Towards cooperative argumentation for MAS: an Actor-based approach

Giuseppe Pisano[a], Roberta Calegari[a] and Andrea Omicini[b]

[a]*Alma AI – Alma Mater Research Institute for Human-Centered Artificial Intelligence, Alma Mater Studiorum—Università di Bologna, Italy*

[b]*Dipartimento di Informatica – Scienza e Ingegneria (DISI), Alma Mater Studiorum—Università di Bologna, Italy*

Abstract

The paper discusses the problem of cooperative argumentation in the context of a multi-agent system (MAS), focusing on the computational model. An actor-based model is proposed as a first step towards cooperative argumentation in MAS to tailor distribution issues—illustrating a preliminary fully-distributed version of the argumentation process completely based on message passing.

Keywords

Argumentation, MAS, cooperative argumentation, distributed argumentation process

1. Introduction

One of the critical problems in distributed and collaborative multi-agent systems (MAS) – where agents cooperate towards a goal – is conflict resolution, where argument evaluation often plays a critical role [1]: agents can provide explicit arguments or justifications for their proposals for resolving conflicts by exploiting the so-called negotiation via argumentation, or *cooperative argumentation*, as an effective approach to resolving conflicts. There, the purpose of multi-agent argumentative dialogues is to let agents reach an agreement on *(i)* the evaluation of goals and corresponding actions (or plans); and *(ii)* the adoption of a decentralised strategy for reaching a goal, by allowing agents to refine or revise other agents' goals and defend one's proposals.

Cooperative argumentation is exploited in some real-world multi-agent applications [2]. However, a key problem in such applications is that a widely-acknowledged well-founded computational model of argumentation is currently missing, thus making it difficult to investigate the convergence and scalability of argumentation techniques in highly-distributed environments [1, 2]. To alleviate those difficulties, we present a first version of a message-based distributed argumentation algorithm as the basic pillar of a computational

WOA'21: 22nd Workshop "From Objects to Agents", September 01–03, 2021, Bologna, Italy

g.pisano@unibo.it (G. Pisano); roberta.calegari@unibo.it (R. Calegari); andrea.omicini@unibo.it (A. Omicini)

0000-0003-0230-8212 (G. Pisano); 0000-0003-3794-2942 (R. Calegari); 0000-0002-6655-3869 (A. Omicini)

model for cooperative argumentation in MAS. In this work we ignore issues such as agent autonomy and MAS coordination artefacts, and focus instead on the distribution issues of cooperative argumentation, based on the logic-based agreement framework Arg-tuProlog [3, 4], which enables agent dialogue and defeasible reasoning in MAS. In particular, we focus on the single query evaluation mode of the tool, aimed at evaluating the admissibility of a single statement with no need to build the entire argumentation graph. We propose a preliminary fully-distributed version of the argumentation algorithm, based on message passing, whose focus is on the requirements for a sound distributed evaluation of the argumentation task. For the purpose of this paper, we exploit the actors' paradigm and its main properties—i.e., *a)* fully reactive computational nodes; *b)* communication through message passing.

Accordingly, the paper is structured as follows. Section 2 contains useful preliminary notions in order to understand the content of the paper. Section 3 and Section 4 illustrate the contribution, introducing a distributed computational model enabling the assessment via argumentation of a single argument. In particular, Section 3 first discusses how the argument evaluation algorithm of Arg-tuProlog can be parallelised , then addresses the problem of knowledge manipulation in a decentralised setting. In Section 4 we deliver a complete and coherent model for decentralised reasoning based on the actor model. Finally, Section 5 concludes the work.

2. Preliminaries

2.1. A basic intro to structured argumentation

In the argumentation language, a literal is an atomic proposition or its negation.

Notation 1. *For any literal ϕ, its complement is denoted by $\bar{\phi}$. That is, if ϕ is a proposition p, then $\bar{\phi} = \neg p$, while if ϕ is $\neg p$, then $\bar{\phi}$ is p.*

Literals are brought into relation through rules.

Definition 1 (Rules). *A **defeasible rule** has the form: $\rho : \phi_1, ..., \phi_n \Rightarrow \psi$ with $0 \leq n$, and where*

- *ρ is the unique identifier for r, denoted by $\mathsf{N}(r)$;*
- *each $\phi_1, \ldots \phi_n, \psi$ is a literal;*
- *the set $\{\phi_1, \ldots \phi_n\}$ is denoted by $Antecedent(r)$ and ψ by $Consequent(r)$.*

Defeasible rules – denoted with DefRules – are rules that can be defeated by contrary evidence. Pragmatically, a defeasible rule is used to represent defeasible knowledge, i.e., tentative information that may be used if nothing could be posed against it. For the sake of simplicity, we define non-axiom premises via defeasible rules with empty *Antecedent*. A theory consists of a set of rules.

Definition 2 (Theory). *A **defeasible theory** is a tuple $\langle Rules \rangle$ where $Rules \subseteq DefRules$.*

Arguments are built from defeasible rules. Given a defeasible theory, arguments can be constructed by chaining rules from the theory, as specified in the definition below—*cf.* [5].

Definition 3 (Argument). *An* ***argument*** *A constructed from a defeasible theory $\langle \mathsf{Rules} \rangle$ is a finite construct of the form:*

1. $A : A_1, \ldots A_n \Rightarrow_r \phi$

with $0 \leq n$, where

- *r is the top rule of A, denoted by $\mathsf{TopRule}(A)$;*
- *A is the argument's unique identifier;*
- *$\mathsf{Sub}(A)$ denotes the entire set of subarguments of A, i.e., $\mathsf{Sub}(A) = \mathsf{Sub}(A_1) \cup \ldots \cup \mathsf{Sub}(A_n) \cup \{A\}$;*
- *ϕ is the* conclusion *of the argument, denoted by $\mathsf{Conc}(A)$;*

Arguments can be in conflict, accordingly to two kinds of attack: rebuts and undercutting, here defined as in [5].

Definition 4 (Attack). *An argument A* ***attacks*** *an argument B (i.e., A is an* ***attacker*** *of B) at $B' \in \mathsf{Sub}(B)$ iff A undercuts, rebuts or undermines B (at B'), where:*

- *A undercuts B (at B') iff $\mathsf{Conc}(A) = \neg \mathsf{N}(r)$;*
- *A rebuts B (at B') iff $\mathsf{Conc}(A) = \neg\phi$ and $\mathsf{Conc}(B') = \phi$;*

An argumentation graph can then be defined exploiting arguments and attacks.

Definition 5 (Argumentation graph). *An* ***argumentation graph*** *constructed from a defeasible theory T is a tuple $\langle \mathcal{A}, \rightsquigarrow \rangle$, where $\mathcal{A}$ is the set of all arguments constructed from T, and $\rightsquigarrow$ is the* attack *relation over $\mathcal{A}$.*

Notation 2. *Given an argumentation graph $G = \langle \mathcal{A}, \rightsquigarrow \rangle$, we write $\mathcal{A}_G$, and $\rightsquigarrow_G$ to denote the graph's arguments and attacks respectively.*

Given an argumentation graph, we leverage on labelling semantics [6, 7] to compute the sets of arguments that are accepted or rejected. Accordingly, each argument is associated with one label which is either IN, OUT, or UND, respectively meaning that the argument is either accepted, rejected, or undecided.

Listing 1: Structured argumentation, Arg-tuProlog answer query algorithm for grounded semantic (pseudo-code).

```
AnswerQuery(Goal):
  A_1, ..., A_n = buildSustainingArguments(Goal)
  Res = ∅
  for A in A_1, ..., A_n:
    Res = Res ∪ Evaluate(A, ∅)
  return Res.

Evaluate(A, Chain):
  if(∃ B ∈ Attacker(A): Evaluate(B, A ∪ Chain) = IN)
    return OUT
  if(∃ B ∈ Attacker(A): B ∈ Chain)
    return UND
  if(∃ B ∈Attacker(A): Evaluate(B, A ∪ Chain) = UND)
    return UND
  return IN.
```

2.2. Structured evaluation in Arg-tuProlog

The Arg-tuProlog [3, 4] engine is a logic-based agreement framework enabling defeasible reasoning and agents' conversation, which reifies the structured argumentation model presented above.

With respect to the available argumentation frameworks, Arg-tuProlog includes the *query-based* mode, which allows for single-query evaluation according to the selected semantics[1]. Single-query evaluation is precisely the algorithm we are interested in, given that cooperative argumentation in highly reactive systems is often based on a quick debate on some beliefs – those concerning the decision to be made at that moment – rather than on a complete assessment of all the agents' knowledge—where a shared agreement is not easily achieved.

This feature is accessible in the tool through the predicate

```
answerQuery(+Goal, -Yes, -No, -Und)
```

which requests the evaluation of the given *Goal*, and gets the set of facts matching the goal distributed in the three sets IN, OUT, and UND as a result.

The algorithm used to evaluate a single claim (or query) according to grounded semantic is inspired by the *DeLP* dialectical trees evaluation [8]. Starting from the given `Goal`, arguments $A_1, ..., A_n$ sustaining that claim are constructed and evaluated via the `evaluate` predicate—see Listing 1, where `A` indicates the argument to be evaluated, `Chain` is the chain of recursive arguments looking for attackers, and `B` is an attacker. To assess the $A_1, ..., A_n$ status (acceptability or rejection), three conditions are evaluated—note that the order is important to guarantee the soundness of the algorithm:

(Cond1) if a conflicting argument labelled as IN exists, then A_1 is OUT;

[1] At the time of writing, only grounded semantic is fully implemented

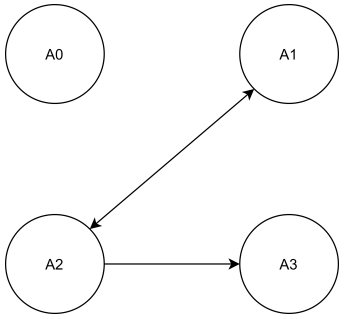

Figure 1: Argumentation graph for arguments from Example 1, in which nodes are arguments and edges are attacks between arguments.

(Cond2) if a cycle in the route from the root to the leaves (`Chain`) exists, then A_1 argument is UND;

(Cond3) if a conflicting argument labelled as UND exists, then also the A_1 argument is UND.

If none of the above conditions is met then the argument can be accepted.

Example 1. *Let us consider the following theory and the corresponding arguments (also depicted in Figure 1 (a))*

$r1$	$:$	$\Rightarrow a$	$A0$	$:$	$\Rightarrow_{r1} a$
$r2$	$: a$	$\Rightarrow b$	$A1$	$: A0$	$\Rightarrow_{r2} b$
$r3$	$:$	$\Rightarrow \neg\, b$	$A2$	$:$	$\Rightarrow_{r3} \neg\, b$
$r4$	$: b$	$\Rightarrow c$	$A3$	$: A1$	$\Rightarrow_{r4} c$

where, according to grounded semantic $A0$ *is* IN *while* $A1$, $A2$ *and* $A3$ *are* UND.

Suppose to require the evaluation of claim b *through the algorithm in Listing 1. First, the arguments sustaining* b *are selected, in this case only* $A1$. *Then the evaluation conditions on* $A1$ *attackers – only* $A2$ *in this case – are assessed. However,* $A2$ *admissibility depends, in turn, on* $A1$. *There is a cycle in the graph (condition (Cond2)), and no other attackers matching condition (Cond1). As a consequence,* $A2$ *is* UND *and thus* $A1$ *(condition (Cond3)). Accordingly, claim* b *is labelled* UND *in the answer query response as expected.*

3. Parallelising arguments evaluation

The first issue when affording computational issues of cooperative argumentation is the parallelisation of the argumentation process. Parallelisation needs to be tackled under two distinct perspective: *a)* the algorithm perspective – i.e., the task perspective – and *b)* the data perspective—i.e., the distributed-knowledge perspective. The algorith perspective aims at parallelising the computational task of the argument evaluation (w.r.t. a given semantics). The action level here is therefore at the algorithmic level, looking for possible sub-tasks to be parallelised within the algorithm itself. The data perspective aims instead at parallelising the data used by the algorithm—i.e., the argumentation defeasible theory.

Listing 2: `Evaluate` predicate with parallel attackers

```
Evaluate(A, Chain):
  if(PARALLEL { ∃ B ∈ Attacker(A): Evaluate(B, A ∪ Chain) = IN })
    return OUT
  if(PARALLEL { ∃ B ∈ Attacker(A): B ∈ Chain })
    return UND
  if(PARALLEL { ∃ B ∈Attacker(A): Evaluate(B, A ∪ Chain) = UND })
    return UND
  return IN
```

The action level here is therefore at the data level, looking for possible data partitioning on which the argumentation process can be run in parallel.

Accordingly, in this section we discuss and address both perspectives, respectively in Subsection 3.1 and Subsection 3.2.

3.1. Task parallelisation

Let us consider the algorithm discussed in Subsection 2.2. The purpose of this section is to analyse the requirements and implications of its parallelisation. Note that the part affected to parallelisation is encapsulated in the `evaluate` predicate, which is why in the following we take into account that predicate only.

The algorithm structure is simple: the argument evaluation leverages the evaluation obtained from its attackers—i.e., the attackers are recursively evaluated using the same algorithm and the result is exploited to determine the state of the target argument. Intuitively, a first point of parallelisation can be found in the search and evaluation of the `Attackers`. Indeed, every condition exploited by the algorithm ((Cond1), (Cond2), and (Cond3)) to evaluate an argument requires one and only one attacker to match the constraint. Those conditions directly suggest an **OR** parallelisation in the search and evaluation of the attackers. We could evaluate the arguments simultaneously under different branches, and the success in one of the branches would lead to the success of the entire search. Listing 2 shows the modified algorithm.

The algorithm exposes another point of parallalisation. As already hinted, the order in the evaluation of the conditions is essential for the soundness of the algorithm—as illustrated by the following example.

Example 2. *Let us consider argument A and its two attackers B and C. Let it be the case in which we know B and C's labelling,* IN *for the former and* UND *for the latter. If we do not respect the order dictated by the algorithm, A's labelling is either* UND *(condition (Cond3)) or* OUT *(condition (Cond1)). Of course, the first result would be in contrast with the original grounded semantic requirements for which every argument having an* IN *attacker should be definitively* OUT*. Conversely, if we respect the evaluation order, A's labelling would be* OUT *in every scenario.*

Although the evaluation order is strict, we can evaluate all the conditions simultane-

Listing 3: `Evaluate` predicate with parallel conditions evaluation

```
Evaluate(A, Chain):
  PARALLEL {
    first = ∃ B ∈ Attacker(A): Evaluate(B, A ∪ Chain) = IN
    second = ∃ B ∈ Attacker(A): B ∈ Chain
    third = ∃ B ∈Attacker(A): Evaluate(B, A ∪ Chain) = UND
  }
  if(first) return OUT
  if(second AND NOT first) return UND
  if(second AND NOT first) return UND
  if(NOT first AND NOT second AND NOT third) return IN
```

Listing 4: `Evaluate` predicate with both parallel conditions evaluation and parallel attackers

```
Evaluate(A, Chain):
  PARALLEL {
    a = PARALLEL { ∃ B ∈ Attacker(A): Evaluate(B, A ∪ Chain) = IN }
    b = PARALLEL { ∃ B ∈ Attacker(A): B ∈ Chain }
    c = PARALLEL { ∃ B ∈ Attacker(A): Evaluate(B, A ∪ Chain) = UND }
  }
  if(a) return OUT
  if(b AND NOT a) return UND
  if(c AND NOT a) return UND
  if(NOT a AND NOT b AND NOT c) return IN
```

ously and consider the ordering only while providing the labelling for the target argument (mixing **AND** and **OR** parallelisation). Listing 3 displays the algorithm modified accordingly. The three conditions are evaluated in parallel, but the result is given accordingly to the defined priorities. If the first condition is met, the argument is labelled as OUT. Conversely, even if the second or the third condition is met, one should first verify that the first condition does not hold. Only then the argument can be labelled as UND.

Listing 4 contains the final version of the algorithm taking into account both points of parallelisation.

3.2. Knowledge-base parallelisation

In the first part of our analysis, we focused on the parallelisation problem from a pure computational perspective—i.e., we tried to understand if we can split the evaluation task into a group of sub-task and then execute them simultaneously. However, there is another perspective to take into account when parallelising: the one concerning the data.

Example 3. *For example, let us consider a job computing the sum and the product of a set of numbers. Using the sub-task approach, we could have two subroutines running in parallel, one computing the sum and the other computing the product of the numbers. However, leveraging the associativity property of sum and multiplication, we can split the*

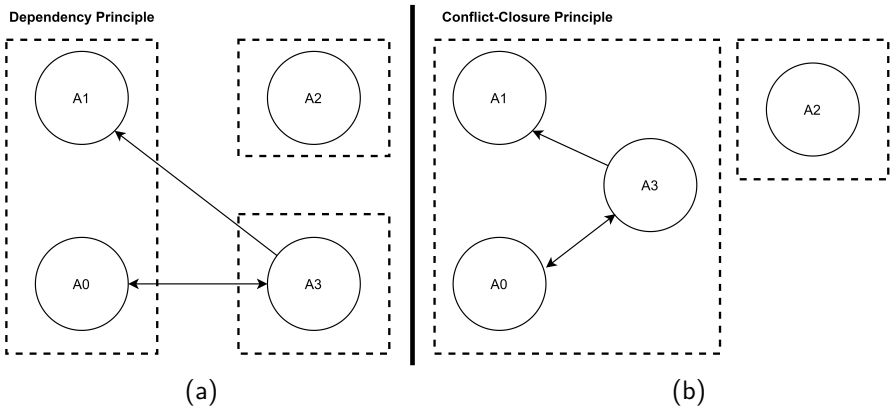

Figure 2: Argumentation graphs and arguments from Example 4 grouped according to dependency (a) and conflict-closure principles (b).

problem into a series of tasks computing both sum and product on a subset of the original data. Then the final result would be the sum and the multiplication of the tasks' results.

Let us suppose to apply the same principle to the argumentation task. We build arguments from a base theory according to the relations illustrated in Subsection 2.1. The logic theory is, for all intents, the input data of our algorithm (argumentation task). Now, the question is if we can effectively split the data into sub-portions to be evaluated in parallel without affecting the global soundness of the original algorithm.

Naive principle. Let us start with a naive solution in which we randomly split the input theory between all the available nodes. Of course, this would lead to evident contradictions.

Example 4. *For instance, let us consider the following theory (left) and its monolithic evaluation according to grounded semantic leading to four arguments (right):*

$r1$	:	$\Rightarrow a$	$A0$	:	$\Rightarrow_{r1} a$
$r2$	$: a$	$\Rightarrow b$	$A1$	$: A0$	$\Rightarrow_{r2} b$
$r3$	:	$\Rightarrow b$	$A2$	:	$\Rightarrow_{r3} b$
$r4$	:	$\Rightarrow \neg a$	$A3$	:	$\Rightarrow_{r4} \neg a$

where $A0$, $A1$ and $A3$ are labelled UND *– since $A0$ and $A3$ attack each other and $A3$ attacks $A1$ – and $A2$ is labelled* IN*. If we leverage a random split, we could have a scenario in which we partition the theory into four parts. Of course, this would lead to a missing argument. Indeed, rules $r1$ and $r2$ are both necessary to conclude $A1$.*

Dependency principle. Now, let us consider a smarter theory splitting principle based on rules dependency—i.e., if two rules can be chained, they must stay together.

Example 5. *Accordingly, if we consider the theory from example 4, we have three subsets of the theory: $r1$ and $r2$, $r3$, $r4$. The evaluation of these three theories would lead to*

the admissibility of all the four arguments, making the result unsatisfactory w.r.t. the original solution (Figure 2 (a)).

Conflict-closure principle. Observing the abstract argumentation graph it is easy to understand that we cannot split rules claiming conflicting knowledge (Figure 2 (b)). Accordingly, we can observe that a safe split can be guaranteed if the graph-connected sub-portions maintain their integrity—i.e., attacker and attacked arguments belong to the same set.

Example 6. *If we apply this principle to the theory in Example 4, we obtain two sub-portions of the original logic theory allowing for a simultaneous evaluation:* $r1$*,* $r2$ *and* $r4$ *(set* KB_a*), and* $r3$ *(set* KB_b*). The application of the argument evaluation algorithm (in Subsection 2.2) to check the admissibility of* b *leads to two results:* b *(*$A1$*) is* UND *(set* KB_a*), and* b *(*$A2$*) is* IN *(set* KB_b*)—coherent with the semantics (Figure 2 (b)).*

Accordingly, in order to guarantee a sound evaluation w.r.t. the original algorithm (Listing 1) the last constraint – conflict-closure – on the theory splitting principle should be considered, yet posing substantial limits on the task parallelisation. However, the limitation can be overcome thanks to the algorithm presented in Subsection 3.1 (Listing 4), in which a parallelisation of the attackers' evaluation is exploited. According to the algorithm, the search and evaluation of the attackers are performed in a distinct subtask (concurrent evaluation). Then, the reification of algorithm decomposition under the data perspective means that we can split the knowledge concerning attacked and attackers into separate sets, since the subtasks to evaluate an attacker require only the knowledge to infer such an attacker—i.e., only the *Dependency* principle must be respected. Indeed, there is no task requiring to know how to build an argument and its attacker – the search is delegated to another process – thus, in case of adopting also the algorithm parallelisation suggested in, the *Conflict-closure* principle can be neglected. Then, w.r.t. data parallelisation, when combined with the algorithm parallelisation, a single subprocess in charge of evaluating an argument needs only the portion of the theory needed to infer the argument itself—i.e., the chainable rules concluding the target claim.

Example 7. *Let us consider again the example 5 based on the rules dependency principle. If we use the algorithm in Listing 4, the evaluation delivers the correct results. For example, when we require the literal* $\neg a$ *(*$A3$*) evaluation, the consequent search for attackers is carried on different processes, having different sub-portions of the original logical theory. Between them, there is the one having in its knowledge base only the rule* $r4$ *– we saw in example 5 that according to the* Dependency *principle the theory has three subsets:* $r1$ *and* $r2$*,* $r3$*,* $r4$ *– and thus concluding literal* a*, causing an undetermined result as expected—there is a cycle in the inference chain. Indeed, it is not needed to have in the same process rules* $r1$*,* $r2$*,* $r4$ *as* Conflict-closure *principle dictates.*

4. The complete model

In the previous section we discussed how the parallelisation of the algorithm concerning the admissibility of a single claim is feasible under multiple perspectives (namely, algorithm and data). In this section, a complete and sound mechanism for the admissibility task in a fully-concurrent way is provided, exploiting the insights from Section 3 and applying them to an actor-based model [9].

In short, the actor model is based on a set of computational entities – the actors – communicating each other through messages. The interaction between actors is the key to computation. Actors are pure reactive entities that only in response to a message can:

- create new actors;
- send messages to other actors;
- change their internal state through a predefined behaviour or change their behaviour.

Actors work in a fully-concurrent way – asynchronous communication and message passing are fundamental to this end – making the actor model extremely suited to concurrent applications and scenarios. We choose this model for its simplicity: it presents very few abstractions making it easy to study both how to model a concurrent system and its properties. The final goal of this research is to provide a sound model for agents' cooperative argumentation in MAS. Since it is an articulated goal, coping with different dimensions – distribution, sociality, coordination, autonomy – we carry on our investigation in two distinct steps: *1)* first, we enable concurrent evaluation of the argumentation algorithms (focusing on distribution), *2)* then, we make available the new computational tool in a MAS context (focusing on sociality, coordination, and autonomy). The actor model, which is the contribution of this work, is a natural choice for the first step of the analysis.

The proposed model embraces both the parallelisation approaches seen in Section 3— i.e., the parallel evaluation of attackers (task parallelisation, Subsection 4.2) and the partitioning of the initial logical theory (data parallelisation, Subsection 4.1).

4.1. Actor-based evaluation: distributing the knowledge base

Let us start with the portion of the model devoted to logic theory distribution according to the *Dependency* principle in Subsection 3.2. Since the actor model focuses on actors and their communication the following design will review the structure and behaviour of the involved actors.

Two main types of actors are conceived in the system: master (Listing 5) and worker (Listing 6). Master actors coordinate the knowledge base distribution phase while the workers hold a portion of the theory. Accordingly, masters' internal state contains a reference to the term to distribute (`elem`) and a list of the feedbacks from the workers' actors on `elem` distribution (`responseList`), while workers' internal state is simply represented by the portion of the theory they manage, identified by `kb`.

Listing 5: Master Actor for knowledge base distribution

```
MasterActor:

  State:
    responseList
    elem

  OnMessage(sender, message):
    if message = AddTheoryMember(term)
      send(ALL, NewTheoryMember(term))
      responseList = []
      elem = term
    if message = Ack(chainingDetails)
      responseList += Ack(chainingDetails)
      evaluateResponses(responseList)
    if message = Nack()
      responseList += Nack()
      evaluateResponses(responseList)

  evaluateResponses(responseList):
    if NOT ALL RESPONSE ARE PRESENT:
      return
    if NO ACK IS PRESENT:
      cretateNewActor(actor)
      send(actor, CreateKnowledgeBase(elem))
    if CONVERGING KB ARE PRESENT:
      selectMergeTarget(target)
      FOR x IN createMergeList(responseList):
        send(x, MergeTheory(target))
```

Messages that masters and workers can exchange are represented by the following types:

- `CreateKnowledgeBase`, the first message sent from the master to a new worker containing its initial knowledge base;
- `NewTheoryMember`, sent from the master to all the available workers, through which the master sends the new theory member to be stored in the workers' kb;
- `Ack`, sent from a worker to its master in response to a `NewTheoryMember` message, confirms the storing of the new rule in the worker's kb;
- `Nack`, sent from a worker to its master in response to a `NewTheoryMember` message, denies the storing of the new rule in the worker's kb;
- `MergeTheory`, sent from the master to a set of workers in the case of overlapping theories, orders the workers to conclude their execution after sending their knowledge bases to a targeted worker;

Listing 6: Worker Actor for knowledge base distribution

```
WorkerActor:

  State:
    kb

  OnMessage(sender, message):
    if message = CreateKnowledgeBase(term)
      kb = term
    if message = NewTheoryMember(term)
      if isChainable(term, kb):
        send(sender, Ack(chainingDetails))
        kb += term
      else:
        send(sender, Nack())
    if message = MergeTheory(target)
      send(target, Kb(kb))
      exit
    if message = Kb(newKb)
      kb += newKb
```

- `Kb`, sent from a worker A to another worker B, contains the knowledge base that B should add to its own.

If the master receive the order to add a new element to the theory (`AddTheoryMember` message), three possible scenarios can be configured:

1. none of the workers contains a compatible knowledge base – i.e., it is not possible to chain the new rule to the knowledge base (`isChainable` returns false) – and consequently, the master creates a new worker containing the portion of the theory (`createNewActor`);

2. one or more workers have a compatible knowledge base (`isChainable` returns true), and they add the element to their kb;

3. a set of workers possess overlapping knowledge bases – i.e. the union set of workers' knowledge bases can be used to create a unique inference chain –, and as a consequence, we merge their knowledge bases and destroy the extra workers (`MergeTheory` message);

Since actors are reactive entities, in order to completely adhere to the actor model the master knowledge base can be changed from outside the actor system—we instruct the master actors to modify the theory through the message `AddTheoryMember`.

Example 8. *Let us consider again the theory in Example 1. Let us assume a single* `MasterActor` *and the following order in the inclusion of the rules in the system:* $r1$,

*r*3, *r*4, *r*2[2]. *As for the first three rules, the behaviour is the same: the* `MasterActor` *issue a* `NewTheoryMember` *and receives back only* `Nack` *messages—since the rules are not chainable. Accordingly, it creates three distinct workers and sends to every one of them a single rule via the* `CreateKnowledgeBase` *message. We now have* `Worker 1`*,* `Worker 2`*, and* `Worker 3` *with respectively r*1*, r*3 *and r*4 *in their knowledge bases. Then the master issues a* `NewTheoryMember` *for r*2*, and both workers 1 and 3 answer with an* `Ack`*. Rule r*2 *is, in fact, the missing link in the inference chain of r*1 *and r*4*. As a consequence, the* `Master` *orders a migration to one of them – let us assume* `Worker 3` *– with the* `MergeTheory` *message.* `Worker 3` *receives the message, sends its kb to* `Worker 1` *via the* `Kb` *message and then stops. At the end of the distribution phase, we have two workers, one containing r*1*, r*2*, and r*4*, and the other only r*3*. The dependency principle is thus respected.*

4.2. Actor-based evaluation: evaluating an argument

Let us proceed with the actor-based evaluation of an argument. For this task, we only need one type of actor—`WorkerActor` in Listing 7. In the final model, we consider workers from Listing 6 and Listing 7 as the same entity. We can evaluate an argument through workers only after they split the logic theory among them according to the mechanism in Subsection 4.1.

Each actor is responsible for evaluating those arguments that can be build using its portion of the theory. When the actor receives an evaluation request, it first checks if attackers exist, w.r.t. its knowledge. Then the actor can: *1)* register the impossibility to evaluate the argument – only if a cycle through the evaluation chain is detected –, *2)* require the attacker arguments evaluation to all the other actors. In the latter case, the actor shall answer the original evaluation request only after receiving a response from others actors. The conditions to match while evaluating an argument are the same as the original algorithm in Listing 1:

- if one counterargument is found admissible, we evaluate the argument as `OUT`;
- if any number of actors votes for the argument undecidability with none voting for the rejection, we mark the argument as `UND`;
- if all the actors agree that no counterarguments can be provided as acceptable, we evaluate the argument as `IN`;

Actors provide their suggestion on the state of the requested argument according to all the labels of their counterarguments.

The messages exchanged among worker actors are:

- `Evaluate`, sent to workers (from outside) to require the evaluation of a claim;
- `Attacker`, sent from a worker to all other workers, requires the evaluation of an argument;

[2]Note that the inclusion order does not affect the final state of the system but only the steps required to converge.

Listing 7: Worker Actor for argument evaluation task

```
WorkerActor:

  State:
    targets

  OnMessage(sender, message):

    if message = Evaluate(claim):
      if buildArgument(claim, arg):
        send(ALL, Attacker(arg, []))

    if message = Attacker(arg, chain):
      if NOT buildAttacker(arg):
        send(sender, In(arg))
      else:
        for attacker IN buildAttacker(arg):
          if attacker IN chain:
            targets += (arg, attaccker, sender, [], Und(arg))
          else:
            send(ALL, Attacker(attacker, chain + [arg]))
            targets += (arg, attaccker, sender, [], None)
        evaluateResponses()

    if message = Und(arg) OR Out(arg) OR In(arg):
      evaluateResponses(arg)

  evaluateResponses(arg):
    for arg, attacker IN targets:
      if ANY OUT:
        targets[attacker] = Out(arg)
      if ANY Und AND NOT ANY OUT:
        targets[attacker] = Und(arg)
      if ALL In:
        targets[attacker] = In(arg)
    if attackersEvaluated(arg):
      sendResponse(sender)
```

- `Und`, `Out`, `In` – sent from a worker to another worker in response to the Attacker message – answering the evaluation request.

Note that the `Evaluate` message comes from outside the actor system and starts the evaluation process. In Listing 7, we omit the details on the collection of the `Evaluate` responses and the return of the final result for the sake of conciseness.

Example 9. *Let us continue the example from 1 and 8 and require the evaluation of claim b. From outside the actor system, we send an* `Evaluate` *message to all the actors.* `Worker 1` *succeeds in building an argument (A1) and sends to all the other* `Workers` *– also* `Worker 1` *is included in the list – an* `Attacker` *message requiring attackers evaluation.* `Worker 1`

answers with an `In` *message – there are no attacking arguments according to its knowledge –, while* `Worker 2` *sends back an* `Und` *response. Indeed,* `Worker 2` *is able to create a valid counterargument (*$A2$*), but a cycle is detected in the inference chain. According to the evaluation algorithm, receiving an* `Und` *and an* `In` *as a response,* `Worker 1` *can finally label* $A1$ *as* UND*.*

5. Related & Conclusions

The work presents a first approach to the problem of cooperative argumentation in the context of a MAS. Starting from the single query evaluation mode of Arg-tuProlog – aimed at evaluating the admissibility of a single statement without the need to build the entire argumentation graph – we introduce the corresponding distributed computational model. We first discuss how the argument evaluation algorithm of Arg-tuProlog can be parallelised, then we deliver a complete model for decentralised reasoning based on the actor model.

Our work follows the insights from the ones in [10] and [11, 12]. The former has been the first proposal of a tool – also based on the tuProlog system – exploiting a dialogical argumentation mechanism—i.e., argumentation is performed across multiple processes proposing arguments and counterarguments. However, attempts to develop it in a completely distributed mode has not been made. Conversely, in [11, 12] the authors directly address the problem of enabling argumentation techniques in MAS. Nonetheless, their technique exploits a centralised evaluation of all the knowledge spread across MAS agents, thus exposing serious problems to the scalability of their approach.

Our work can be extended in various directions. First, we shall provide an implementation of the model in the Arg-tuProlog framework. After that, it will be possible to compare the performances of the monolithic and distributed versions of the algorithm properly addressing a discussion on efficiency and scalability issues.

Then, a well-founded analysis of the model, taking into account its soundness is required.

Moreover, experiments need to be run in a MAS context. There, the open issues are many, e.g., how could agents benefit from this mechanism? Or, how could the use of coordination media impact on the actual model?

Finally, it is worth highlighting that in this work we distribute the knowledge base across actors in order to maximise the scalability of the system. The consequences of using the model in a context where the nodes possess an arbitrary knowledge – as agents in MAS – is still to be inspected.

Acknowledgments

G. Pisano and R. Calegari have been supported by the H2020 ERC Project “CompuLaw” (G.A. 833647). A. Omicini has been supported by CHIST-ERA project “Expectation” (G.A. CHIST-ERA-19-XAI-005).

References

[1] H. Jung, M. Tambe, S. Kulkarni, Argumentation as distributed constraint satisfaction: Applications and results, in: Proceedings of the fifth international conference on Autonomous agents, 2001, pp. 324–331.

[2] Á. Carrera, C. A. Iglesias, A systematic review of argumentation techniques for multi-agent systems research, Artificial Intelligence Review 44 (2015) 509–535.

[3] G. Pisano, R. Calegari, A. Omicini, G. Sartor, Arg-tuprolog: a tuprolog-based argumentation framework, in: Proceedings of the 35th Italian Conference on Computational Logic - CILC 2020, volume 2710 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2020, pp. 51–66. URL: http://ceur-ws.org/Vol-2710/paper4.pdf.

[4] R. Calegari, G. Contissa, G. Pisano, G. Sartor, G. Sartor, Arg-tuProlog: a modular logic argumentation tool for PIL, in: S. Villata, J. Harašta, P. Křemen (Eds.), Legal Knowledge and Information Systems. JURIX 2020: The Thirty-third Annual Conference, volume 334 of *Frontiers in Artificial Intelligence and Applications*, 2020, pp. 265–268. doi:10.3233/FAIA200880.

[5] S. Modgil, H. Prakken, The $ASPIC^+$ framework for structured argumentation: a tutorial, Argument Computation 5 (2014) 31–62. doi:10.1080/19462166.2013.869766.

[6] P. M. Dung, On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games, Artificial Intelligence 77 (1995) 321–358. doi:10.1016/0004-3702(94)00041-X.

[7] P. Baroni, M. Caminada, M. Giacomin, An introduction to argumentation semantics, The Knowledge Engineering Review 26 (2011) 365–410. doi:10.1017/S0269888911000166.

[8] A. J. García, G. R. Simari, Defeasible logic programming: An argumentative approach, Theory and Practice of Logic Programming 4 (2004) 95–138. doi:10.1017/S1471068403001674.

[9] C. Hewitt, P. B. Bishop, R. Steiger, A universal modular ACTOR formalism for artificial intelligence, in: Proceedings of the 3rd International Joint Conference on Artificial Intelligence., William Kaufmann, 1973, pp. 235–245. URL: http://ijcai.org/Proceedings/73/Papers/027B.pdf.

[10] D. Bryant, P. J. Krause, G. Vreeswijk, Argue tuprolog: A lightweight argumentation engine for agent applications, in: Computational Models of Argument, volume 144 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2006, pp. 27–32. URL: http://www.booksonline.iospress.nl/Content/View.aspx?piid=1922.

[11] E. Oliva, P. McBurney, A. Omicini, Co-argumentation artifact for agent societies, in: Argumentation in Multi-Agent Systems, 4th International Workshop, ArgMAS 2007, volume 4946 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 31–46. doi:10.1007/978-3-540-78915-4_3.

[12] E. Oliva, M. Viroli, A. Omicini, P. McBurney, Argumentation and artifact for dialogue support, in: Argumentation in Multi-Agent Systems, Fifth International Workshop, ArgMAS 2008, volume 5384 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 107–121. doi:10.1007/978-3-642-00207-6_7.