Automatic Agent Protocol Generation from Argumentation

Ashwag Omar Maghraby

School of Informatics
The University of Edinburgh
Edinburgh EH8 9LE, UK

A.O.Maghraby@sms.ed.ac.uk

ABSTRACT

Research on argumentation has concentrated on abstract specification of arguments between a protagonist and an antagonist. However, as we build complete multi-agent systems that involve argumentation, there is a need to produce concrete implementations in which these abstract specifications are realised via protocols coordinating agent behavior. This creates a gap between argument specification and implementation which we bridge using a combination of transformational synthesis and model checking. The resulting system provides engineers with a means of moving rapidly from argument specification to implementation, using the Argument Interchange Format as the specification language and the Lightweight Coordination Calculus as an implementation language.

Categories and Subject Descriptors

I.2.2 [**Automatic Programming**]: Program synthesis, Program transformation, and Program verification; D.2.4 [**Software Engineering**] Software/Program Verification-Model checking

General Terms

Languages, Design, Verification

Keywords

Transformational Synthesis, Interaction models, Argumentation, Dialogue Game, Verification, Model Checking

1. INTRODUCTION

Today, argumentation [1] is gaining more prominence since it is being used as part of the high level specification of multi-agent systems (MAS). However, 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 Argument Interchange Format (AIF) [2,3], but it is here we present an approach for tackling the second problem.

To solve the first problem, the argumentation community has developed AIF [2,3], 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, concerned with only 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.

Cite as: Bridging the Specification-Protocol Gap in Argumentation, Ashwag Omar Maghraby, *Proc. of* 13th European Agent Systems Summer School (EASSS 2011), July 11-15 2011. Copyright © 2011, European Agent Systems Summer School (http://eia.udg.edu/easss2011/). All rights reserved.

This research describes an approach which bridges the gap between argument specification and multi-agent implementation using a combination of transformational synthesis and model checking. The resulting system provides engineers with a means of moving rapidly from argument specification to implementation in a peer-to-peer setting. It uses AIF as an example of an argumentation language and Lightweight Coordination Calculus (LCC) [4,5], an executable specification language, as a multi-agent implementation language.

2. PROBLEM DESCRIPTIONS AND MOTIVATION

AIF is an effective argument structure language. It enables users to structure arguments using diagramatic linkage of natural language sentences. However, AIF is not an executable specification language. It specifies the properties that define an argument without prescribing how that argument may be made operational. An example of this problem is shown in Figure 1 [2,3]. 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":

1. There are two arguments: one for ~flies(P) and one for flies(P);
2. The argument for ~flies(P) is composed of one Rule of inference Application node (RA-node that define the support or inference of argument), namely Modus Ponens and two child nodes (premises);
3. The argument for flies(P) is composed of one RA-node, namely defeasible Modus Ponens and two child nodes (premises);
4. The argument for ~flies(P) has a higher degree of support because the premises support it with a higher degree of probability (1 degree). Conversely, the argument for flies(P) is weak because the premises support it with only 0.8 degree (a low probability). So ~flies(P) is preferred to the argument for flies(P). That is why the intermediate Preference Application node (PA-node that defines the value judgments or preference orderings of argument), namely Logical attack, linking ~flies(P) to flies(P).

In particular, this example demonstrates that a persuasion dialogue can be specified abstractly by using arguments expressed in AIF. However, this is a long way from the machinery required to build a concrete discussion system since the AIF is used to represent data not to process data (not to represent argumentation protocol).

Chesnevar *et al*. [2] and Willmott *et al*. [3] suggest a way to solve this problem by identifying two elements: (1) Locutions, which are particular words, phrases or form of expressions which are used by agents, and (2) Interaction Protocols, which define

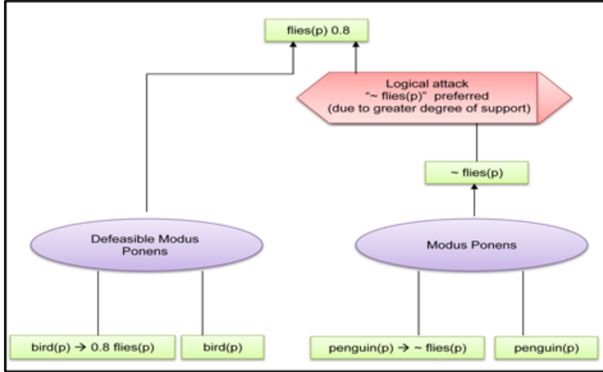

Figure 1: Specification in AIF of the Arguments Exchanged by Agents Discussing the Flying Abilities of the "P" Bird

communication between agents via a set of rules governing how two or more agents should interact in order to reach a specific goal. However, these papers only give some suggestions for solving the AIF implementation problem. The only study which has attempted to solve the AIF problem is in [6]. This study extends AIF to represent argumentation based dialogues. It also represents interaction protocols using LCC. 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 several ways. Firstly, it is limited to dialogues between two agents. Secondly, it does not explain how to (semi)automatically synthesise protocols for the given argumentation, it only gives an example of argumentation in LCC.

In this paper, I will present another approach (which is a continuation of Modgil and McGinnis [6], Chesnevar *et al*. [2] and Willmott *et al*. [3]) to solve this problem. As shown in Figure 2, our approach consists of two parts. Section 3 provides an overview of 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) definition of a new high level control flow specification language for multi-agent protocols called Dialogue Interaction Diagram (DID), which is allowed to specify the argument protocol in an abstract way by extending the AIF; (2) implementation of a tool which automatically synthesises concrete LCC protocols from DID specifications using a new pattern-based synthesis method.

Part two consists on proposing a verification methodology based on model checking to check the semantics of the DID specification used as a starting point against the semantics of the synthesised LCC protocol. In Section 4 we explain the model checker, which was built in three stages: (1) mapping the LCC specification in equivalent Coloured Petri Nets (CPNs); (2) generating Standard ML (Meta-Language) specifications [7,8] the key properties inferred from the DID specification (3) verifying the properties defined in (2) over the state space graph generated from the CPN obtained in (1).

3. TRANSFORMATIONAL SYNTHESIS

This section describes how our approach bridges the gap between an argument specification language and an agent's implementation language by: (1) Defining DID as a high level argument specification language for multi-agent protocols; (2) Synthesising concrete LCC protocols from DID specifications.

3.1 Argument Specification Language

As mentioned in the previous sections**,** the AIF language is intuitive but too abstract, while multi-agent protocol languages such as LCC are concrete but it contain too much detail which are for everyday use by argumentation protocol designers and difficult to learn and this means that we need a language in the middle between AIF and LCC. Also, specifying argumentation protocols using programming-level protocol languages is error-prone, and a higher-level graphical language can help avoid low-level errors that can occur and that means we need a high level language. In this section, we propose a new high-level specification language for multi-agent protocols called a 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 used to specify the dialogue game protocol in an abstract way. It provides mechanisms to represent interaction protocol rules between $N \geq 2$ 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 only 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) and immediate-reply moves (the turn-taking between agents switches after each move - moving from a level to the next level- and each agent must reply to the move of the previous agent). These assumptions are commonly made in argument specification languages like the AIF and they help to keep the specifications in these languages compact.

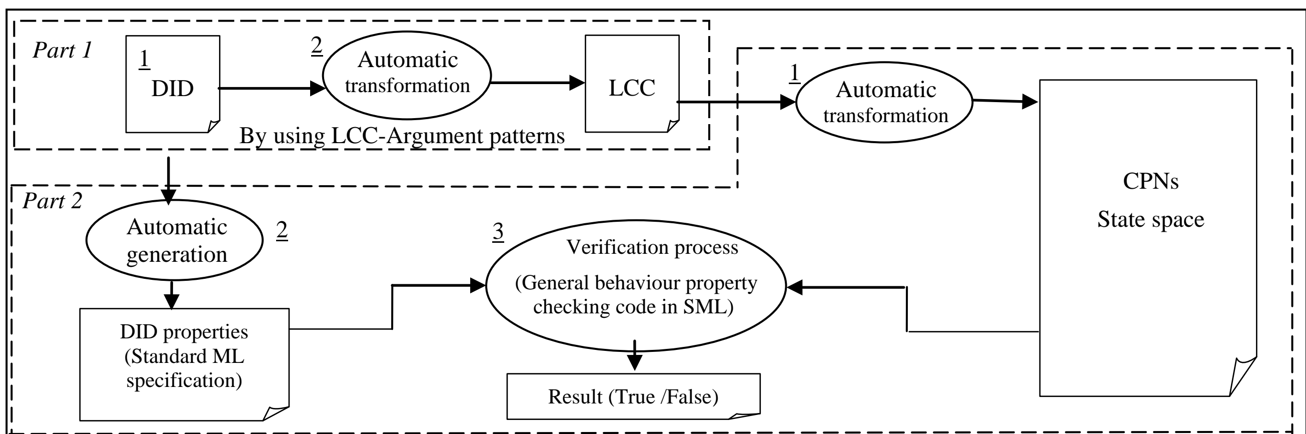

Figure 2: Overall structure of this research

3.1.1 DID Formal Definition

Our formal definition of the DID is based on the standard terminology considered for the specification of protocols in dialogue games (Dialogue games are interactions between two or more agents, where each agent makes a move by speaking in a common communication language and according to combination rules) [8] :

1. Locutions: represent the set of permitted moves. As in [6] the parameters of the locutions are arguments expressed in AIF, what makes the DID an extension of AIF;
2. One Commitment Store (*CS*) for each participant. The CSs of the participants reflect the state of the dialogue;
3. Commitment rules (post-conditions): define the propositional commitments made by each participant with each move during the dialogue;
4. Structural rules (reply rules or dialogue rules): define legal moves in terms of the available moves that a participant can select to follow on the previous move;
5. Turn Taking (Next player).

In order to represent argument protocol in full, more concepts are required, therefore, we have added to DID the following concepts:

1. Precondition rules define the pre-conditions under which the move will be achieved;
2. Location types (Act types);
3. Sender and receiver agent's roles.

The most notable differences between DID and existing languages for argumentation-based interaction protocols are:

1. DID arguments are expressed in AIF. Others have assumed specific argument formats which are dependent on the type of dialogue or the particular context of application considered;
2. DID allows the specification of dialogues between $N \geq 2$ agents, while existing works mainly focus on 2 agent dialogues. For an extensive review of the state of the art in the field of argumentation-based dialogues in MAS we refer the reader to [1,11].
3. DID is easier to use because it is a high-level graphical language and people in agent community are familiar with high level language or graphical notation language like Agent UML [12].

3.1.2 DID Elements

The basic element of every DID is a locution which is represented as an icon. A locution icon (as shown in Figure 3) is simply a rectangle divided into three sections. The topmost section contains the name of the locution. 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). A rhombus shape represents conditions which apply to each move; when connected to the left hand section it represents sender conditions and when connected to the right hand section it represents receiver conditions. Dotted rectangles represent the locution type: Starting (can be used to open a dialogue), Termination (can be used to terminate the dialogue), and Recursive locution (can be used to remain in the dialogue).

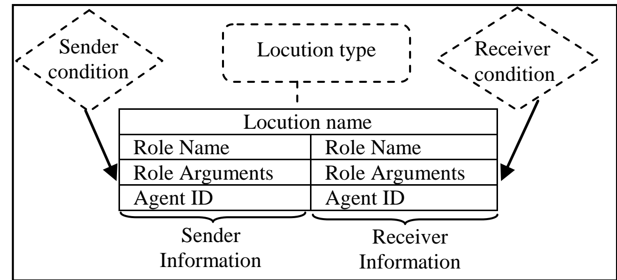

Figure 3: Locution icon

3.1.3 DID Example

Figure 4 shows a partial DID structure of a persuasion dialogue. A DID is created by linking the locution icons together. The links between locution icons represent reply relations between arguments. In Figure 4, there are three locutions: two attack locutions which has a reply move (*claim*, and *why*), and one surrender locution which does not have a reply move (*concede*). There are three types of locution: starting (*claim*), termination (*concede*), and recursive (*why*) locution.

In this example, a dialogue always starts with a *claim* and ends with a *concede* locution. A rhombus shape represents conditions which applies to each move. The variable *KB* (knowledge base list) represents the agent's private knowledge represented 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, *Agent1* can open the discussion by sending a *claim*(Ʊ) locution if he is able to satisfy *AddToCS*(Ʊ,CS_P)

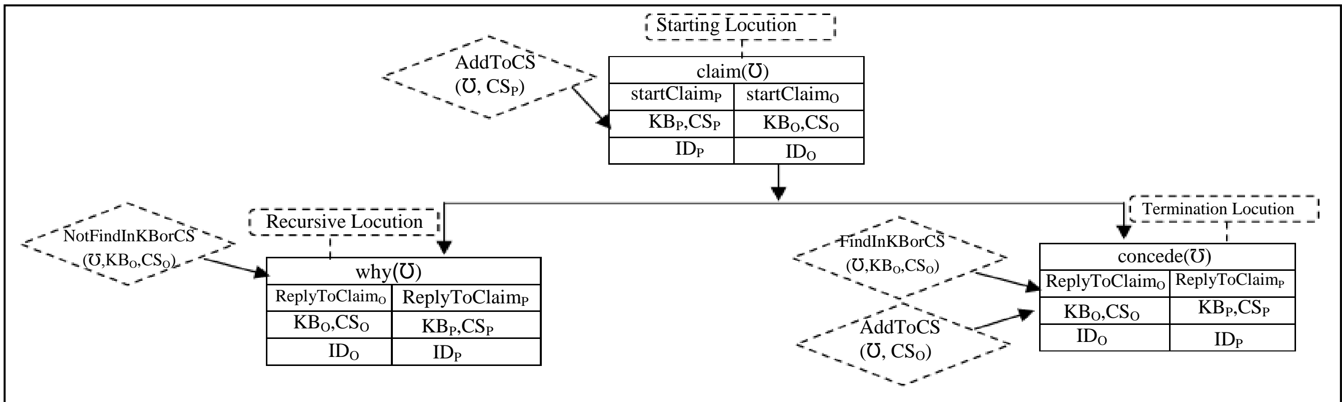

Figure 4: Partial DID for a Persuasion Dialogue

condition (*AddToCS*(℧,CS_P)) is used to update agent commitment store CS_P by adding ℧ to it). Then, turn-taking switches to *Agent2*. *Agent2* has to choose between two different possible reply locutions: *why*(℧) or *concede*(℧). *Agent2* will make his choice using the conditions which appear in the rhombus shape. In order to choose *concede*(℧), *Agent2* must be able to satisfy the two conditions which connect with concede: Condition 1: *FindInKBorCS*(℧, KB_O,CS_O) which is used to check whether ℧ is acceptable in the agent argumentation system KB_O and CS_O or not. If ℧ is acceptable, this constraint returns true; Condition 2: *AddToCS*(℧, CS_O) $Agent_O$ will use this constraint to update its commitment store by adding ℧ to CS_O. If *Agent2* is not able to satisfy these conditions, *Agent2* will send *why*(℧). After that, the turn switches to *Agent1*, and so forth.

Although; this example is for 2 agents, DID can handle more than 2 agents interactions protocol but at the cost of more complex graphical notation.

3.2 Implementation Language

To support formal analysis and verification, the AIF [2,3] community suggests using a process and declarative language to implement the dialogue games protocol. For this reason we choose LCC, a declarative, process calculus-based, executable specification language used for specifying the message-passing behavior of MAS interaction protocols.

The abstract syntax of a LCC clause [4,13] is shown in Figure 5.

```
Framework := {Clause,….}
Clause := Agent :: Dn
Agent := a(Role, Id)
Dn := Agent | Message |  null ← C | Dn then Dn | Dn or Dn
Message := M ==> Agent | M ==> Agent ←C | M <== Agent
              |  C ←M <== Agent
C := Term | C and C | C or C
Role := Term
M := Term
```

Figure 5: The abstract Syntax of LCC

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. A 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 role, exchange (receive or send) messages and exit the dialogue under a certain constraint *C* (null ← C). A constraint is defined as a propositional formula defined over *term*s (variables and constants) 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 other agent (M ==> Agent), and receive a message from another agent (M <== Agent). There are two types of constraints over the message exchanged: pre- and post-condition. Pre-conditions (M ==> Agent ←C) specify the required conditions for an agent to send a message and for the receiver to accept and process it. Post-conditions (C ←M <== Agent) explain the states of the sender after sending a message and of the receiver after receiving a message. An agent can check the satisfaction of the constraints inspecting private or shared knowledge.

3.2.1 An Example LCC protocol

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

```
a(R1,P)::
   claim(℧) ==> a(R2, O) ← C1
   then
         a(R3,P).
a(R2,O)::
   claim(℧) <== a(R1, P)
   then
          a(R4,O).
```

This is read as: role *R1* of agent *P* sends a claim message, which is achieved by the constraint *C1*, to the role *R2* of agent *O* and then role *R2* of agent *O* receives the claim message from role *R1* of agent *P*. Then *P* change its role to *R3* and *O* change its role to *R4*.

3.3 Synthesis of Concrete Protocols from DID

The main aim of this research (as shown in Figure 2-part1) is to automatically synthesise LCC protocols from DID specifications by recursively applying LCC-Argument patterns.

3.3.1 LCC- Argument Patterns

LCC-Argument patterns were first described in the structured design method by Grivas [5]. The general idea is analogous to that used in *Techniques editing* [14], to synthesise Prolog clauses. The most notable differences between our LCC-Argument patterns and Grivas' [5] patterns are: 1. Grivas did not base his system on a high level language; 2. Grivas describes the patterns which appearing in general LCC protocols, while our patterns are specific to argumentation. LCC-Argument patterns are generic LCC argument codes which are independent of any particular algorithm or problem domain. LCC–Argument patterns provide generalised pieces of LCC code which can be reused by software engineers to implement part of a LCC specification. The reuse of patterns could potentially reduce the effort of building interaction protocols. Maghraby [15] describes these patterns in details. To expedite our argument, we will not repeat these here. Instead we will describe the simplest LCC-Argument pattern called the *Starter pattern*. This pattern is used to start the dialogue between two agents (*P* and *O*). To explain this pattern we will use the following five general characterisations [16,17]: *1.Problem*: a statement or a question of the problem which describes the problem that the pattern solves; *2.Solution*: relationship between the pattern's roles which describes how to realize the desired outcome, often including a diagram which describes how the problem is solved; *3.Context (Pre-conditions)*: the initial configuration of the protocol before the pattern is applied; *4.Consequence (Post-conditions):* the configuration of the protocol after the pattern has been applied; *5.Structure*: identify the pattern's structure, its roles and their relations.

Starter pattern

1. Problem: How to start a dialogue?;
2. Solution: This pattern is composed of two roles: sender role,R_{P1} , and receiver role,R_{O1}.The general idea of this pattern (as shown in Figure 6) is that the agent with role R_{P1} sends an

initial message, $SL(Ʊ)$, to the agent playing role R_{O1} and then both change their roles in order to remain in the dialogue;

3. Context: Use *Starter Pattern* when *P* agent has not already started a dialogue;

4. Consequence: (a) Both *P* and *O* agents engage in a dialogue; (b) *P* agent is committed to $Ʊ \in CS_P$ (updated its commitment store by adding $Ʊ$ to it); (c) Both *P* and *O* change their roles so as to remain in dialogue;

5. Structure: *Starter Pattern* structure is shown in Figure 7. *SL* represents the starter locution and $C1(Ʊ,CS_P)$ represents a Boolean function (or condition) with parameters $Ʊ$ and CS_P. In case $C1(Ʊ,CS_P)$ returns true the locution *SL* can be uttered.

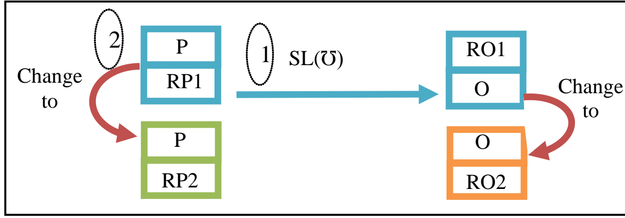

Figure 6: Starter Pattern Solution

$a(R_{P1}(KB_P,CS_P, Ʊ, ID_O),ID_P)$::
$SL(Ʊ) \Longrightarrow a(R_{O1}(KB_O,CS_O),ID_O) \leftarrow C1(Ʊ, CS_P)$
then
$a(R_{P2}(KB_P,CS_P),ID_P)$.
$a(R_{O1}(KB_O,CS_O, ID_P),ID_O)$::
$SL(Ʊ) \Longleftarrow a(R_{P1}(KB_P,CS_P),ID_P)$
then
$a(R_{O2}(KB_O,CS_O),ID_O)$.

Figure 7: Starter Pattern Structure

$a(startClaim_P(KB_P,CS_P, Ʊ,ID_O),ID_P)$ ::=
$claim(Ʊ) \Rightarrow a(startClaim_O(_ ,_ ,ID_P),ID_O)$
<-- $AddToCS(Ʊ, CS_P)$
then
$a(replyToClaim_P(KB_P,CS_P, Ʊ,ID_O),ID_P)$.
$a(startClaim_O(KB_O,CS_O,ID_P),ID_O)$ ::=
$claim(Ʊ) \Leftarrow a(startClaim_P(_ , _ , Ʊ,ID_O),ID_P)$
then
$a(replyToClaim_O(KB_O,CS_O, Ʊ,ID_P),ID_O)$.

Figure 8: General LCC Protocol for Claim Locution

3.4 Example

Due to space limitations, a full description of the LCC protocol generated from the DID specification depicted in Figure 4 is not possible. Instead we will describe how to automatically synthesise, using the *Starter Pattern* (Figure 7), a partial LCC protocol (Figure 8) from the starting locution (the claim) from the DID in Figure 4:

1. Determine the starting locution (*SL*) in the DID. As we can see from Figure 4, there is one starting locution which is located at the top of the DID, *SL=claim;*

2. Apply the *Starting Pattern*. . By matching variables from the pattern (Figure 7) with variables from the DID (Figure 4). In this example :

    a. Matching *SL* to *claim*;

    b. Matching the sender role R_{P1} with the sender role $startClaim_P$ from the *claim locution*;

    c. Matching the receiver role R_{O1} with the receiver role $startClaim_O$ from the *claim locution*;

    d. Matching $C1(Ʊ,CS_P)$ in the sender role with $AddToCS(Ʊ, CS_P)$ in the rhombus shape which is connected to the left hand section of *claim locution*;

    e. Matching the lines after the word "then" in both the sender and receiver roles to the *why (or concede) locution* icon . We match the lines after "then" with the roles of the locution of the next level of DID. When we move from level to the next level in the DID, the turn-taking between agents switches denoting that the sender will be in the right hand section of the *locution* icon and the receiver will be in the left hand section of the *locution icon*. Therefore (e) consists in:

        e1. Matching the sender role R_{P2}, with the sender role $replyToClaim_P$ from the right hand section of the *why (or concede) locution*;

        e2. Matching the receiver role R_{O2}, with the receiver role $replyToClaim_O$ from the left hand section from the *why (or concede) locution*.

4. MODEL CHECKING

Our model checker (as shown in Figure 2- part 2) was built in three stages: (1) automatically mapping the LCC specification into an equivalent Coloured Petri Net (CPN) [18]. The formal semantics of the CPN models allow us to prove that certain (un)desirable properties are (un)satisfied in a LCC protocol; (2) automatically generating DID properties as a Standard ML specification. For instance, in the DID shown in Figure 4 the *claim* locution is a starting locution, therefore we can infer as a significant property that every LCC synthesised dialogue should start with a *claim* locution; (3) automatically verifying the satisfaction of the Standard ML specification in the state-space graph computed from the LCC protocol.

4.1 Mapping the LCC Specification into a CPN

CPNs are defined as Petri Nets (PNs) which have been extended with the notion of colors or types. As a variant of PN CPNs are defined as networks of input/output places (ovals), transitions (squares) and arcs (arrows connecting places with transitions and transitions with places). Colours (types) and tokens are used to simulate the network flow. For instance, the CPN modeled in the CPN tool[1] (depicted in Figure 9), the $StartClaim_P$ is a transition with input places *Open* and *P* and output place *claim1* and *ChangeRole1*. The Message colour is a composed type (comprising topic, sender identifier and receiver identifier) used to represent messages exchanged between agents, while the Role colour is used to represent the agent's profile (played role, agent's identifier, agent's private knowledge based, agent's CS, topic and other agent's identifier). For the example depicted in Figure 9 an agent can send a claim if an open place is active (there is a token

[1] http://cpntools.org/

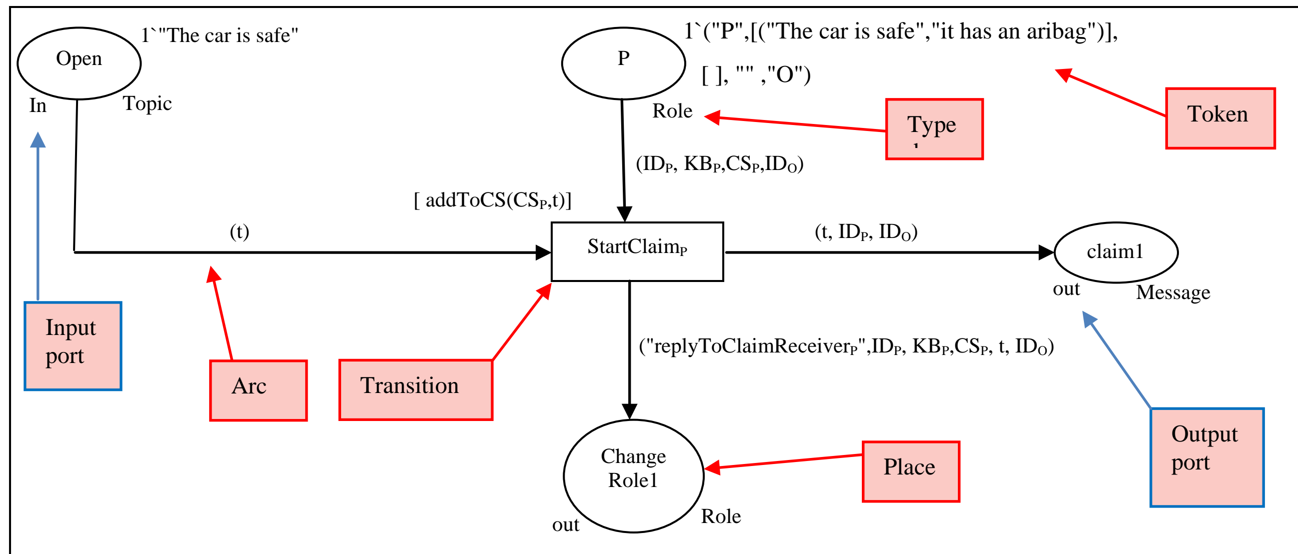

Figure 9: CPN Protocol Graph for startclaim$_P$ LCC Role

in *Topic* state) and an agent playing role StartClaim$_P$ is active (there is a token in state *P*).

One of the key features of CPNs is their ability to construct large models in a hierarchical manner [19] by using a set of CPN modules called subpages to build superpages. The pages interact with each other and with superpage through a substitution transitions and a set of interfaces (fusion places).

A substitution transition is a transition (drawn as rectangular double lines boxes in Figure 10) which is located in a superpage and refined by a subpage. A fusion place is composed of one socket place and one port place. In practice, sockets and ports represent the 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 receive date 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 port and receive/send data from/to other CPNs models.

The steps applied to get the CPNs files from LCC protocol are as follows:

1. Generation of a CPN subpage for each LCC role. This page represents the different internal behviours of each role. Each role (as shown in Figure 9) has at least one input port and one output port.
2. Generation of one CPN superpage which describes the interaction between roles where messages passed between two roles determine the interaction between the subpages of the two roles (shown in Figure 10 and explained in detail in Section 4.4).

4.2 Generating DID Properties

Our tool can automatically generate from the DID specification used as starting point a set of properties, expressed as Standard ML specifications, which can be verified over the synthetised LCC protocol. For the persuasion dialogue explained in Section 3.1.3 the tool identified five properties:

1. 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.
2. Property-2 Termination of a dialogue: This property should guarantee that the LCC protocol will end when a specific agent sends a DID termination locution.
3. Property-3 Turn taking between agents: This property should guarantee that in the LCC protocol the turn-taking between agents switches after each move (after agent sends a message).
4. Property-4 Message Sequence: This property should guarantee that the LCC protocol message exchange respects the DID. For instance for the DID depicted in Figure 4 one thing that should be proved is that after an agent makes a claim the other agent can only answer with a concede or a why locution.
5. Property-5 Recursive Message: This property should guarantee that the LCC protocol recurs when agent sends a message with a recursive DID locution.

These five properties are provided by the module checker system. However, the system allows users to add and run more properties.

4.3 Verification of Properties

Proof of the properties generated from the DID (explained in Section 4.2) over the CPNs resulting from the LCC protocols synthesised (explained in Section 4.1) is supported by a state space technique [20].

For developing our model checker we used the CPN tool which allows to automatically compute all possible execution states resulting from the exhaustive enactment of a CPN model.

The verification process consists on automatically checking the satisfaction of the properties specified in Standard ML over the obtained state space graph. A report is presented to the user indicated which properties are satisfied and which are unsatisfied.

4.4 Example

4.4.1 Mapping the LCC Specification into CPNs

Due to space limitation an exhaustive description of the mapping of the LCC protocol presented in Figure 8 into a CPN model is not possible, instead we describe below sections of the resulting CPN model which capture the main features of our modelling approach.

As described earlier in section 4.1, the first step to map the LCC specification into a CPN model is to construct a new CPN subpage for each role in the LCC protocol. As we can see from Figure 8 in our example there are two main roles: startclaim_P and startclaim_O. Therefore two subpages are created: startclaim_P and startclaim_O. The resulting subpages are similar , therefore, we will only describe one of them, the startclaim_P subpage.

StartClaim_P Subpage

As described earlier in section 3, the $startclaim_P$ role begins by sending an initial message, *claim(Topic),* to the $startclaim_O$ role and then changes its role to $replyToClaimReceiver_P$.

Figure 9 shows the subpage $startclaim_P$, resulting from mapping the LCC role with the same name into a CPN. The name of the role is represented by the transition $startclaim_P$. To model this, an agent with agent identifier (ID_P) and role parameters (KB_P , CS_P , ID_O) can introduce a starting claim locution which we specify as a type Role and we assign a token of type Role to the place *P*.

When a token of type Message is present in the place *claim1*, it means that a message has been sent from an agent playing the role $startclaim_P$ to an agent playing the role $startclaim_O$ which specifies all this information (topic *t*, sender agent ID_P, receiver agent ID_O).

The condition under which the role can be sent this message is represent as a transition, $startclaim_P$, condition.

The changing of role $startclaim_P$ to $replyToClaimReceiver_P$ is represented by the output place *changeRole1,* which is of type Role. The name of the new role and its parameters are captured in the output inscription arc from $startclaim_P$ transition to the place *changeRole.*

The *open* input place, which is of type Topic, represents the required data to start a dialogue.

Secondly, constructing one protocol superpage to describe the general interaction relation between $startclaim_P$, $startclaim_O$, $replyToClaimReceiver_P$ and $replyToClaimSender_O$ roles.

Protocol Superpage

The resulting superpage is shown in Figure 10. The four roles are represented by the substitution transition. The message *claim* between $startclaim_P$ and $startclaim_O$ is represented by the place *claim1*. The role changing relation between $startclaim_P$ and $replyToClaimReceiver_P$ is represented by the place *ChangeRole1* and the role changing relation between $startclaim_O$, and $replyToClaimSender_O$ is represented by the place *ChangeRole2*. Connect the *open* place, which represents the required data to start a dialogue, with $startclaim_P$ substitution transition.

Thirdly, generate state space. State space cannot be generated unless all places in subpages have been initialized. To initialize places tokens, we use a simple car safety example. Two agents *P* and *O* interchange argument. Where each agent has its own KB and CS. Agent *P* has: (1) KB_P =[("The car is safe","it has an aribag")]; (2) CS_P =[""]. Agent *O* has (1) KB_O =[("it has an aribag","The car is safe")]; (2) CS_O =[""]. Therefore, as shown in Figure 9, we initialize *P* palce with ("P",[("The car is safe","it has an aribag")], [ ], "" ,"O"). Agent *P* want to open a dialogue by sending a claim message "My car is safe" to agent *O*. Therefore, as shown in Figure 9, we initialize *Open* place with "The car is safe".

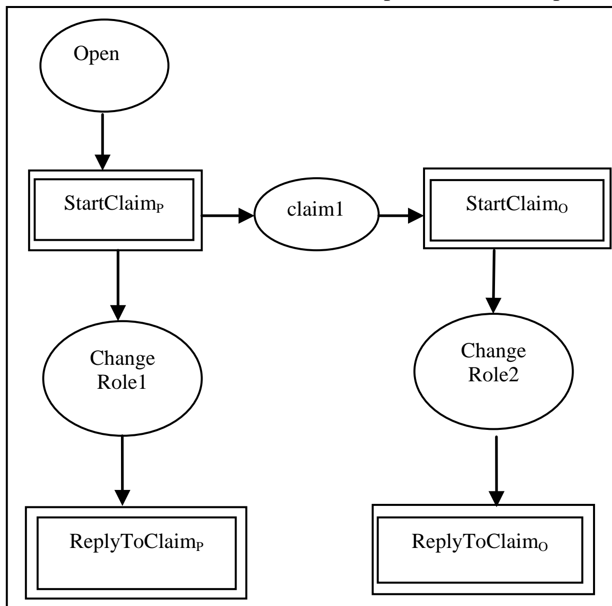

Figure 10: CPN Protocol Graph for Claim Locution

4.4.2 Verification of the LCC Protocol

Over the CPN depicted in Figure 10 there is one property (Property-1 Dialogue opening) which can be verified. Figure 11 shows the Standard ML specification of this property. Function CheckProperty1 compares the first message in the state space graph constructed from the CPN shown in Figure 9 and Figure 10 with the starting claim locution from the DID in Figure 4.

Lines 3-5 explain how to get from the state space graph the first exchanged message. Line 6 compares the first exchanged in the state space graph with the starting locution from the DID. Lines 8 to 11 are used to inform the user of the result of the comparison. A positive (negative) result indicates that Property 1 is satisfied (unsatisfied).

For space reasons we only show here how we verified Property 1, but we have proved the satisfaction of all the properties explained in Section 4.2 over the LCC protocol synthesised from the whole DID persuasion dialogue from where the partial DID depicted in Figure 4 was extracted.

5. CONCLUSION

This paper describes an approach to bridging the gap between argument specification and multi-agent implementation using AIF as an example of an argumentation language and LCC as an example of a multi-agent implementation (coordination) language. The proposed approach is based on a pattern-based synthesis method and it allows us to automatically transform the new specification argument language (DID) to peer-to-peer LCC protocols. Model checking is used to ensure that key properties of the DID specification are preserved by the resulting LCC

protocol. Based on this approach, we have implemented a tool which automatically synthesises LCC protocols from DID specifications. The tool has been provided with a model checker which automatically verifies that the LCC protocols synthesised by the tool preserving the properties from the DID specification used as starting point. As future work we would like to explore the tool's functionality in the context of complex dialogues such as embedded agent dialogues.

```
1     fun CheckProperty1(DIDoDmessages) =
2     let
3        val arcMove1=st_BE(ArcToBE(2))
4        val arcMove1Size = String.size(arcMove1)
5        val message1= extractString(arcMove1,
                            "l=",",", arcMove1Size ,0)
6        val checkODM =
         compare(DIDoDmessages , message1)
7     in
8     if (checkODM )  then
9     "Property 1(Dialogue opening ) is Satisfied"
10    else
11    "Property 1(Dialogue opening) is not
       Satisfied"
12    end;
```

Figure 11: Property 1 as an Standard ML Function.

6. ACKNOWLEDGMENTS

I would like to thank my supervisors, Prof. David Robertson, Dr. Adela Grando and Michael Rovatsos from Edinburgh University, for their supervision and contributions to this work.

7. REFERENCES

[1] Rahwan I. and Moraitis P. 2009. Argumentation in Multi-Agent Systems. In Proceedings of the 5th International Workshop on Argumentation in Multi-Agent Systems (ArgMAS2008).

[2] Chesnevar, C., McGinnis, J., Modgil, S., Rahwan I., Reed, C., Simari, G., South, M., Vreeswijk, G., Willmott, S. 2007. Towards an argument interchange format. The Knowledge Engineering Review, 21(4), 293–316.

[3] Willmott, S., Vreeswijk, G., Chesnevar, C., South, M., McGinnis, J., Modgil, S., Rahwan I., Redd C., Simari G., 2006. Towards an Argument Interchange Format for Multi-Agent Systems. In Proceedings of the 3ed International Workshop on Argumentation in Multi-Agent Systems (ArgMAS2006).

[4] Robertson D. 2004. Multi-agent coordination as distributed logic programming. In "Logic programming" 20th International Conference, Proceedings, Lecture Notes in Computer Science, 3132,416-430.

[5] Grivas A. 2005. A Structural Synthesis System for LCC Protocols. PhD thesis, University of Edinburgh.

[6] Modgil S. and McGinnis J. 2007. Towards Characterising Argumentation Based Dialogue in the Argument Interchange Format. In Proceedings of the 4th International Workshop on Argumentation in Multi-Agent Systems (ArgMAS2007), 80-93.

[7] Jensen K. and Kristensen L. 2009. Coloured Petri Nets Modelling and Validation of Concurrent System*s*. Springer Verlag, 43–77.

[8] Ullman J. 1998. Elements of ML Programming. Prentice-Hall,Englewood Cliffs.

[9] Prakken, H. 2000. On dialogue systems with speech acts, arguments, and counterarguments. Springer Verlag, 224–238.

[10]Rahwan I., Zablith F. and Reed C. 2007. Laying the Foundations for a World Wide Argument Web. Artificial Intelligence Journal, 171(10-15),897—921.

[11] Maudet N., Parsons S., and Rahwan I. 2007. Argumentation in Multi-Agent Systems: Context and Recent Development. In Proceedings of the 3ed International Workshop on Argumentation in Multi-Agent Systems (ArgMAS2006).

[12] Bauer B., Müller J., and Odell J.: Agent UML: A Formalism for Specifying Multiagent Interaction. *Agent-Oriented Software Engineering*, Paolo Ciancarini and Michael Wooldridge eds., Springer, Berlin, pp. 91-103, 2001.

[13] Hassan F., Robertson D., and Walton C. 2005. Addressing Constraint Failures in Agent Interaction Protocol. Centre for Intelligent Systems and their Applications, University of Edinburgh.

[14]Bowles A., Robertson D., VasconcelosW., Vargas-Vera M., and Bental D. 1994. Applying prolog programming techniques. International Journal of Human-Computer Studies, 41(3), 329–350.

[15] Maghraby A. 2011. LCC argument patterns. School of Informatics, Edinburgh university.

DOI= http://homepages.inf.ed.ac.uk/s0961321/index.html

[16] Appleton B. 1998. Patterns and Software: Essential Concepts and Terminology. Object Magazine Online ,3(5). DOI= http://www.bradapp.net/

[17] TaylorG. and Wray R. 2004. Behavior Design Patterns: Engineering Human Behavior Models. In Proceedings of the 2004 Behavioral Representation in Modeling and Simulation Conference (BRIMS).

[18] Jensen K. 1997. Coloured Petri Nets:Basic Concepts, Analysis Methods and Practical Use. Basic Concepts. Monographs in Theoretical Computer Science, Springer-Verlag.

[19] Jensen K., Kristensen L., and Wells L. 2007. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. Int.J.Softw.Tools Technol.Transf., 9(3):213–254.

[20]Jensen K., Christnsen S. and Kristensen L. 2006. CPN Tools State Space Manual. University of Aarhus, Department of computer science.