Argumentation and Artifact for Dialog Support

Enrico OLIVA [a,1], Mirko VIROLI [a], Andrea OMICINI [a] and Peter MCBURNEY [b]
[a] ALMA MATER STUDIORUM–*Università di Bologna, Cesena, Italy*
[b] *University of Liverpool, Liverpool L69 3BX UK*

Abstract. Intelligent and autonomous software agents may engage in dialog and argument with one another, and much recent research has considered protocols, architectures and frameworks for this. Just as with human dialogs, such agent dialogs may be facilitated by the presence of a mediator, able to summarize different positions, identify common assumptions and inconsistencies, and make appropriate interventions in the dialog. Drawing on the theory of co-ordination artifacts in multi-agent systems, we propose a formal framework to explicitly represent the functions of a mediator artifact. We then describe an implementation of this framework using the TuCSoN coordination infrastructure for MAS, where the mediator artifact is realized by a tuple centre—a programmable tuple space.

Keywords. Argumentation, Artifacts, Dialogs, Mediators, Multi-Agent Systems

1. Introduction

Proponents of public policy conversations and decision-making processes usually emphasize the need for a human moderator or mediator to be involved in the interaction, e.g., [3]. The mediator may act to ensure fairness and equality of access by all participants, may assist participants to clarify their positions and to argue more effectively, and may even seek to reconcile opposing views. Similarly, the designers of computer-aided argumentation systems have also provided support for human mediators; for example, the developers of *Zeno* define their system as "a mediation system" [5, p. 10]:

> *"a kind of computer-based discussion forum with particular support for argumentation. In addition to the generic functions for viewing, browsing and responding to messages, a mediation system uses a formal model of argumentation to facilitate retrieval, to show and manage dependencies between arguments, to provide heuristic information focusing the discussion on solutions which appear most promising, and to assist human mediators in providing advice about the rights and obligations of the participants in formally regulated decision making procedures."*

Just as with human interactions, and for the same reasons, many of the functions provided by mediators could be useful when software agents engage in argumentation with one another. Some of these mediator functions require only limited intelligence, for example,

[1]Corresponding Author: Enrico Oliva, Università di Bologna, via Venezia 52, 47023, Cesena, Italy; E-mail: enrico.oliva@unibo.it.

supporting the storage and sharing of arguments between participants. In earlier work [8], we presented a conceptual framework for a central co-ordinating entity in an argumentation dialog, called a *Co-Argumentation Artifact (CAA)*, to provide co-ordination services to the participating agents, allowing them to share, store and exchange arguments with one another. Vesting the CAA with its own argumentation capabilities meant that this entity, like the participants, could undertake reasoning across the arguments it stored. For example, the CAA could determine whether a particular argument is acceptable (under a specified semantics of argumentation) with respect to the global knowledge of all the participants. We reprise the CAA framework in Section 3.1.

It is easy to imagine that the CAA could undertake more sophisticated interventions in the dialog, resembling complex, automated tasks of a human mediator. To this end, in this paper we extend our earlier concept of a central co-ordinating artifact to be a dialog artifact (DA), acting as a mediator between the participating agents. We do this, first, by articulating, in Section 2, the possible functions of the mediator artifact; for reasons of space, we do not consider all these functions here. We then present a formalization of some of these mediator artifact functions in Section 4, drawing on recent work in the theory of communications artifacts in multi-agent systems [9,16]. We follow this with a description of a prototype implementation we have undertaken in the TuCSoN coordination infrastructure, in Section 5. Finally, Section 6 concludes the paper.

2. Functions of the Argumentation System

Our model of a multi-agent argumentation system has two types of entities: intelligent dialog agents, and a mediator artifact. Dialog participants, of course, need to be able to generate, evaluate, contest and defend arguments as they interact with one another through dialog. But the mediator artifact also needs this argumentation functionality if it is to find common ground between different participants, or to clarify their differences. For example, if the mediator is to convince two participants that their opposed positions in fact share common assumptions or that one position implies the other, then the mediator artifact may need – in an automated way – to create, present and defend a case to the participants.

Consequently, we describe two conceptual components required variously by the entities in our system: an argumentation component, required by all the participating agents and by the mediator artifact; and a central dialog component, required only by the mediator artifact. We now list the functions of each component.

2.1. Argumentation Component

The Argumentation component is used both by the participating agents and by the mediation artifact. It comprises both a set of arguments, represented in a suitable computational form, and a collection of algorithms deployed over this argument set. The set of arguments may naturally differ from one agent to another. The computational model is composed of several modules, each module providing a specific set of constraints resulting from the analysis of the input argument set.

Argument base module contains all argument of the domain

Argumentation consistency check module verifies monotonicity of argument composition

Contrary module finds predicates that are in contrast

Argument set module makes argument division in sets be based on a given semantic

Prolog meta-interpreter works over argument set

2.2. Central Dialog Artifact

The mediator artifact requires some basic functionality to support the exchange of arguments in a dialog between the participants. This basic functionality includes:

1. Storage of the dialog protocol (e.g. in a library of such protocols)
2. Storage of the specifications of the dialog protocol
3. Storage of the complete history of a dialog as it proceeds
4. The ability to refuse to allow agent utterances which do conform to the current protocol in use
5. The ability to suggest next moves which are legal according to the current protocol in use in a dialog
6. The ability to receive and store confidential information from the participating agents, such as their preferences in a negotiation. The mediator could then aggregate such information (across multiple agents), and/or seek to identify and reconcile differences.

Also, the central artifact could act a sophisticated mediator of the discussion, by providing in an automated way the following services:

1. Seeking to resolve any disputes over the rules of the protocol
2. Providing give rewards or penalties to agents for breaking the protocol rules
3. Having the power to admit or to expel agents to/from the dialog
4. Suggesting a new protocol, when needed.
5. Supporting multiple simultaneous bilateral interactions.
6. Assigning roles, rights and responsibilities to agents at run-time, as, for example, in an action protocol, assigning the role of winner to a particular agent near the end of the interaction.
7. Identifying conflicts and inconsistencies between commitments made by agents in a dialog, for example, if an agent commits to sell a car it is also trying to purchase.
8. Identifying agent utterances which are not relevant to the current state of the dialog, and refusing to permit these to be made.
9. Providing automated alerts to inform agents that dialogs on particular topics are about to start, or to end, or that particular commitments have just been made.
10. Combining different dialogs on the same topic.

More advanced functions of the mediator artifact could also include:

1. Annotation of protocols with their properties, for protocols stored in the protocol library, for instance, the possible outcomes of a protocol, its computational complexity, and so on.
2. Storing the outcomes of past dialogs, for example, the commitments remaining at the end of the dialog.

3. Tracking agent commitments across multiple dialogs.
4. Using previous dialogs to create an independent assessment of the reputation of participating agents.
5. Storage of the entire history of past dialogs. These may be required for regulatory or legal reasons, e.g., in stock market transactions.

In this paper, we present a formalization of the Dialog Artifact which conceptually supports the basic functionalities listed above.

3. Argumentation and Dialog: Formal Definitions

In our earlier work [8], we introduced an argumentation system and an artifact abstraction to support the co-ordination functions in a multi-agent dialog. We now extend that framework by formally defining the components, drawing on the theory of organizations and roles in multi-agent systems of Omicini and colleagues [9] and the formal language used to define an agent interaction protocol, in the work of Viroli and colleagues [16]. In our approach we describe a dialog in term of a labeled process algebra, where labels denote roles, as in [9], and the process algebra specifies the interaction protocol, as in [16].

We assume that the interaction is between a finite number N of intelligent software agents, and that each agent has a range of possible utterances (or actions) at each step in the dialogue (i.e., this is a multi-move protocol). Formally, a multi-agent dialog system for argumentation is composed of two parts: an argumentation system; and a dialog system. The definition of the argumentation system is discussed based on the work in [8], and builds on various earlier argumentation frameworks.

3.1. Argumentation System

Prakken and Vreeswijk [15] observe that an argumentation system is generally composed of five elements (although sometimes implicitly): 1) a logical language; 2) an argument definition; 3) a concept of conflict among arguments; 4) a concept of defeated argument; 5) a concept of argument acceptability. In this section we define an argumentation system as a reference point for our work. We take inspiration from Dung's framework [2], and we also define the structure inside the arguments.

The object language of our system is a first-order language, where Σ contains all well-formed formulae. The symbol $\vdash$ denotes classical inference (different styles will be used like deduction, induction and abduction) $\equiv$ denotes logical equivalence, and $\neg$ or *non* is used for logical negation.

Definition 1 *An argument is a triple* $A = \langle B, I, C\rangle$ *where* $B = \{p_1, \ldots, p_n\} \subseteq \Sigma$ *is a set of* beliefs*,* $I \in \{\vdash_d, \vdash_i, \vdash_a\}$ *is the inference style (respectively, deducion, induction, or abduction), and* $C = \{c_1, \ldots, c_n\} \subseteq \Sigma$ *is a set of* conclusions*, such that:*

1. *B is consistent*
2. $B \vdash_I C$
3. *B is minimal, so no subset of B satisfying both 1 and 2 exists*

	Deductive Inference		Inductive Inference
MP	$\frac{\neg A \quad B \rightarrow A}{\neg B}$	θ-su	$\frac{B}{R}$ *where* $R\theta \subseteq B$
MT	$\frac{\neg A \quad B \rightarrow A}{\neg B}$		Abductive Inference
MMP	$\frac{B_1 \quad B_2 \quad B_3 \quad (B_1 \wedge B_2 \wedge B_3) \rightarrow C}{C}$	Ab	$\frac{B \quad A \rightarrow B}{A}$

Table 1. Deductive Inference: (MP) Modus Ponens, (MMP) Multi-Modus Ponens and (MT) Modus Tollens; Inductive and Abductive Inference: (θ-su) θ-subsumption, (Ab) Abductive

The types of inference we consider for deduction, induction and abduction are shown in Table 1. Modus Ponens (MP) is a particular case of Multi-Modus Ponens (MMP) with only one premise. The inference process θ-subsumption derives a general rule R from specific beliefs B, but is not a legal inference in a strict sense.

For defeat of arguments, the definition is not straightforward because there are different type of attack well defined in [15]. Following those definitions, two possible types of attack are 'conclusions against conclusions' – called *rebuttals* – and 'conclusions against beliefs'—called *undercuts*.

Definition 2 *Let* $A_1 = \langle B_1, I_1, C_1 \rangle$ *and* $A_2 = \langle B_2, I_2, C_2 \rangle$ *be two distinct arguments,* A_1 *is an* ***undercut*** *for* A_2 *iff* $\exists h \in C_1$ *such that* $h \equiv \neg b_i$ *where* $b_i \in B_2$

Definition 3 *Let* $A_1 = \langle B_1, I_1, C_1 \rangle$ *and* $A_2 = \langle B_2, I_2, C_2 \rangle$ *be two distinct arguments,* A_1 *is a* ***rebuttal*** *for* A_2 *iff* $\exists h \in C_1$ *such that* $h \equiv \neg c_i$ *where* $c_i \in C_2$

The definitions of acceptability and admissibility used in our framework are those of Dung in [2]. The following definitions are the basic ones in our argumentation system and follow from Dung's framework.

Definition 4 *An argument set S is a* ***conflict free*** *set iff there exist no* $A_i, A_j \in S$ *such that* A_i *attacks* A_j.

Definition 5 *An argument set S* ***defends collectively*** *all its elements if* $\forall$ *argument* $B \notin S$ *where B attacks* $A \in S \quad \exists \; C \in S : C$ *attacks B.*

Definition 6 *An argument set S is a* ***admissible*** *set iff S is conflict free and S defends collectively all its elements.*

Definition 7 *An argument set S is a* ***preferred extension*** *iff S is a maximal set among the admissible set of A.*

An argument is acceptable in the context of preferred semantics if an argument is in some/all preferred extensions (credulous/sceptical acceptance).

Definition 8 *An argument A is* ***credulous acceptable*** *if* $A \in$ *at least one preferred extension.*

Definition 9 *An argument A is* ***sceptical acceptable*** *if* $A \in$ *all preferred extensions.*

Further details, including an implementation and examples of this argumentation framework, can be found in our earlier paper [8].

3.2. Dialog System

In this section we present a novel formalization of a multi-agent dialog system. Our intention is to capture the rules which govern legal utterances, and which govern the effects of utterances on the commitment stores of the dialog. We use a process algebra approach in the style of [16] to represent the possible paths which a dialog may take, and to represent explicitly the operations to and from the commitment stores. We proceed by considering each element of a dialog system in turn: 1) the communication language; 2) the interaction protocol; and 3) the procol semantics.

Because a dialog is a dialectical exchange of arguments, we assume that arguments and counter-arguments are represented and expressed in the formal language defined above in Section 3.1. Agents may exchange arguments, along with facts, with one another in the form of instantiated parameters in their utterances.

3.2.1. Communication Language

The agents need to share a same communication language *CL* in order to exchange information. The role of *CL* as a language used for internal knowledge representation and reasoning is explained in [13]. We let F denote a set of terms representing *facts*, and $\mathscr{A}$ the set terms representing all *arguments* able to be represented in Σ following the definition of an argument given in Definition 1. Our *CL* is defined in order to support all six primary dialogue types as identified by [17]: persuasion, inquiry, negotiation, information seeking, deliberation and eristic.

Definition 10 *Our communication language is a set of locutions L_c. A locution $l \in L_c$ is a term of the form $perf_{name}(Arg_1, \ldots, Arg_n)$ where $perf_{name}$ is a element of the set P of performatives and Arg_x is either a fact or an argument.*

An agent performing a dialog exploiting using the communication language can utter a locution composed of facts and arguments. A fact is represented by syntax `fact(Terms)` and an argument with `argument(B,I,C)`. The definitions to manage attacking and undercutting arguments are provided by the underlying argumentation system given in Definition 1. In the example 1 an agent wants to communicate the classical example of argument like *All men are mortal, Socrates is a man, Socrates is mortal*, and it uses a `Argue` locution with an `argument` parameter.

Example 1 `Argue(argument(name,beliefs([human(Socrates)],[clause(mortal(X),[human(X)])]),infer(MP),conclusions([mortal(Socrates)]))).`

Examples of performatives to support an instance of an *Information Seeking Dialog* could be: `OpenDialog`, `Ask`, `Tell`, `DontTell`, `Provide`, `Argue`, and so on. Further details about this form of dialog and its complete locutions are presented in [1].

3.2.2. Dialog Protocol

In our framework the dialog protocol is a complete description of all possible dialog paths, from the perspective of an external entity observing the dialog between the agents. The protocol indicates the possible paths of a dialog, specifies the source and target of each message, and shows the relationship between utterances and the content of commitment stores. Our approach basically describes the step-by-step behaviour of an external entity acting as a mediator, hence enabling the allowed interactions. Hence, technically, we find it useful to model a dialog in terms of a process algebra with standard composition operators (sequence, parallel, iteration), and whose atomic actions represent either agent utterances, or interactions with the commitment store (writing, reading, or removing a commitment).

On the one hand, Prakken [14] proposes a general definition of locution where a move *m* is denoted by four elements: 1) identifier, 2) speaker (or source), 3) speech act and 4) intended recipient (or target). Following this model, we provides a definition of a speech act, as follows:

Definition 11 *An action A is defined by the syntax* $A ::= s : L_c | s[t_1, \dots, t_n] : L_c$ *where s indicates the source, and* $[t1, \dots, t_n]$ *indicates the (optional) targets of the message.*

On the other, beyond this, we include additional atomic operations *K* over commitment stores—many of theme can actually occur into one argumentation artifact. To this end, the commitment store is viewed as a set of tuples as in [7]: such tuples are manipulated by the commands of the Linda language [4]—`in`, `rd` and `out`.

Definition 12 *A term action K has the syntax* $K ::= in(C,X) | out(C,X) | rd(C,X)$*, where C is a term representing the commitment store identifier, and X is a term representing the commitment.*

Specifically, the commands `in(C,t)`, `rd(C,t)` and `out(C,t)` respectively consumes, reads and puts a tuple `t` in the commitment store `C`. These actions are useful to manage the private or public commitment store in relation to the dialog execution. In particular, they can operate, for example, as action-preconditions in order to restrict or constrain the next action choice, and thus enable only certain future dialog paths. For instance, if at a given time a sub-dialog is guarded by operation `rd(c,commit(a))`, then it is allowed to proceed only if `commit(a)` occurs in the commitment store.

Definition 13 *A protocol P is a composition of action from sets A and K, defined by syntax* $P ::= 0 | A.P | K.P | P + P | (P \parallel P) | !P$ *where the symbols* $., +, \parallel, !$ *denote respectively sequence (action prefix), choice, parallel composition, and infinite replication operators.*

For example, an abstract dialog protocol definition is given by $D := (s : a_1 + s : a_2).(s : a_3 + s : a_4); s : a_5$ where agent *s* is only allowed to execute a sequence of three actions: the sequence composed of a first action consisting of either action a_1 or action a_2, then a second action consisting of either a_3 or a_4, and then a third action comprising a_5. A protocol specifies a set of actions histories, that the agents might execute. As another example of a protocol definition, consider $D := s : a_1 \parallel s : a_1 \parallel s : a_1 \parallel t : a_2 \parallel t : a_3$ where agent *s* can invoke a_1 at most three times, agent *t* can invoke a_2 and a_3 only once, but in whichever order.

To illustrate this framework, we present a specification for an Information-Seeking dialog (*f* is seen as a variable over the content of communication:

Example 2 (Information Seeking Dialog) `c:OpenDialog.`
```
s:OpenDialog.
!(c:Ask(f).
  s:Tell(f).(
     rd(perm(c,f)).
     s:Provide(f).
     s:Argue(perm(c,f),YES,A)
     +
     s:DontTell(f).
     s:Argue(perm(c,f),NO,B).
     c:Argue(perm(c,$\phi$),ADD,A). (
        s:Argue(perm(c,f),NO,B)
        +
        s:Accept(A,perm(c,f)).
        in(Accept(perm(c,f)))
) )  )
```

3.3. Operational Semantics

Following Hamblin [6], we assume that each agent is associated to a knowledge base, accessible to all agents, containing its commitments made in the course of the dialogue. Commitments are understood as statements which the associated agent must support, while they remain in the commitment store, if these statements are questioned or attacked by other agents. We can now use the notion of commitment store and the transition system given in Definition 15 to define an operational semantics for the dialog system. This semantics describes the evolution over time of the dialog state and the states of commitment store (seen as composition of all commitment stores). In essence, the commitment store is the knowledge repository of the dialog as a whole, and it is expressed in our framework as a multiset of terms.

Definition 14 *A commitment store C is a multiset of terms and it is defined by the syntax $C ::= 0|(C|C)|X$ where X is a term.*

Definition 15 *The operational semantics of our dialog system is described by a labelled transition system $\langle S, \rightarrow, I\rangle$, where $S ::= (C)P$ represents the state of dialog system (protocol P running with commitment store C), I is the set of interactions (labels) composed of $i ::= \tau|a$, and $\rightarrow$ is a transition relation of the kind $\rightarrow \subseteq S \times I \times S$.*

As usual, we write $s \xrightarrow{i} s'$ in place of $\langle s, i, s'\rangle \in \leftarrow$, meaning the dialog system moves from state s to s' due to interaction i—either an action a, or an internal step τ (an operation over the commitment store). We introduce a congruence relation $\equiv$, which syntactically equates similar states:

$$0+P \equiv P \qquad P+Q \equiv Q+P \qquad (P+Q)+R \equiv P+(Q+R) \qquad !P \equiv P|!P$$
$$0 \parallel P \qquad P \parallel Q \equiv Q \parallel P \qquad (P \parallel Q) \parallel R \equiv P \parallel (Q \parallel R)$$

We use also notation $t\{x/y\}$, to mean term t after applying the most general substitution between terms x and y—x should be an instance of y, otherwise the substitution notation does not make sense. Finally, we define operational rules that describe the behavior of the dialog system as follows:

$$
\begin{array}{rlr}
(C)out(x).P \xrightarrow{\tau} (C|x)P & & (K-OUT) \\
(C|x)rd(y).P \xrightarrow{\tau} (C|x)P\{x/y\} & & (K-RD) \\
(C|x)in(y).P \xrightarrow{\tau} (C)P\{x/y\} & & (K-IN) \\
(C)(P+Q) \xrightarrow{i} (C')P' & \text{if} \quad (C)P \xrightarrow{i} (C')P' & (OP-SUM) \\
(C)(P|Q) \xrightarrow{i} (C')(P'|Q) & \text{if} \quad (C)P \xrightarrow{i} (C')P' & (OP-PAR) \\
(C)a.P \xrightarrow{a'} (C)P\{a'/a\} & & (ACT)
\end{array}
$$

Rule (K-OUT) provides the semantic of `out` operation, expressing that x term is added to the commitment store C, and process continuation can carry on. Rules (K-RD) and (K-IN) similarly handle operation `rd` and `in`: the use of substitution operator guarantees that the term x in the commitment store is an instance of the term x to be retrieved. Rules (OP-SUM) and (OP-PAR) provide the semantics for choice and parallel operators in the standard way. Finally rule (ACT) expresses that locution a' is executed that is an instance of the allowed one a, and accordingly process continuation P can carry on.

4. The Dialog Artifact

As mentioned above, the A&A meta-model for MAS as discussed in [8] views agents engaged in argumentative communication as making use of an abstraction, called a Co-Argumentation Artifact, to communicate, to exchange information, data and arguments, and to record their public commitments. The current work extends this abstraction by formally defining a Dialog Artifact (DA), able to support and mediate the communication between agents engaged in a dialog under the system defined in Section 3 above.

We define the Dialog Artifact as a triple $DA = \langle DP, CS, IC \rangle$, where: DP is a collection of specifications of dialog protocols; CS is a collection of commitments stores; and IC is a collection of specifications of interaction control (IC). We now define each of these components in turn.

Dialog Protocols

The class DP is a collection of formal specifications of dialog protocols, with each protocol specified using a labeled process algebra, as in Definition 13. Protocols in DP may also be annotated with identifiers and with their properties, such as their termination complexity. When agents engage in dialog using a protocol in the collection DP, they make utterances according to the permitted sequences defined by the protocol specification. Accordingly, the Dialog Artifact is able to verify that utterances proposed by agents in a dialog are valid under the protocol; the DA is also able to use the specification to suggest potential legal utterances to participating agents at each point in the dialog.

Commitment Stores

For any particular collection of agents and any particular dialog they undertake, the collection CS specifies a set of stores representing the private and public Commitment Stores of each participant, together with a central Commitment Store for the dialog as a whole. The Dialog Artifact can support the dialog by holding these stores. The private Commitment Stores are also held by the DA to record confidential information entrusted to

Type	Agent A	All Agents	Mediator Artifact
Private Commitment Store of Agent A	R/W/D	-	R
Public Commitment Store of Agent A	R/W/D	R	R
Central Commitment Store	-	R	R/W/D

Table 2. Commitment Stores - Read (R), write (W) and delete (D) Permissions

it by the participants, such as their private valuations of some scarce resource (in the case of Negotiation dialogs) or arguments based on privileged information (in the case of dialogues over beliefs). Sharing such information with the DA may allow the DA to reason across these stores in a manner which does not reveal the private information of individual agents.

We can classify these various types of stores according to the access permissions (write-, read-, and delete-permissions) holding on each store, as shown in Table 2. The cells of this table indicate the access permissions pertaining to different types of Commitment Stores (the rows of the table), depending on the agent seeking access (the columns of the table). The Dialog Artifact may also store other relevant information, such as the sequence of locutions exchanged in the current dialog, which would be stored in the Central Commitment Store. These stores do not have an algebraic structure but a declarative representation of the contents with a proper classification.

Interaction Control

The third component of the Dialog Artifact, denoted as *IC*, is a collection of specifications for the interaction control. We roughly follow the pattern MVC (Model View Control) where the model is the dialog specification in *DP* the view is the *CS* component with dialog trace and the control is represented by *IC* specification. The control rule of the dialog is represented by the label transition system introduced in previous section, modelling the evolution over time of the agent interaction protocol. Three operators can be used to control the dialog:

$$next^I(s) = \left\{ i : s \xrightarrow{i} s' \right\} \; next^S(s) = \left\{ s' : \exists i, s \xrightarrow{i} s' \right\} \; next^{IS} = \left\{ (i, s') : s \xrightarrow{i} s' \right\}$$

Operator $next^I(s)$ yields the next admissible interactions i from state s. Operator $next^S(s)$ yields the states reachable from s in one step. Operator $next^{IS}$ yields couples (i,s) instead.

The component *IC* realizes the above three operators in order to identify which potential utterances for any agent at any point in the dialog are legal. The basic primitives `in, rd, out` to manage arguments and facts in commitment stores allow the *IC* to identify which constraints on the future course of dialogs are created by the existing commitments. For instance, the *IC* could permit only one utterance in a choice point basing the decision on state of commitment store. Also, it can work with argument set over some advanced structures such as conflict free sets and preferred extensions presented in section 3.1 to determine for instance an argument acceptability.

DA Functionalities

It is straightforward to see that all six basic functionalities of the central dialog artifact listed in Section 2.2 can be performed by a Dialog Artifact defined as a triple $DA = \langle DP, CS, LI \rangle$ as above. The collection *DP* provides the functionalities of items 1 and 2, the storage of protocols and their formal specifications; the Central Commitment Store of the collection *CS* provides storage for the history of a dialog, item 3; similarly, the private Commitment Store components of the collection *CS* provide storage for confidential information communicated from agents to the DA, item 6; the formal specification of a protocol in *DP* (as given by the process algebra formalism we have used above) permits the DA to identify potential utterances which do not conform to the protocol, item 4; and, both the formal protocol specifications in the collection *DP* and the logics of interaction in *IC* permit the DA to suggest possible legal next moves, item 5.

5. TuCSoN Implementation

The technological support to build a *DA* is provided by the TuCSoN coordination infrastructure for MAS introduced in [12]. TuCSoN provides MAS with coordination abstractions called *tuple centres* where agents write, read and consume logic tuples via simple communication operations (`out`, `rd`, `in`, `inp`, `rdp`). As programmable tuple spaces [11], tuple centers can play the role of agent mediator, where coordination rules are expressed in terms of logic specification tuplesof the ReSpecT language—an event driven language over the multi-set of tuples [10]. The tuple center could be considered such as a general support for artefacts. In order to realize the *DA* we exploited TuCSoN a logic tuple centre with programmable behaviour.

Agents utter a locution by means of a `out(move(Dialog, AgentID,Locution))` in the tuple space. The automatic actions executed over the commitment store are represented by the term `cs(ID,out(commit(...)))`—where `out` could be replaced by `in` or `rd` operations. The *CS* class is composed of `commit` tuples that are put in the tuple space as facts and arguments express in logic tuple notation.

The dialog is written in terms of tuples `dialog(name,AList)` where `AList` is the list of actions reifying in tuple form the operators choice `act(A1)+(act(A2))`, parallel `par(A1,A2)` and sequence `A1,A2`. Figure 1 shows a dialog protocol composed by some basic information on dialog state and few steps of the *information seeking dialog* protocol. The tuples that form the *DP* component are: `participant` (number of participants), `dialog` (dialog protocol), `dialogstate` (actual protocol dialog state), and `currentpar` (actual number of participant). In addition, an open dialog session also uses tuple `session(AgentID,infoseek,open)` for each dialog participant.

The key idea of the *IC* implementation is shown in figure 5, where the reactions implementing the control of dialog interaction are presented. In particular, the code implements the dialog state transition after an agent action, the search of next admissible move after an agent request, and also makes it possible the automatic interaction with the commitment store automatically executing `cs` actions. Such mechanisms make it possi-

```
dialogsession(infoseek,close)
participant(infoseek,2)
dialog(infoseek,[act(C,opendialog(C,T)),
       act(T,opendialog(C,T)),act(C,ask(Arg)+
       (act(T,tell(arg1),cs(T,out(commit(arg1)))))])
currentpar(infoseek,0)
```

Figure 1. Example of Dialog State (*DP* component)

```
%reacts from agent next moves request
reaction(rd(nextmoves(Dialog,S)),(
  rd_r(dialogstate(Dialog,S)),
  out_r(findall(S,Dialog))
)).
reaction(out_r(findall(S,Dialog)),(
  in_r(findall(S,Dialog)),
  findall(A,transition(S,A,Q),L),%collect all next legal moves
  out_r(nextmoves(Dialog,L))
)).
```

Figure 2. Implementation of *next*^{I} operator in ReSpecT

ble for a dialog to be driven automatically by the state of the commitment store. Figure 5 shows the ReSpecT implementation of the *next*^{I} operator.

6. Conclusions

In this paper we have proposed a conceptual architecture for a multi-agent dialog system in which participants are assisted by a mediator, called a *Dialog Artifact*. The functions of this mediator are the basic functionalities we have identified as part of a longer list of potential mediation or moderation functions in agent argumentation dialogs. Our Dialog Artifact is an extension of our previous concept of a Co-Argumentation Artifact (CAA), and builds on that earlier work. We also draw on the recent theory of communication artifacts in MAS to formalize the properties of the Dialog Artifact. Our paper also reported on a prototype implementation of these ideas we have undertaken in the programmable Tuple Space framework TuCSoN. In future work, we hope to formalize more of the potential functions of the mediator which we listed above in Section 2. Some of these functions will be straighforward to formalize, for identifying conflicts between commitments or providing automated alerts to agents concerning upcoming dialogs. Others, however, such as run-time assignment of rights and responsibilities to dialog participants will be more challenging.

Acknowledgments We are grateful for partial financial support from the EC's *Information Society Technologies* programme through project ASPIC (IST-FP6-002307).

References

[1] S. Doutre, P. McBurney, and M. Wooldridge. Law-governed Linda as a semantics for agent dialogue protocols. In F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. P. Singh, and M. Wooldridge, editors, *4rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)*, pages 1257–1258, Utrecht, The Netherlands, 25–29 July 2005. ACM Press.

[2] P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77(2):321–358, 1995.

[3] J. Forester. *The Deliberative Practitioner: Encouraging Participatory Planning Processes*. MIT Press, Cambridge, MA, USA, 1999.

[4] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[5] T. F. Gordon and N. Karacapilidis. The Zeno argumentation framework. In *Proceedings of the Sixth International Conference on AI and Law*, pages 10–18, New York, NY, USA, 1997. ACM Press.

[6] C. L. Hamblin. *Fallacies*. Methuen, London, UK, 1970.

[7] P. McBurney and S. Parsons. Posit spaces: a performative theory of e-commerce. In M. Wooldridge J. S. Rosenschein, T. Sandholm and M. Yokoo, editors, *Proceedings of AAMAS 2003*, pages 624–631, New York City, NY, USA, 2003. ACM Press.

[8] E. Oliva, P. McBurney, and A. Omicini. Co-argumentation artifact for agent societies. In I. Rahwan, C. Reed, and S. Parsons, editors, *Proceedings of the Fourth International Workshop on Argumentation in Multi-Agent Systems (ArgMAS 2007)*, pages 115–130, AAMAS 2007, Honolulu, Hawai'i, USA, 2007.

[9] A. Omicini, A. Ricci, and M. Viroli. An algebraic approach for modelling organisation, roles and contexts in MAS. *Applicable Algebra in Engineering, Communication and Computing*, 16(2-3):151–178, August 2005. Special Issue: Process Algebras and Multi-Agent Systems.

[10] Andrea Omicini. Formal ReSpecT in the A&A perspective. In Carlos Canal and Mirko Viroli, editors, *5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06)*, pages 93–115, CONCUR 2006, Bonn, Germany, 31 August 2006. University of Málaga, Spain. Proceedings.

[11] Andrea Omicini and Enrico Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, November 2001.

[12] Andrea Omicini and Franco Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, September 1999.

[13] S. Parsons and P. McBurney. Argumentation-based communication between agents. In M-P. Huget, editor, *Communication in Multiagent Systems*, volume 2650 of *LNCS*, pages 164–178. Springer, Berlin, September 2003.

[14] H. Prakken. Coherence and flexibility in dialogue games for argumentation. *Journal of Logic and Computation*, 15(6):1009–1040, 2005.

[15] H. Prakken and G. Vreeswijk. Logical systems for defeasible argumentation. In D. M. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic, Volume 4*, pages 219–318. Kluwer, Dordrecht, 2002.

[16] M. Viroli, A. Ricci, and A. Omicini. Operating instructions for intelligent agent coordination. *Knowledge Engineering Review*, 21(1):49–69, March 2006.

[17] D. N. Walton and E. C. W. Krabbe. *Commitment in Dialogue: Basic Concepts of Interpersonal Reasoning*. SUNY Press, 1996.

```
transition(cs(Id,A),cs(Id,A),zero).
transition(act(Id,A),act(Id,A),zero).
transition([Act],A,zero):-!,transition(Act,A,zero).
transition([Act,Act2],A,Act2):-!,transition(Act,A,zero).
transition([Act|S],A,S):-transition(Act,A,zero).
transition(S1+S2,A,R1):-transition(S1,A,R1).
transition(S1+S2,A,R2):-transition(S2,A,R2).
%Start reaction
reaction(out(move(Dialog,Id,Act)),(
   in_r(dialogstate(Dialog,S)),
   out_r(transition(S,act(Id,Act),C,Dialog))
)).
reaction(out_r(transition(S,A,S1,Dialog)),(
   transition(S,A,S2), %make the state transition
   in_r(transition(S,A,S1,Dialog)),
   out_r(dialogstate(Dialog,S2)),
   out_r(findall(S2,Dialog))
)).
reaction(out_r(findall(S,Dialog)),(
   in_r(findall(S,Dialog)),
   findall(cs(Id,Commit),transition(S,cs(Id,Commit),Q),L), %collect all next commits
   out_r(nextcsmoves(Dialog,L))
)).
reaction(out_r(nextcsmoves(D,[H|T])),(
   in_r(nextcsmoves(D,[H|T])),
   out_r(excommit(H)), %call execution commit
   out_r(looknext(D,T))
)).
reaction(out_r(looknext(D,[E])),(
   in_r(looknext(D,T)),
   out_r(nextcsmoves(D,T))
)).
reaction(out_r(looknext(D,T)),(
   T==[], in_r(looknext(D,[])),
   in_r(nextcsmoves(D,[]))
)).
%Implementation of K-OUT, K-IN and K-RD
reaction(out_r(excommit(cs(Id,out(A)))),(
   out_r(A), in_r(excommit(cs(Id,out(A)))),
   in_r(dialogstate(Dialog,S)),
   out_r(transition(S,cs(Id,Act),C,Dialog))
)).
reaction(out_r(excommit(cs(Id,in(A)))),(
   in_r(A),  out_r(excommit(cs(Id,in(A)))),
   in_r(dialogstate(Dialog,S)),
   out_r(transition(S,cs(Id,Act),C,Dialog))
)).
reaction(out_r(excommit(cs(Id,rd(A)))),(
   rd_r(A),  in_r(excommit(cs(Id,rd(A)))),
   in_r(dialogstate(Dialog,S)),
   out_r(transition(S,cs(Id,Act),C,Dialog))
)).
```

Figure 3. Control of Interaction: Checking agent legal locution, Making dialog protocol transition and executing automatically *cs* actions are the basic function here implemented in ReSpecT