Computational Models of Argument
B. Verheij et al. (Eds.)
IOS Press, 2012

© 2012 The authors and IOS Press. All rights reserved.
doi:10.3233/978-1-61499-111-3-197

Automated Deployment of Argumentation Protocols

Ashwag MAGHRABY, Dave ROBERTSON,
Adela GRANDO and Michael ROVATSOS
School of Informatics, The University of Edinburgh, Edinburgh EH8 9LE, UK
A.O.Maghraby@sms.ed.ac.uk

Abstract. The objective of this paper is to try to fill the gap between: argumentation, electronic institutions and protocols by using a combination of automated synthesis and model checking methods. More precisely, this paper proposes a means of moving rapidly from argument specification to protocol implementation, using an extension of the Argument Interchange Format as the specification language and the Lightweight Coordination Calculus as an implementation language.

Keywords. Argumentation, Dialogue Games, Automated Synthesis, Interaction models, Verification, Model Checking.

Introduction

Coordinating agents in open environments is a difficult problem that has engaged multi-agent systems researchers on three broad fronts: (1) argumentation [1] (the basis for negotiation between agents); (2) electronic institutions [2] (the norms of social interaction in agent groups) and (3) protocols (which focuses on deployment of interactions). None of these areas subsumes the others but there is a strong interaction between them in many cases. For example, if one wishes to construct a multi-agent trading system then this will contain negotiation, restrictions on agents' group behaviour and protocols relevant to each agent.

This is broadly analogous to the relationships between different views of formal system requirements in software engineering, where different viewpoints give a structure within which complementary aspects of a system may be expressed. Accordingly, an interesting challenge is how (or whether) the specification of one of these components (argumentation, electronic institutions and protocols) can be used to constrain the specification of the others. One way of addressing this issue is through automated synthesis.

In this paper, a means of towards closing the gap between these three components is suggested. We demonstrate 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 Argument Interchange Format ***AIF*** [3] (a generic specification language for argument structure). As our protocol language we have chosen the Lightweight Coordination Calculus ***LCC*** [4] (an executable specification language used for coordinating agents in open systems).

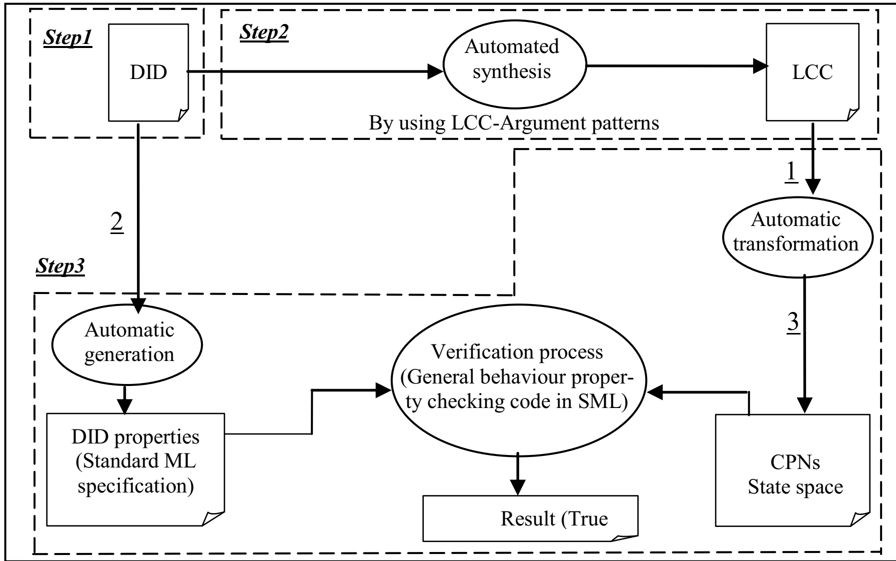

Figure 1. Overall structure of this research

1. Overall Structure of our System

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 three steps (as shown in Figure 1): (1) Proposing a new high-level specification language, between the AIF and LCC, for multi-agent protocols called a Dialogue Interaction Diagram (DID). The definition of DIDs is provided in section 2; (2) Synthesising concrete LCC protocols from DID specifications (automatically synthesise LCC protocols from DID specifications by recursively applying LCC-Argument patterns). The fully automated synthesis is provided in section 3; (3) Providing a verification methodology based on model checking to verify the semantics of the original DID specification against the semantics of the synthesised LCC protocol. The verification methodology is provided in section 4.

2. Dialogue Interaction Diagram Language

The Argument Interchange Format (AIF) [3] would be a natural choice of a high level specification language but fully automated synthesis starting only from the AIF is not possible because AIF 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. Papers [5,6] discuss this problem in more detail.

To remedy the AIF problem, we will propose a new intermediate language between the AIF and LCC called a Dialogue-Interaction-Diagram (DID), which contains information that cannot be deduced from AIF. In practice, DID is a new layer on top of AIF. DID is used to specify the dialogue game protocols in a compact way. It has nine elements: (1) Locutions: represent the set of permitted moves; (2) Participants Com-

mitment Store: 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 from the previous move; (5) Turn taking rules: specify the next player; (6) Starting rules (commencement rules): define the conditions beginning the dialogue; (7) Termination rules: define the conditions ending the dialogue; (8) Precondition rules: define the preconditions under which the move will be achieved; (9) Sender and receiver roles: a set of functions that an agent can used to interact with each other. Each role identifies the messages that an agent can send or receive.

DID is not a general protocol specification language. In particular, it is more restrictive than any protocol specification language (such as LCC). It restricts agent moves to unique-moves (*agents can make just one move before the turn-taking shifts and agents can reply just once to the other agent's mov*e) 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*). This still allows us to include a large class of argumentation systems in our 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 recursive steps (each of which involves turn-taking between the pair of agents) terminating in a base case.

2.1. DID Elements

The basic element of every DID is a locution icon. A locution icon (as shown in Figure 2) is simply a rectangle divided into three sections. The topmost section contains the name of the locution. The left section contains sender attributes (Role name, Role arguments, and Agent ID), and the right 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 section it represents sender conditions and when connected to the right 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).

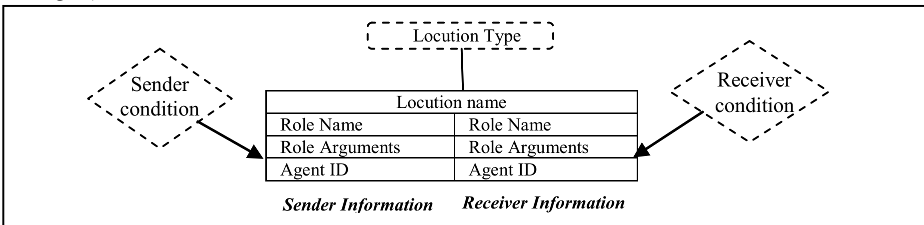

Figure 2. Locution icon

2.2. DID Example

A DID is created by linking the locution icons together. The links between locution icons represent reply relations between arguments. In Figure 3, there are three locutions: two attack locutions which have a reply move (*claim*, and *why*), and one surrend-

er 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 apply 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 list of arguments expressed in the AIF to which the player has committed during the discussion. Initially the CS is empty.

In this dialogue, $Agent_{A1}$ can open the discussion by sending a *claim*(Topic) (e.g. *claim("Tweety flies")*) locution if it is able to satisfy $AddToCS(Topic,CS_{A1})$ condition ($AddToCS(Topic,CS_{A1})$ which is used to update the agent commitment store CS_{A1} by adding *Topic* to it). Then, turn-taking switches to $Agent_{A2}$. $Agent_{A2}$ has to choose between two different possible reply locutions: *why*(Topic) (e.g. *why(Why does Tweety fly?)*) or *concede*(Topic) *(e.g. concede(You are right. Tweety flies)*). $Agent_{A2}$ will make its choice using the conditions which appear in the rhombus shape. In order to choose *concede*(Topic), $Agent_{A2}$ must be able to satisfy the two conditions which connect with concede: Condition 1: $FindInKBorCS(Topic,KB_{A2},CS_{A2})$ which is used to check whether *Topic* is acceptable in the agent argumentation system KB_{A2} and CS_{A2} or not. If *Topic* is acceptable, this constraint returns true; Condition 2: $AddToCS(Topic,CS_{A2})$ $Agent_{A2}$ will use this constraint to update its commitment store by adding *Topic* to CS_{A2}. If $Agent_{A2}$ is not able to satisfy these conditions, $Agent_{A2}$ will send *why*(Topic). After that, the turn switches to $Agent_{A1}$, and so forth.

Although this example is simple, DID can handle embedded dialogues (complex dialogues) [7] which involve embedding more than one type of dialogue game within another game such as an embedded persuasion dialogue within an inquiry dialogue.

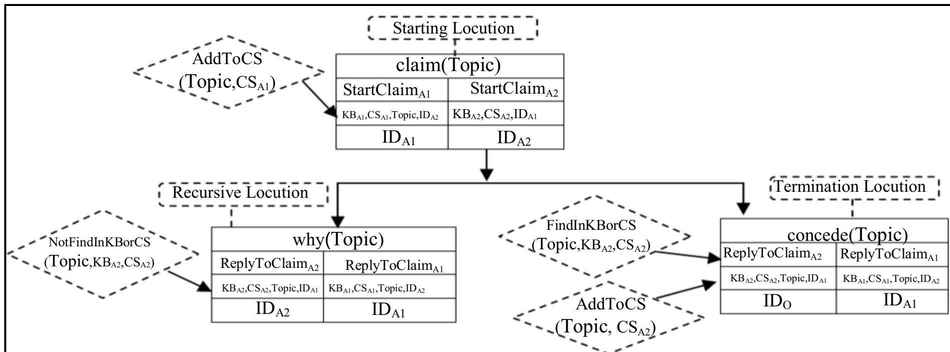

Figure 3. Partial DID for a Persuasion Dialogue

2.3. Differences between DID and Existing Languages

The most notable differences between DID and existing languages for argumentation-based interaction protocols are: (1) DID arguments are ideally expressed in AIF. Others have assumed specific argument formats which are dependent on the type of dialogue or the particular context of application considered. 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]; (2) DID uses a high-level graphical language resembling ones with which people in the agent community are familiar, such as Agent UML [8]. Also, specifying argumen-

tation protocols using programming-level protocol languages is error-prone, and a higher-level graphical language can help avoid low-level errors through automated synthesis of low level details.

3. Automated Synthesis Method

Given the turn-taking assumption implicit in DID diagram, we can synthesise agent 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, 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 to generate argumentation protocols.

3.1. LCC

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

An Example LCC protocol

We now demonstrate LCC using the simplest example of a persuasion protocol between two agents *A1* and *A2*. *A1* and *A2* have arguments for and against *Topic* (e.g. the Flying Abilities of the "Tweety" Bird). Agent *A1* sends a claim(*Topic)* message and agent *A2* receives this claim(*Topic)* message. A fragment of theLCC protocol for this interchange in this argument is:

```
a(R1,A1)::
        claim(Topic) => a(R2, A2) ← C1    then     a(R3,A1).
a(R2,A2)::
        claim(Topic) <= a(R1, A1)   then     a(R4,A2).
```

This is read as: role *R1* of agent *A1* sends a claim message, which is achieved by the constraint *C1*, to the role *R2* of agent *A2* and then role *R2* of agent *A2* receives the claim message from role *R1* of agent *A1*. Then *A1* changes its role to *R3* and *A2* changes its role to *R4*. See paper [4] for more information about the abstract syntax of an LCC clause.

3.2. LCC- Argument Patterns

Our patterns capture the different relationships and interactions between LCC agents' roles. These LCC-Argument patterns provide common 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.

Maghraby [9] describes these patterns in detail so we will not repeat these here. Instead we give in Figure 4 the simplest LCC-Argument pattern called the *Starter pattern*.

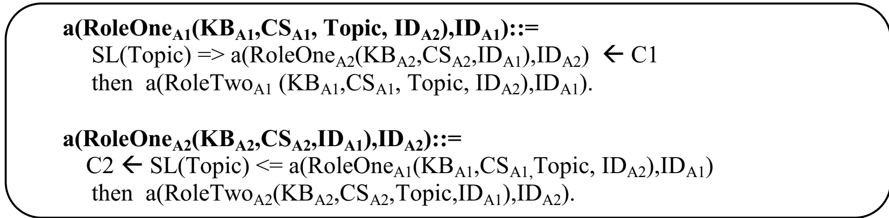

Figure 4. Starter Pattern Structure

This pattern is used to start the dialogue between two agents (*A1* and *A2*). It is composed of two roles: sender role, RoleOne$_{A1}$, and receiver role, RoleOne$_{A2}$.The general idea of this pattern is that the agent with role RoleOne$_{A1}$ sends a starting message, *SL(Topic),* to the agent playing role RoleOne$_{A2}$ and then both change their roles in order to remain in the dialogue.

Patterns are parameterisable LCC components that, when the parameters are initiated via DID, return executable LCC code. For the *Starter pattern* in Figure 4, the designer must supply: the names of the roles of the two agents (RoleOne$_{A1}$, RoleOne$_{A2}$, RoleTwo$_{A1}$, RoleTwo$_{A2}$); the name of the initial message (SL); the condition on sending the initial message (C1); and the condition on receiving it (C2).

3.3. Automated Synthesis of Agent Protocols from DID

The main aim of this research (as shown in Figure 1- step 2) is to synthesise LCC protocols automatically from DID specifications by recursively applying LCC-Argument patterns.

Automated Synthesis Steps Example

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 became the receiver and vice versa). The automated synthesis process follows the diagram from top to bottom and from left to right. This matches each level of the DID with only one LCC-Argument pattern.

In this section, we will describe how to synthesise a partial LCC protocol from the starting locution (the claim) from the DID in Figure 3 automatically, using the *Starter Pattern* (Figure 4)*,* Figure 5 illustrates the final LCC protocol.

The automated synthesis process of two-agents protocol consists of four steps: (1) The tool begins with the locution icon at the top of the DID of the persuasion dialogue, which is *claim(Topic)*. 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. This pattern depends on the locution type. In this example, the tool selects the *Starting Pattern* (since the locution type is *Starting Locution*). (3) Next, the tool applies the *Starting Pattern* by matching formal parameters in the *Starting Pattern* to its corresponding values in the *claim(Topic)* icon, starting from the top to bottom and left to right and

matches: (a) Starting from the top of the locution icon, the tool matches *SL* to *claim(Topic);* (b) Moving to the left section of the locution icon, the tool matches $RoleOne_{A1}$ with $StartClaim_{A1}$, role parameters with $(KB_{A1},CS_{A1},Topic,ID_{A2})$, and role id with ID_{A1}; (c) Moving to the right section of the locution icon, the tool matches $RoleOne_{A2}$ with $StartClaim_{A2}$, role parameters with $(KB_{A2}, CS_{A2},ID_{A1})$, and role id with ID_{A2};(d) Moving to the left section conditions, the tool matches *C1*with *AddToCS(Topic,* CS_{A1}*)*; Moving to the next level, 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 *A1's* recursive role with the right section of the locution icon. It matches $RoleTwo_{A1}$ with $replyToClaim_{A1}$, role parameters with $(KB_{A1},CS_{A1},Topic,ID_{A2})$, and role id with ID_{A1}. Then, the tool matches agent *A2*'s recursive role with the left section of the locution icon. It matches $RoleTwo_{A2}$ with $replyToClaim_{A2}$, role parameters with $(KB_{A2}, CS_{A2}, Topic,ID_{A1})$, and role id with ID_{A2}. (4) Finally, the tool moves to the next level (in this example, it moves to level two) in the DID and repeats steps 2 and 3. Note that the automated synthesis process finishes when the tool matches the last level in the DID with one LCC-Argument pattern. 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 the this level.

$a(startClaim_{A1}(KB_{A1},CS_{A1}, Topic, ID_{A2}),ID_{A1})$::=
claim (Topic) => $a(startClaim_{A2}(KB_{A2},CS_{A2},ID_{A1}),ID_{A2})$ ← $AddToCS(Topic, CS_{A1})$
then $a(replyToClaim_{A1}(KB_{A1},CS_{A1}, Topic, ID_{A2}),ID_{A1})$.
$a(startClaim_{A2}(KB_{A2},CS_{A2},ID_{A1}),ID_{A2})$::=
claim(Topic) <= $a(startClaim_{A1}(KB_{A1},CS_{A1},Topic, ID_{A2}),ID_{A1})$
then $a(replyToClaim_{A2}(KB_{A2},CS_{A2},Topic,ID_{A1}),ID_{A2})$

Figure 5: General LCC Protocol for Claim Locution

3.4. Automated Synthesis of Agent Protocols for N-agents

The DID notation is general across all dialogues but is limited to two agents. Our design patterns extend to N-agents but only accommodate those protocols that fit the patterns, so we extend protocol coverage but we are only as general as our library of patterns. However, we can reclaim some generality for patterns in which parts of the protocol are dialogues. Our 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. Practically, our automated synthesis method uses an LCC-argumentation broadcasting pattern [9] to divide agents into groups composed of two agents and then it follows the automated synthesis process of two agents' protocol (see section 3.3) to generate the LCC protocol for DID for two agents, which allows the selected pairs of each group to communicate with each other.

4. Verification Method

We also provide a verification methodology based on model checking (as shown in Figure 1- step 3) to verify the semantics of the DID specification against the semantics of the synthesised LCC protocol. Space limitations prevent us from gives details of this but we sketch the main elements here. The model checking system is built in three stages: (1) automatically mapping the LCC specification into an equivalent Coloured

Petri Net (CPN) [10]. The formal semantics of the CPN model allows us to prove that certain (un)desirable properties are (un)satisfied in a LCC protocol. Proof of properties in LCC protocols mapped into CPNs is supported by a state-space technique, which is used to compute exhaustively all possible execution states; (2) automatically generating DID properties as a Standard ML(Meta-Language) specification. For instance, in the DID shown in Figure 3 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. For more details please see [5].

5. CONCLUSION

This research describes a synthesis and verification 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. Although the resulting synthesis and verification system is not an industry-strength specification tool, it demonstrates how automated synthesis methods can connect argumentation to the class of electronic institutions that can be expressed as 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).

References

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

[2] Esteva M., Vasconcelos W., Sierra C., and Rodríguez-Aguilar J. Norm consistency in electronic institutions. In Proceedings of the XVII Brazilian Symposium on Artificial Intelligence - SBIA'04, Lecture Notes in Artificial Intelligence, volume LNAI. 2004; 3171: 494-505.

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

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

[5] Maghraby A. Automatic Agent Protocol Generation from Argumentation. 13th European Agent Systems Summer School, Girona, Catalonia (Spain). 2011.

[6] Maghraby A., Robertson D., Grando A., and Rovatsosr M. Bridging the Speci_cation Protocol Gap in Argumentation. The Ninth International Workshop on Argumentation in Multi-Agent Systems (ArgMAS2012).2012.

[7] Walton D. and Krabbe E. Commitment in Dialogue: Basic concept of interpersonal reasoning. State University of New York Press, Albany, NY, USA.1995.

[8] 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. 2001; 11: 91-103.

[9] Maghraby A. LCC argument patterns. School of Informatics, Edinburgh university. 2011. Available from: http://homepages.inf.ed.ac.uk/s0961321/index.html

[10] Jensen K., Kristensen L., and Wells L. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. International Journal on Software Tools for Technology Transfer (STTT). 2007; 9(3):213–254.