</td>
<td>Recursive Termination-Divided</td>
</tr>
<tr>
<td>Locution</td>
<td></td>
</tr>
<tr>
<td>Recursive Termination Locution and Recursive</td>
<td>Recursive Termination-Sending Pattern</td>
</tr>
<tr>
<td>Starting Locution</td>
<td></td>
</tr>
</tbody>
</table>

**Figure B.9: N- Agents Protocol Automated Synthesis Algorithm**

Appendix B: N-Agent Dialogue
(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 **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 x 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 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
Bridging the Specification Protocol Gap in Argumentation

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(FirstReceiverRole_ID (KB_ID,CS_ID, ID_proposal), 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:
Bridging the Specification Protocol Gap in Argumentation

Appendix B: N-Agent Dialogue

<table>
<thead>
<tr>
<th>Proposal</th>
<th>Other Agents</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(proposalSender_proposal_protocol(AgentList,NAgent,NSupporters,Topic), ID) := proposal(Topic) =&gt; a(proposalReceiver_ID(KB_ID,CS_ID, ID))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>proposal(Topic) &lt;&lt;= getAgentIDFromList(AgentList,otherAgents,ID) and addTopicToCS(Topic,CS_proposal)</td>
</tr>
<tr>
<td></td>
<td>then (a(replyToProposalReceiver_protocol, AgentList,NAgent, NSupporters,Topic), ID) &lt;= agentListEmpty(AgentList)</td>
</tr>
<tr>
<td></td>
<td>or a(proposalSender_protocol(OtherAgents,NAgent, NSupporters,Topic), ID) := proposal(Topic) &lt;&lt;= a(proposalSender_protocol(AgentList,NAgent, NSupporters,Topic), ID)</td>
</tr>
<tr>
<td></td>
<td>then a(replyToProposalSender(KB_ID,CS_ID, Topic, ID), ID).</td>
</tr>
</tbody>
</table>

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).
### Appendix B: N-Agent Dialogue

<table>
<thead>
<tr>
<th>Proposal</th>
<th>Other agents</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a(\text{replyToProposalReceiver}<em>{\text{proposal}}(\text{AgentList}, \text{NAgent}, \text{NSupporters}, \text{Topic}, \text{AcceptingList}, \text{RejectionList}, \text{NAccepting}, \text{NRejection}, \text{SendingList}), \text{ID}</em>{\text{proposal}}) ::= )</td>
<td>( a(\text{replyToProposalSender}<em>{\text{ID}}(\text{KB}</em>{\text{ID}}, \text{CS}<em>{\text{ID}}, \text{Topic}, \text{ID}</em>{\text{proposal}}), \text{ID}) ::= )</td>
</tr>
<tr>
<td>( add\text{ID}ToList(SendingList, OtherSedingList, ID) and add\text{ToAcceptingList}(\text{AcceptingList}, AccList, ID) and increase\text{Accepting}(\text{NAccepting}, NAcc) and RejList= \text{RejectionList} and NRej is N\text{Rejection} \equiv \text{accept(Topic)} &lt;= a(\text{replyToProposalSender}<em>{\text{ID}}(\text{KB}</em>{\text{ID}}, \text{CS}<em>{\text{ID}}, \text{Topic}, \text{ID}</em>{\text{proposal}}), \text{ID}) ) or</td>
<td>( accept(Topic) =&gt; a(\text{replyToProposalReceiver}<em>{\text{proposal}} (</em> , _ , _ , _ , _ , _ , _ , _ , _ , _ , _ , <em>, ID</em>{\text{proposal}}) \equiv \text{findTopicInKB(Topic, KB}<em>{\text{ID}) and notFindTopicInCS (Topic, CS}</em>{\text{ID}) and notFindOppTopicInCS (\text{not(Topic)}, \text{CS}<em>{\text{ID}) and addTopicToCS(Topic, CS}</em>{\text{ID})} ) or</td>
</tr>
<tr>
<td>add\text{ToRejectingList}(\text{RejectingList}, RejList, ID) and increase\text{Rejecting}(\text{NRejecting}, NRej) and increase\text{Reply}(\text{NReply}, NRep) and AccList= \text{AcceptingList} and NAcc is N\text{Accepting} \equiv \text{reject(Topic)} &lt;= a(\text{replyToProposalSender}<em>{\text{ID}}(\text{KB}</em>{\text{ID}}, \text{CS}<em>{\text{ID}}, \text{Topic}, \text{ID}</em>{\text{proposal}}), \text{ID}) )</td>
<td>reject(Topic) =&gt; a(\text{replyToProposalReceiver}<em>{\text{proposal}} (</em> , _ , _ , _ , _ , _ , _ , _ , _ , _ , <em>, ID</em>{\text{proposal}}) \equiv \text{findTopicInKB(Topic, KB}<em>{\text{Proposal}) and notFindTopicInCS(Topic, CS}</em>{\text{Proposal})} ) then</td>
</tr>
<tr>
<td>) )</td>
<td>a(\text{resultReceiver}<em>{\text{ID}}(\text{KB}</em>{\text{ID}}, \text{CS}<em>{\text{ID}}, \text{Topic}, \text{ID}</em>{\text{proposal}}, \text{ID}) .</td>
</tr>
</tbody>
</table>

**Figure B.10(b): Generated LCC Protocol for N-agent Dialogue**
### Bridging the Specification Protocol Gap in Argumentation

#### Appendix B: N-Agent Dialogue

<table>
<thead>
<tr>
<th>Proposal</th>
<th>Other agents</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a(resultSender_proposal(AgentList,NAgent, NSupporters,Topic,AcceptingList,RejectionList, AgentGroup), ID_proposal) ::=</code></td>
<td><code>a(resultReceiver_p(KB_p,CS_p,Topic,ID_Sender), P)</code></td>
</tr>
<tr>
<td><code>a(sendReachAgreement_proposal (AgentList,Topic),ID_proposal) &lt;= greaterThanOrEqual(NAccepting,NSupporters)</code></td>
<td><code>reachAgreement(Topic) &lt;= a(sendReachAgreement_proposal (AgentList, Topic ),IDProposal)</code></td>
</tr>
<tr>
<td><code>or</code></td>
<td><code>or</code></td>
</tr>
<tr>
<td><code>a(divideGroup_proposal (AgentList ,NAgent,NSupporters ,Topic,AcceptingList,RejectionList, [ ]) ,ID_proposal)</code></td>
<td><code>argueWith(Topic,P,O) &lt;= a(divideGroupSender(AgentList,NAgent,NSupporters,Topic,AcceptingList, RejectionList,AgentGroup),IDSender)</code></td>
</tr>
<tr>
<td><code>&lt; lessThan(NAccepting,NSupporters) and</code></td>
<td><code>then</code></td>
</tr>
<tr>
<td><code>isNotEmpty(RejectionList) and</code></td>
<td><code>a(startDID(KB_p,CS_p,Topic,ID_Sender, O),P)</code></td>
</tr>
<tr>
<td><code>isNotEmpty(AcceptingList)</code></td>
<td><code>) .</code></td>
</tr>
<tr>
<td><code>a(sendReachAgreementProposal(AgentList, Topic ),IDProposal) ::=</code></td>
<td><code>a(resultReceiver_o(KB_o,CS_o,Topic,ID_Sender), O) ::=</code></td>
</tr>
<tr>
<td><code>reachAgreement(Topic) =&gt; a(resultReceiverID(KBID,CSID,ID_proposal), ID)</code></td>
<td><code>reachAgreement(Topic) &lt;= a(sendReachAgreementProposal (AgentList, Topic ),IDProposal)</code></td>
</tr>
<tr>
<td><code>getAgentIDFromList(AgentList,otherAgents,ID)</code></td>
<td><code>or</code></td>
</tr>
<tr>
<td><code>then</code></td>
<td><code>( argueWith(Topic,O,P) &lt;= a(divideGroupSender(AgentList,NAgent,NSupporters,Topic,AcceptingList, RejectionList,AgentGroup),IDSender)</code></td>
</tr>
<tr>
<td><code>null &lt;= isAgentListEmpty(AgentList)</code></td>
<td><code>then</code></td>
</tr>
<tr>
<td><code>or</code></td>
<td><code>a(startDID(KB_o,CS_o,Topic,ID_Sender, P),O)</code></td>
</tr>
<tr>
<td><code>a(sendReachAgreement_proposal (OtherAgents, Topic, ID_proposal) ) .)</code></td>
<td><code>) .</code></td>
</tr>
<tr>
<td><code>a(divideGroupProposal(AgentList,NAgent, NSupporters,Topic,AcceptingList,RejectionList, AgentGroup), ID_proposal):=</code></td>
<td><code>a(recurs_proposal (AgentList,NAgent,NSupporters,0 ,Topic),ID_proposal)</code></td>
</tr>
<tr>
<td><code>{</code></td>
<td><code>&lt; isListEmpty(Re) or isListEmpty(Ac)</code></td>
</tr>
<tr>
<td><code>argueWith(Topic,O,P) =&gt;</code></td>
<td><code>or</code></td>
</tr>
<tr>
<td><code>a(resultReceiver_o(KB_o,CS_o,Topic,ID_proposal), O)</code></td>
<td><code>( argueWith(Topic,O,P) &lt;= a(divideGroupSender(AgentList,NAgent,NSupporters,Topic,AcceptingList, RejectionList,AgentGroup),IDSender)</code></td>
</tr>
<tr>
<td><code>)</code></td>
<td><code>then</code></td>
</tr>
<tr>
<td><code>then</code></td>
<td><code>a(startDID(KB_o,CS_o,Topic, ID_Sender, P),O)</code></td>
</tr>
<tr>
<td><code>{</code></td>
<td><code>)</code></td>
</tr>
<tr>
<td><code>a(recur_s_proposal (AgentList,NAgent,NSupporters,0 ,Topic),ID_proposal)</code></td>
<td><code>a(divideGroupProposal(AgentList ,NAgent, NSupporters,Topic,Ac,Re,AgentGroup ),IDProposal)) .</code></td>
</tr>
</tbody>
</table>

**Figure D.10(c): Generated LCC Protocol for N-agent Dialogue**
Bridging the Specification Protocol Gap in Argumentation

<table>
<thead>
<tr>
<th>Proposal</th>
<th>Other agents</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a(\text{recur}^{\text{Proposal}}(\text{AgentList}, \text{NAgent}, \text{NSupporters}, \text{replyN}, \text{Topic}), \text{ID}_P^{\text{Proposal}}) )</td>
<td>( a(\text{startDID}_P(\text{KB}_P, \text{CS}_P, \text{CS}_O, \text{ID}^{\text{Proposal}}_P, \text{ID}_P), \text{ID}_O) )</td>
</tr>
<tr>
<td>( := )</td>
<td>( := )</td>
</tr>
<tr>
<td>( { )</td>
<td>( )</td>
</tr>
<tr>
<td>( N = \text{replyN} + 1 \not\in \text{end(Topic)} )</td>
<td>( )</td>
</tr>
<tr>
<td>( \text{a(replyToClaimSender}_P(\text{KB}_P, \text{CS}_P, \text{CS}_O, \text{ID}^{\text{Proposal}}_P, \text{ID}_P), \text{ID}_O) )</td>
<td>( \text{a(claimSender} (\text{KB}_P, \text{CS}_P, \text{CS}_O, \text{ID}^{\text{Proposal}}_P, \text{ID}_P) )</td>
</tr>
<tr>
<td>( )</td>
<td>( \not\leftarrow )</td>
</tr>
<tr>
<td>( \text{or} )</td>
<td>( )</td>
</tr>
<tr>
<td>( N = \text{replyN} + 1 \not\in \text{end(Topic)} )</td>
<td>( )</td>
</tr>
<tr>
<td>( \text{a(replyToClaimReceiver}_P(\text{KB}_P, \text{CS}_P, \text{CS}_O, \text{ID}^{\text{Proposal}}_P, \text{ID}_P), \text{ID}_O) )</td>
<td>( \text{a(claimReceiver} (\text{KB}_P, \text{CS}_P, \text{CS}_O, \text{ID}^{\text{Proposal}}_P, \text{ID}_P) )</td>
</tr>
<tr>
<td>( )</td>
<td>( \not\leftarrow )</td>
</tr>
<tr>
<td>( \text{or} )</td>
<td>( )</td>
</tr>
<tr>
<td>( N = \text{replyN} + 1 \not\in \text{end(Topic)} )</td>
<td>( )</td>
</tr>
<tr>
<td>( \text{a(replyToWhySender}_P(\text{KB}_P, \text{CS}_P, \text{CS}_O, \text{ID}^{\text{Proposal}}_P, \text{ID}_P), \text{ID}_O) )</td>
<td>( \text{a(claimSender} (\text{KB}_P, \text{CS}_P, \text{CS}_O, \text{ID}^{\text{Proposal}}_P, \text{ID}_P) )</td>
</tr>
<tr>
<td>( )</td>
<td>( \not\leftarrow )</td>
</tr>
<tr>
<td>( \text{or} )</td>
<td>( )</td>
</tr>
<tr>
<td>( N = \text{replyN} + 1 \not\in \text{end(Topic)} )</td>
<td>( )</td>
</tr>
<tr>
<td>( \text{a(replyToArgumentSender}_P(\text{KB}_P, \text{CS}_P, \text{CS}_O, \text{ID}^{\text{Proposal}}_P, \text{ID}_P), \text{ID}_O) )</td>
<td>( \text{a(claimReceiver} (\text{KB}_P, \text{CS}_P, \text{CS}_O, \text{ID}^{\text{Proposal}}_P, \text{ID}_P) )</td>
</tr>
<tr>
<td>( )</td>
<td>( \not\leftarrow )</td>
</tr>
<tr>
<td>then</td>
<td>( )</td>
</tr>
<tr>
<td>( { )</td>
<td>( )</td>
</tr>
<tr>
<td>( \text{a(proposalSender}_P(\text{AgentList}, \text{NAgent}, \text{NSupporters}, \text{Topic}, ) ) )</td>
<td>( )</td>
</tr>
<tr>
<td>( \text{ID}^{\text{Proposal}}_P) } )</td>
<td>( \text{a(Proposal}_P(\text{AgentList}, \text{NAgent}, \text{NSupporters} ) )</td>
</tr>
<tr>
<td>( \not\leftarrow )</td>
<td>( )</td>
</tr>
<tr>
<td>( \text{isEqual}(N, \text{NAgent}) )</td>
<td>( \text{isEqual}(N, \text{NAgent}) )</td>
</tr>
<tr>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>or</td>
<td>( )</td>
</tr>
<tr>
<td>( )</td>
<td>( \text{a( recur}^{\text{Proposal}}(\text{AgentList}, \text{NAgent}, \text{NSupporters}, ) )</td>
</tr>
<tr>
<td>( )</td>
<td>( \text{N,Topic}, \text{ID}^{\text{Proposal}}_P) } )</td>
</tr>
</tbody>
</table>

**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*.
(10) Selects and Applies the *Recurs-To-N-Discourse 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 Location* (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 $claimSender_{P1}$, 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 $claimReceiver_{O1}$, 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 RSender1 with replyToClaimSenderO, 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 RReceiver1 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) 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 RSender2 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 RReceiver2 with replyToWhyReceiver_P, 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
Bridging the Specification Protocol Gap in Argumentation

*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_{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 \((KB_{P},CS_{P},CS_{O},T,KID_{O})\), and role id with \(ID_{P}\).

d) Moving to the left side conditions, the tool matches *C1* with \((\text{findTopicInKB}(T,KB_{O}) \text{ and notFindTopicInCS}(T,CS_{O}) \text{ and notFindOppTopicInCS}(\text{not}(T), CS_{O}) \text{ and addTopicToCS}(T,CS_{O}))\). 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

#### Appendix C

**Agent P**

<table>
<thead>
<tr>
<th>a(claimSenderP(KBP, CSP, CSO, T, IDP), IDP) ::=</th>
</tr>
</thead>
<tbody>
<tr>
<td>claim(T) =&gt; a(claimReceiverO(KB0, CS0, CSO), IDO)</td>
</tr>
<tr>
<td>addTopicToCS(T, CSP)</td>
</tr>
<tr>
<td>then</td>
</tr>
<tr>
<td>a(replyToClaimReceiverP(KBP, CSP, CSO, T, IDP), IDP).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a(replyToClaimReceiverP(KBP, CSP, CSO, T, IDP), IDP) ::=</th>
</tr>
</thead>
<tbody>
<tr>
<td>concede(T) &lt;= a(replyToClaimSenderP(KBP, CSP, CSO, T, IDP), IDP)</td>
</tr>
<tr>
<td>(notFindTopicInKB(T, KBP) and notFindTopicInCS(T, CSP) and addTopicToCS(T, CSP)).</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>why(T) &lt;= a(replyToClaimSenderP(KB0, CS0, CSP, T, IDP), IDP)</td>
</tr>
<tr>
<td>then</td>
</tr>
<tr>
<td>a(replyToWhySenderP(KBP, CSP, CSO, T, IDP), IDP).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a(replyToWhySenderP(KBP, CSP, CSO, T, IDP), IDP) ::=</th>
</tr>
</thead>
<tbody>
<tr>
<td>retract(T) =&gt; a(replyToWhyReceiverO(KB0, CS0, CSP, T, IDP), IDO)</td>
</tr>
<tr>
<td>(notFindPreInKB(T, KBP) and findTopicInCS(T, CSP) and subtractFromCS(T, CSP)).</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>(argue(Pre, T) =&gt; a(replyToWhyReceiverO(KB0, CS0, CSP, T, IDP), IDO)</td>
</tr>
<tr>
<td>(Pre = findPremise (T, KBP, CSP) and addPreToCS(T, Pre, CSP) )</td>
</tr>
<tr>
<td>then</td>
</tr>
<tr>
<td>a(replyToArgueReceiverP(KBP, CSP, CSO, T, IDP), IDP) ).</td>
</tr>
</tbody>
</table>

#### Agent O

<table>
<thead>
<tr>
<th>a(claimReceiverO(KB0, CS0, CSP, IDP), IDO) ::=</th>
</tr>
</thead>
<tbody>
<tr>
<td>claim(T) &lt;= a(claimSenderP(KBP, CSP, CSO, T, IDP), IDP)</td>
</tr>
<tr>
<td>then</td>
</tr>
<tr>
<td>a(replyToClaimSenderO(KB0, CS0, CSP, T, IDP), IDO).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a(replyToClaimSenderO(KB0, CS0, CSP, T, IDP), IDO) ::=</th>
</tr>
</thead>
<tbody>
<tr>
<td>concede(T) =&gt; a(replyToClaimReceiverO(KB0, CS0, CSP, T, IDP), IDO)</td>
</tr>
<tr>
<td>(notFindTopicInKB(T, KBO) and notFindTopicInCS(T, CSO) and addTopicToCS(T, CSO)).</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>why(T) =&gt; a(replyToClaimReceiverO(KB0, CS0, CSP, T, IDP), IDO)</td>
</tr>
<tr>
<td>then</td>
</tr>
<tr>
<td>a(replyToWhyReceiverO(KBO, CS0, CSP, T, IDP), IDO).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a(replyToWhyReceiverO(KBO, CS0, CSP, T, IDP), IDO) ::=</th>
</tr>
</thead>
<tbody>
<tr>
<td>retract(T) &lt;= a(replyToWhySenderO(KB0, CS0, CSP, T, IDP), IDP)</td>
</tr>
<tr>
<td>(notFindPreInKB(T, KBO) and findTopicInCS(T, CSP) and subtractFromCS(T, CSP)).</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>(argue(Pre, T) &lt;= a(replyToWhySenderO(KB0, CS0, CSP, T, IDP), IDP)</td>
</tr>
<tr>
<td>(Pre = findPremise (T, KBP, CSP) and addPreToCS(T, Pre, CSP) )</td>
</tr>
<tr>
<td>then</td>
</tr>
<tr>
<td>a(replyToArgueSenderO(KB0, CS0, CSP, T, Pre, IDP), IDO) ).</td>
</tr>
</tbody>
</table>

---

**Figure C.1(a): Generated LCC Protocol for Persuasion Dialogue (Part 1)**

**Appendix C: Persuasion Dialogue**

319
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)
Bridging the Specification Protocol Gap in Argumentation

Appendix C: Persuasion Dialogue

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, DivideC2, AgentGroupC3, and RecursC4 conditions (See chapter 5 for more detail).

- TerminationC1 = greaterThanOrEqual(NAccepting,NSupporters)
- DivideC2 = lessThan(NAccepting,NSupporters) and
  isNotEmpty(RejectionList) and isNotEmpty(AcceptingList)
- AgentGroupC3 = creatOneAgentGroup
  (RejectingList,Re,AcceptingList,Ac, AgentGroup, AGroup,P,O)
- RecursC4 = 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 RSender1 with claimSenderP1.
  b) Moving to the right side of the locution icon, the tool matches RReceiver1 with claimReceiverP1.
The tool matches $C1$ 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 $C1$ 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 ($KB_O,CS_O,CS_P,T,ID_{Proposal},ID_O$, $ID_O$), 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):
Bridging the Specification Protocol Gap in Argumentation

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 $replyToArgueSender_{O}$, 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 $O$.
Bridging the Specification Protocol Gap in Argumentation

Appendix C: Persuasion Dialogue

Figure C.8(a): Generated LCC Protocol for Persuasion Dialogue (Part 1)

<table>
<thead>
<tr>
<th>Proposal</th>
<th>Other Agents</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a(proposalSender_{proposal}(AgentList,NAgent,NSupporters,Topic),ID_{proposal}) ::= )</td>
<td>( a(proposalReceiver_{id}(KB_{id},CS_{id},ID_{proposal}),ID) ::= )</td>
</tr>
<tr>
<td>( proposal(Topic) =&gt; a(proposalReceiver_{id}(KB_{id},CS_{id},ID_{proposal}), ID) )</td>
<td>( proposal(Topic)&lt;= a(proposalSender_{proposal}(AgentList,NAgent,NSupporters,Topic),ID_{proposal}) )</td>
</tr>
<tr>
<td>( \leftarrow getAgentIDFromList(AgentList,otherAgents,ID) ) and ( addTopicToCS(Topic,CS_{proposal}) ) )</td>
<td>then</td>
</tr>
<tr>
<td>( )</td>
<td>( a(replyToProposalSender(KB_{id},CS_{id},Topic,ID_{proposal}), ID). )</td>
</tr>
</tbody>
</table>

Figures C.8(a): Generated LCC Protocol for Persuasion Dialogue (Part 1)
<table>
<thead>
<tr>
<th>Proposal</th>
<th>Other agents</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(resultSender_\text{proposal}(AgentList,NAgent,NSupporters,Topic,AcceptingList,RejectionList,AgentGroup), ID_{proposal}) ::=</td>
<td>a(resultReceiver_\text{P}(KB_P,CS_P,CS_O,Topic, ID_{proposal}),P) ::=</td>
</tr>
<tr>
<td>a(sendReachAgreement_\text{proposal}(AgentList,NAgent,Topic,ID_{proposal}) \leftarrow greaterThanOrEqual(NAccepting, NSupporters) or</td>
<td>reachAgreement(Topic) \leftarrow</td>
</tr>
<tr>
<td>a(divideGroup_\text{proposal}(AgentList,NAgent,NSupporters,Topic,AcceptingList,RejectionList,AgentGroup), ID_{proposal}) \leftarrow ( lessThan(NAccepting,NSupporters) AND isNotEmpty(RejectionList) AND isNotEmpty(AcceptingList) ).</td>
<td>a(sendReachAgreement_\text{Proposal}(AgentList, Topic ),ID_{Proposal}) or</td>
</tr>
<tr>
<td>a(sendReachAgreement_\text{Proposal}(AgentList,Topic,ID_{Proposal}) ::=</td>
<td>(</td>
</tr>
<tr>
<td>reachAgreement(Topic) \leftarrow a(resultReceiver_\text{P}(KB_P,CS_P,ID_{proposal}), ID) \leftarrow getAgentIDFromList (AgentList,otherAgents,ID) then ( null \leftarrow isAgentListEmpty(AgentList) or</td>
<td>then</td>
</tr>
<tr>
<td>a(divideGroup_\text{Proposal}(AgentList,NAgent,NSupporters,Topic,AcceptingList,RejectionList,AgentGroup), ID_{proposal}) ::=</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>arguWith(Topic,P,O) \leftarrow a(resultReceiver_\text{O}(KB_O,CS_O,CS_P,Topic,ID_{proposal}), O) ) then</td>
</tr>
<tr>
<td>a(recursive_\text{Proposal}(AgentList,NAgent,NSupporters,Topic,0,Topic,ID_{proposal}) \leftarrow isListEmpty(Re) or isListEmpty(Ac) or</td>
<td></td>
</tr>
<tr>
<td>a(divideGroup_\text{Proposal}(AgentList,NAgent,NSupporters,Topic,Ac,Re,AGroup), ID_{Proposal}) )).</td>
<td></td>
</tr>
</tbody>
</table>

Figure C.8(b): Generated LCC Protocol for Persuasion Dialogue (Part 2)
<table>
<thead>
<tr>
<th>Proposal</th>
<th>Other agents</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(recurs&lt;sub&gt;Proposal&lt;/sub&gt; (AgentList, NAgent, NSupporters, replyN, Topic), ID&lt;sub&gt;Proposal&lt;/sub&gt;) ::=</td>
<td></td>
</tr>
<tr>
<td>( N = replyN +1 \leftarrow end(Topic) \iff a(replyToClaimSender&lt;sub&gt;0&lt;/sub&gt; (KB&lt;sub&gt;0&lt;/sub&gt;, CS&lt;sub&gt;0&lt;/sub&gt;, CS&lt;sub&gt;1&lt;/sub&gt;, T, ID&lt;sub&gt;Proposal&lt;/sub&gt;, ID&lt;sub&gt;P&lt;/sub&gt;), ID&lt;sub&gt;0&lt;/sub&gt;) or</td>
<td></td>
</tr>
<tr>
<td>N = replyN +1 \leftarrow end(Topic) \iff a(replyToClaimReceiver&lt;sub&gt;1&lt;/sub&gt; (KB&lt;sub&gt;1&lt;/sub&gt;, CS&lt;sub&gt;1&lt;/sub&gt;, CS&lt;sub&gt;2&lt;/sub&gt;, T, ID&lt;sub&gt;Proposal&lt;/sub&gt;, ID&lt;sub&gt;P&lt;/sub&gt;), ID&lt;sub&gt;1&lt;/sub&gt;) or</td>
<td></td>
</tr>
<tr>
<td>N = replyN +1 \leftarrow end(Topic) \iff (a(replyToWhySender&lt;sub&gt;3&lt;/sub&gt; (KB&lt;sub&gt;2&lt;/sub&gt;, CS&lt;sub&gt;2&lt;/sub&gt;, CS&lt;sub&gt;3&lt;/sub&gt;, T, ID&lt;sub&gt;Proposal&lt;/sub&gt;, ID&lt;sub&gt;0&lt;/sub&gt;), ID&lt;sub&gt;0&lt;/sub&gt;) or</td>
<td></td>
</tr>
<tr>
<td>N = replyN +1 \leftarrow end(Topic) \iff (a(replyToWhyReceiver&lt;sub&gt;4&lt;/sub&gt; (KB&lt;sub&gt;4&lt;/sub&gt;, CS&lt;sub&gt;4&lt;/sub&gt;, CS&lt;sub&gt;5&lt;/sub&gt;, T, ID&lt;sub&gt;Proposal&lt;/sub&gt;, ID&lt;sub&gt;1&lt;/sub&gt;), ID&lt;sub&gt;1&lt;/sub&gt;) or</td>
<td></td>
</tr>
<tr>
<td>N = replyN +1 \leftarrow end(Topic) \iff a(replyToArgueSender&lt;sub&gt;5&lt;/sub&gt; (KB&lt;sub&gt;5&lt;/sub&gt;, CS&lt;sub&gt;5&lt;/sub&gt;, CS&lt;sub&gt;6&lt;/sub&gt;, T, Pre&lt;sub&gt;DID&lt;/sub&gt;, ID&lt;sub&gt;Proposal&lt;/sub&gt;, ID&lt;sub&gt;0&lt;/sub&gt;), ID&lt;sub&gt;0&lt;/sub&gt;) or</td>
<td></td>
</tr>
<tr>
<td>N = replyN +1 \leftarrow end(Topic) \iff a(replyToArgueReceiver&lt;sub&gt;6&lt;/sub&gt; (KB&lt;sub&gt;6&lt;/sub&gt;, CS&lt;sub&gt;6&lt;/sub&gt;, CS&lt;sub&gt;7&lt;/sub&gt;, T, ID&lt;sub&gt;Proposal&lt;/sub&gt;, ID&lt;sub&gt;P&lt;/sub&gt;), ID&lt;sub&gt;1&lt;/sub&gt;) or</td>
<td></td>
</tr>
<tr>
<td>then</td>
<td></td>
</tr>
<tr>
<td>( a(proposalSender&lt;sub&gt;7&lt;/sub&gt; (AgentList, NAgent, NSupporters, Topic), ID&lt;sub&gt;Proposal&lt;/sub&gt;) \leftarrow isEqual(N, NAgent) or</td>
<td></td>
</tr>
<tr>
<td>a( recurs&lt;sub&gt;Proposal&lt;/sub&gt; (AgentList, NAgent, NSupporters, N, Topic), ID&lt;sub&gt;Proposal&lt;/sub&gt; ) ).</td>
<td></td>
</tr>
</tbody>
</table>

Figure C.8(c): Generated LCC Protocol for Persuasion Dialogue (Part 3)
<table>
<thead>
<tr>
<th>Agent P</th>
<th>Agent O</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a(claimSender_P(KB_P,CS_P,CS_O,T,ID_O),ID_P)</code> ::=</td>
<td></td>
</tr>
<tr>
<td><code>claim(T)</code> =&gt;</td>
<td></td>
</tr>
<tr>
<td><code>a(claimReceiver_O(KB_O,CS_O,CS_P,T,ID_P),ID_O)</code></td>
<td></td>
</tr>
<tr>
<td><code>addTopicToCS(T,CS_P)</code></td>
<td></td>
</tr>
<tr>
<td>then</td>
<td></td>
</tr>
<tr>
<td><code>a(replyToClaimReceiver_O(KB_P,CS_P,CS_O,T,ID_O),ID_P)</code></td>
<td></td>
</tr>
<tr>
<td><code>a(claimReceiver_O(KB_O,CS_O,CS_P,T,ID_P),ID_O)</code> ::=</td>
<td></td>
</tr>
<tr>
<td><code>claim(T)</code></td>
<td></td>
</tr>
<tr>
<td><code>a(claimSender_P(KB_P,CS_P,CS_O,T,ID_O),ID_P)</code></td>
<td></td>
</tr>
<tr>
<td>then</td>
<td></td>
</tr>
<tr>
<td><code>a(replyToClaimSender_O(KB_O,CS_O,CS_P,T,ID_P),ID_O)</code></td>
<td></td>
</tr>
</tbody>
</table>
| **Figure C.8(d): Generated LCC Protocol for Persuasion Dialogue (Part 4)**
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)).
<table>
<thead>
<tr>
<th>Agent P</th>
<th>Agent O</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(concede(T) &lt;= a(replyToArgueSender_o(KB_o, CS_o, CS_p, T, Pre, ID_p), ID_o)) then</code></td>
<td><code>(concede(T) =&gt; a(replyToArgueReceiver_p(KB_p, CS_p, CS_o, T, Pre, ID_p), ID_o))</code></td>
</tr>
<tr>
<td><code>end(Topic) =&gt; a(recurse_Proposal(AgentList, NAgent, NSupporters, NReply, Topic), ID_Proposal)</code></td>
<td><code>end(Topic) =&gt; a(recurse_Proposal(AgentList, NAgent, NSupporters, NReply, Topic), ID_Proposal)</code></td>
</tr>
<tr>
<td><code>a(proposalReceiver_ID(KB_ID, CS_ID, ID_Proposal), ID)</code></td>
<td><code>a(proposalReceiver_ID(KB_ID, CS_ID, ID_Proposal), ID)</code></td>
</tr>
<tr>
<td><code>or</code></td>
<td><code>or</code></td>
</tr>
<tr>
<td><code>(argue(Def, T') &lt;= a(replyToArgueSender_p(KB_p, CS_p, CS_o, T, Pre, ID_o), ID_o))</code></td>
<td><code>(Def = findDefeats(T, Pre, KB_o, CS_o) and addDefeatToCS(Def, CS_o))</code></td>
</tr>
<tr>
<td><code>then</code></td>
<td><code>then</code></td>
</tr>
<tr>
<td><code>a(replyToArgueReceiver_p(KB_p, CS_p, CS_o, T, Pre, Def, ID_o), ID_o)</code></td>
<td><code>a(replyToArgueReceiver_o(KB_o, CS_o, CS_p, T, Pre, ID_p), ID_o)</code></td>
</tr>
<tr>
<td><code>or</code></td>
<td><code>or</code></td>
</tr>
<tr>
<td><code>(why(Pre) &lt;= a(replyToArgueSender_o(KB_o, CS_o, CS_p, T, Pre, ID_p), ID_o))</code></td>
<td><code>(why(Pre) =&gt; a(replyToArgueReceiver_o(KB_o, CS_o, CS_p, T, Pre, ID_p), ID_o))</code></td>
</tr>
<tr>
<td><code>then</code></td>
<td><code>then</code></td>
</tr>
<tr>
<td><code>a(replyToWhySender_o(KB_o, CS_o, CS_p, T, Pre, ID_p), ID_p)</code></td>
<td><code>a(replyToWhyReceiver_o(KB_o, CS_o, CS_p, T, Pre, ID_p), ID_o)</code></td>
</tr>
</tbody>
</table>

**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-Discourse 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)
Bridging the Specification Protocol Gap in Argumentation

Appendix C: Persuasion Dialogue

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\textsubscript{P} CPN Subpage

- Figure C.10 shows the claimSender\textsubscript{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\textsubscript{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\textsubscript{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.11: The claimReceiver_o CPN Subpage

Figure C.12: The replyToClaimSender_o CPN Subpage

- Figure C.12 shows the replyToClaimSender_o role CPN subpage. This subpage sends two messages: (1) sends why message using why3 output place and changes its role to ReplyToWhyReceiver using ChangeRoleToWhyReceiver output place; (2) sends concede message using concede2 output place and then ends the dialogue using end output place.

- Figure C.13 shows the replyToClaimReceiver_p role CPN subpage. This subpage receives two messages (why or concede) and generates responses depending on some conditions. If it receives the concede message using concede2 input place, it responses by ending the dialogue using end output place. Otherwise, if it receives the why message using why3 input place, it...
responses by changing its role to \textit{ReplyToWhySender} using \textit{ChangeRoleToWhySender}.

- Figure C.14 shows the \textit{replyToWhySender} role CPN subpage. This subpage sends two messages: (1) sends \textit{argue} message using \textit{argue5} output place and changes its role to \textit{ReplyToArgueReceiver} using \textit{ChangeRoleToArgueReceiver}
output place; (2) sends \textit{retract} message using \textit{retract4} output place and then ends the dialogue using \textit{end} output place.

- Figure C.15 shows the \textit{replyToWhyReceiver} role CPN subpage. This subpage receives two messages (\textit{argue} or \textit{retract}) and generates responses depending on some conditions. If it receives the \textit{retract} message using \textit{retract4} input place, it responses by ending the dialogue using \textit{end} output place. Otherwise, if it receives the \textit{argue} message using \textit{argue5} input place, it responses by changing its role to \textit{ReplyToArgueSender} using \textit{ChangeRoleToArgueSender}.
Bridging the Specification Protocol Gap in Argumentation

Appendix C: Persuasion Dialogue

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 concede\_6 output place and then ends the dialogue using end output place; (2) sends why message using why\_7 output place and changes its role to ReplyToWhyReceiver using ChangeRoleToWhyReceiver output place; (3) sends argue message using argue\_8 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 concede\_6 input place, it responses by ending the dialogue using end output place. If it receives the why message using why\_7 input place, it responses by changing its role to ReplyToWhySender using ChangeRoleToWhySender. If it receives argue message using argue\_8 input place, it responses 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,
Appendix C: Persuasion Dialogue

Figure C.18: The protocol CPN Superpage
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:**

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.
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:**
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

Appendix C: Persuasion Dialogue

Figure C.24: Recursive Message Property Page

Figure C.25: The Verification Result of the Five Basic Properties
This appendix presents basic CPN functions code, where:

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

rmall = removes an item from the list

**CPN Functions**

(14) Add an argument 't' to a commitment store list 'sCSL':

```haskell
fun addTopicToCS(sCSL, t) = ins_new sCSL t;
```

(15) Add a premise of an argument 't' to a commitment store list 'sCSL':

```haskell
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':

```haskell
fun addDefeatToCS(sCSL, def) = ins_new sCSL def;
```

---

(17) Subtract an argument 't' from a commitment store list 'sCSL':

\[
\text{fun subtractFromCS(sCSL,t) = rmall t sCSL;}
\]

(18) Find an argument 't' in a commitment store list 'sCSL':

\[
\text{fun findTopicInCS(sCSL,t) = mem sCSL t;}
\]

(19) Find a premise of an argument 't' in a commitment store list 'sCSL':

\[
\text{fun findPreInCS(sCSL,t) = mem sCSL t;}
\]

(20) Find an argument in a knowledge base list 'KBlist' where 'f' represents a fact and 'pre' represents a premise:

\[
\text{fun findTopicInKB((f,pre)::KBlist,t)=}
\]

\[
\begin{align*}
& \text{if } ((f = t)) \text{ then } \text{true} \\
& \text{else if } (\text{length } \text{KBlist} \geq 1) \text{ then } \text{findTopicInKB(KBlist,t)} \\
& \text{else false;}
\end{align*}
\]

(21) Find a premise of an argument in a knowledge base list 'KBlist' where 'f' represents a fact and 'pre' represents a premise:

\[
\text{fun findPreInKB((f,pre)::KBlist,t)=}
\]

\[
\begin{align*}
& \text{if } (f=t) \text{ then } \text{true} \\
& \text{else if } (\text{length } \text{KBlist} \geq 1) \text{ then } \text{findPreInKB(KBlist,t)} \\
& \text{else false;}
\end{align*}
\]

(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':

Appendix D: CPN Functions
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

Appendix E: GenerateLCCProtocol Tool Graphical User Interface
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.
Bridging the Specification Protocol Gap in Argumentation

Appendix E: GenerateLCCProtocol Tool Graphical User Interface
(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;

   ![File menu bar](image)

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).

   ![Dialogue Interaction Diagram menu bar](image)

3. **LCC menu bar:** this menu has three buttons:

   ![LCC menu bar](image)
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:
Bridging the Specification Protocol Gap in Argumentation

a) Agents KB: used to get the agents Knolwidge 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:
Appendix E: GenerateLCCProtocol Tool Graphical User Interface

Figure E.7 (a): How to Read DID
Meaning of color tab

Locution Types tab

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 conditions;

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 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).

<table>
<thead>
<tr>
<th>1. Level number:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. <strong>Locution</strong>[Locution Type,Locution, Structural rules],</td>
</tr>
<tr>
<td>3. <strong>Sender-Information</strong>[Role Name,Role arguments,Agent ID,Conditions],</td>
</tr>
<tr>
<td>4. <strong>Receiver-Information</strong>[Role Name,Role arguments,Agent ID,Conditions],</td>
</tr>
</tbody>
</table>

Figure E.10: DID Textual Representation

**DID Textual Representation**

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 $I$ (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$_P$;

Appendix E: GenerateLCCProtocol Tool Graphical User Interface
Bridging the Specification Protocol Gap in Argumentation

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td><strong>Locution</strong>[Starting,claim(T),null],</td>
</tr>
<tr>
<td>3.</td>
<td><strong>Sender-Information</strong>[claimSender_{p}(KB_{p}, CS_{p}, CS_{o}, T,ID_{o}),ID_{p},addTopicToCS(T,CS_{p})],</td>
</tr>
<tr>
<td>4.</td>
<td><strong>Receiver-Information</strong>[claimReceiver_{o}(KB_{o}, CS_{o}, CS_{p}, ID_{p},ID_{o})].</td>
</tr>
</tbody>
</table>

**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 Representation 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).
Appendix E: Generatelccprotocol Tool Graphical User Interface

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):

Appendix E: GenerateLCCProtocol Tool Graphical User Interface
Figure E.14: Generate a Concrete LCC Protocol for the Persuasion Dialogue

Appendix E: GenerateLCCProtocol Tool Graphical User Interface
Bridging the Specification Protocol Gap in Argumentation

Appendix E: GenerateLCCProtocol Tool Graphical User Interface

Figure E.15: Generate a Concrete LCC Protocol for the Persuasion Dialogue among N-agent

1- Select DID among two agents
2- Select the divide group condition
3- Click on Generate LCC Protocol
(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:

   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

Bibliography


Bridging the Specification Protocol Gap in Argumentation


[Bowles et al., 1994] BOWLES, ANDREW, ROBERTSON, DAVE, VASCONCELOS, WAMBERTO, VARGAS-VERA, MARIA, and BENTAL,


Bibliography 378
Bridging the Specification Protocol Gap in Argumentation


[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*
Bridging the Specification Protocol Gap in Argumentation


Bibliography

381
Bridging the Specification Protocol Gap in Argumentation


[McBurney et.al., 2002] MCBURNEY, PETER, PARSONS, SIMON and WOOLDRIDGE, MICHAEL. Desiderata for agent argumentation protocols. In Bibliography 382


Bibliography 385


[Shin et. al., 2005] SHIN, MICHAEL, ALEXANDER, LEVIS and LEE, WAGENHALS. Analyzing Dynamic Behavior of Large-Scale Systems through
Bridging the Specification Protocol Gap in Argumentation


Bibliography 387


