# Deriving Dependency Graphs from Abstract Argumentation Frameworks: a Preliminary Report

Stefano Bistarelli<sup>1,\*</sup>, Carlo Taticchi<sup>1</sup>

<sup>1</sup>Department of Mathematics and Computer Science, University of Perugia, Perugia, Italy

#### Abstract

Abstract Argumentation Frameworks (AFs) are used, in the field of Artificial Intelligence, to evaluate the justification state of conflicting information, thus allowing the development of automatic reasoning techniques and systems. Complex argumentative processes, such as decision-making and negotiation, which take place over time (usually marked by shifts in which two or more counterparts exchange their opinions), can be modelled through the Concurrent Language for Argumentation, a formalism for handling concurrent interactions between intelligent agents that use an AF as shared memory. In this paper, we first show how AFs can be interpreted as dependency graphs by exploiting the causal relation between arguments induced by the attacks. Then, we describe a methodology for obtaining a procedure that generates the given AF. Such a procedure allows to dynamically represent dialogues and other forms of interaction that brought to the instantiation of the specific AF. The dependency graph also provides an explanation for the acceptance/rejection of a given argument: the path from a leaf to a root of the underlying graph can be seen as a motivation for the assigned justification state.

#### Keywords

Computational Argumentation, Dependency Graphs, Explainable AI

## 1. Introduction

Argumentation Theory [1] deals with the problem of representing and reasoning with conflicting information. In this context, Argumentation Frameworks constitute the basic tool for studying complex phenomena like the cognitive processes through which humans draw conclusions from a set of premises. The logic underlying the single arguments is neglected in Abstract Argumentation Frameworks (AFs) [2], which can be represented as directed graphs where nodes and edges are interpreted as arguments and attacks, respectively. On the one hand, abstracting the internal structure of arguments entails the possibility of automating tasks such as the selection of acceptable conclusions. On the other hand, AFs only provide information regarding the relations between arguments, and not about the arguments themselves. This limits the understanding we can have of the argumentative process which leads to the instantiation of a given AF, an understanding that is crucial for achieving real-world results [3].

The purpose of this paper is twofold. First, we use the causal relation between the arguments

- D 0000-0001-7411-9678 (S. Bistarelli); 0000-0003-1260-4672 (C. Taticchi)
- © 0 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
- CEUR Workshop Proceedings (CEUR-WS.org)

 $Al^3$  2022 - 6th Workshop on Advances in Argumentation in Artificial Intelligence, November 28, 2022, University of Udine, Udine, Italy

<sup>\*</sup>Corresponding author.

Stefano.bistarelli@unipg.it (S. Bistarelli); carlo.taticch@unipg.it (C. Taticchi)

of an AF induced by attacks (the attacking argument must have been brought forward in the reasoning after the attacked argument) to interpret AFs as dependency graphs, namely structures that describe the dependencies between their elements. In particular, attacking arguments will depend on the attacked ones. Considering this type of dependency, we can derive the order in which the arguments are presented, thus obtaining a potential description of how the instantiation of the AF took place. Then, we show how AFs can be generated through the Concurrent Language for Argumentation (CLA), a language able to model interactions between intelligent agents which communicate and reason through a shared AF. A CLA program is given as a series of actions to perform for building the target AF. In this short paper, we restrict the study to acyclic graphs.

## 2. Preliminaries

We briefly recall the fundamental notions of AFs and argumentation semantics.

**Definition 1 (AFs).** An Abstract Argumentation Framework is a pair  $\langle Arg, R \rangle$  where Arg is a finite set of arguments and R is a binary relation on Arg.

For two arguments  $a, b \in Arg$ , the notation  $(a, b) \in R$  represents an attack directed from a against b. Moreover, we denote as  $a^+$  and  $a^-$  the set of attacks respectively incoming into and outgoing from a. A notion of defence can be used as a criterion for distinguishing acceptable sets of arguments (extensions) in the framework [2, 4].

**Definition 2 (Defended Argument).** Given an  $AFF = \langle Arg, R \rangle$ , an argument  $a \in Arg$  is acceptable with respect to  $D \subseteq Arg$  if and only if  $\forall b \in Arg$  such that  $(b, a) \in R$ ,  $\exists c \in D$  such that  $(c, b) \in R$ , and we say that a is **defended** from D.

The Concurrent Language for Argumentation (CLA) [6, 7, 8] is a framework for modelling concurrent interactions between agents that reason and take decisions through argumentation processes. Agents communicating through CLA constructs share a knowledge base, represented by an AF, to perform reasoning tasks. This shared store can be accessed and updated by the various agents via specifically designed operators that are also able to change the underlying AF. Please refer to [6] for a complete overview of the language.

# 3. AFs as Dependency Graphs

In addition to modelling conflicts in AFs, attacks between arguments also establish a causal relationship between the arguments themselves. Consider an AF  $F = \langle Arg, R \rangle$  with two arguments  $a, b \in Arg$  and the attack relation  $(b, a) \in R$ . We interpret this attack as a conflict between the two arguments, with the further knowledge that b is the argument from which the attack against a starts. This means that b is specifically introduced into the framework to contrast argument a and undermine its validity. In the case a was not present from the beginning, b would have no reason to be part of the AF and this fact allows us to identify a

causal relation between a and b. In our example, the existence of a is preliminary to that of b, so we can say that b depends on a.

Following these considerations, an AF can be interpreted as a **dependency graph**, i.e. a directed graph representing the dependencies of various elements (in our case, arguments). Formally, a dependency graph is a couple D = (S, T) where S is a set of elements and T the transitive reduction of a relation  $R \subseteq S \times S$ . In a dependency graph, one can look for an *evaluation order* respecting the given dependencies. A correct evaluation order is a numbering that orders two elements a and b in such a way that if a is evaluated before b, then a must not depend on b. For example, the element corresponding to argument a in Figure 1 should come before b and c in a correct evaluation order of the represented graph, while b should come before d and after both a and c.



Figure 1: Example of a directed graph that can be interpreted both as an AF and a dependency graph.

Finding a correct evaluation order for a dependency graph amounts to reconstructing the reasoning process that leads to the generation of an AF superseding the same graph. Indeed, AFs represent conflicting information and they can be seen as the instantiation of an argumentative process between intelligent agents. In the real world, such kinds of processes take place over time and can be imagined as a succession of statements made by one or more counterparties, with the various statement referring to (attacking) each other. The beginning of this argumentative process is to be found, therefore, in the leaves of the AF, which, not attacking any other argument, must be the first sentences that started the process.

# 4. A CLA Program for Generating AFs

Using the constructs of CLA, we can realise procedures able to generate AFs by modifying the shared store, which will take the shape of a desired AF. In this preliminary study, we only take into account acyclic AFs, (giving some hints on how to deal with cycles in the concluding section). For example an AF like the one illustrated in Figure 1 can be obtained as a result of the following CLA procedure, which can also be executed through our web interface.<sup>1</sup>

checkw({},{}) -> add({a},{}) -> checkw({a,c},{}) -> add({b},{(b,a),(b,c)}) -> checkw({b},{}) -> add({d},{(d,b)}) -> success || checkw({a},{}) -> add({c},{(c,a)}) -> success

<sup>&</sup>lt;sup>1</sup>Link to CLA web interface: https://conarg.dmi.unipg.it/cla.

The operation checkw(A, R) verifies if the given subsets of arguments A and attacks R belong to the shared store, while add(A, R) inserts A and R into the AF; the execution, then, terminates with a *success* (for a detailed discussion of CLA operations, the reader is referred to [6]). We provide and algorithm for automatically generating AFs through CLA procedures which use check and add operations in accordance with a correct evaluation order for the nodes.

Algorithm 1: AFs generation through CLA		
<b>Data:</b> AF $F = \langle Arg, R \rangle, I \subseteq Arg$ , string S		
Result: CLA program $S$		
1 <b>procedure</b> AFtoProg(F,I,S):		
2	fo	reach $a$ in $I$ do
3		if a is not discovered then
4		mark $a$ as discovered
5		if a is the only discovered node in I then
6		$S = S + \text{``checkw(a^+, \{\}) -> add(\{a\}, \{(a, b) \mid b \in a^+)\}) -> "$
7		else
8		$S = S + "\  \operatorname{checkw}(a^+, \{\}) \twoheadrightarrow \operatorname{add}(\{a\}, \{(a, b) \mid b \in a^+)\}) \twoheadrightarrow "$
9		if $a^-$ is empty then
10		S = S + "success"
11		else
12		$AFtoProg(F,a^-,S)$

The recursive procedure AFtoProg in Algorithm 1 takes in input an AF F, a subset of argument I and a string S which at the end of the execution will contain the desired CLA program. The set I initially contains the leafs of the AF. All the elements in I are processed (line 2) and only if they have not been visited yet, the procedure continues (line 3). Nodes are marked as discovered in line 4 and then are added into the CLA program. If there are no other discovered nodes in I apart from the one being visited (let us call it a), we first check that all the arguments attacked by a have already been added, and then we also add a with all its outgoing attacks (lines 5 - 6). If a is not the only discovered nodes, we build the same CLA process, this time adding the parallel construct at the beginning (lines 7 - 8). In lines 9 - 10 we have the terminal case: whenever we reach a root in the AF, we make the (branch of the) CLA program terminate with success. If the visited node has incoming attacks, we recursively call AFtoProg passing the attackers of a as a parameter (lines 11 - 12).

## 5. Discussion

We studied AFs from the perspective of dependency between arguments. We first showed how causality can be derived from the attack relations, allowing us to interpret an AF as a dependency graph in which arguments depend on those they attack. Second, we resorted to CLA constructs for obtaining a program which generates a desired AF. The advantage of producing an AF in this way lies in the fact that, by reading the trace of the CLA program, it is possible both to reconstruct the process that generated the AF and to obtain an explanation for the justification state assigned to the various arguments.

In this paper, we conducted a preliminary investigation of various aspects related to the similarity between dependency graphs and AFs, and in the future, we plan to deepen this study under several aspects. First of all, we want to remove the constraints on the AF structure that we have imposed in the current work. To generalise our approach also to AFs with cycles, we want to devise a methodology for selecting a set of initial arguments for the procedure of Algorithm 1 even when there are no non-attacked arguments. In particular, we plan to use the Kosaraju-Sharir's algorithm [9] to detect the strongly connected components in the AF and select in a non-deterministic fashion an argument in the cycle that can be reached by any other arguments in the graph. Then, we also want to understand how to obtain optimised programs (for instance, with shorter traces) representing the same argumentative process. In this sense, we could exploit the monoid structure of dependency graphs to identify minimal traces of CLA programs. We also plan to investigate other models for concurrent execution (like Petri nets) in order to study how argumentative processes can be represented and interpreted with the ultimate goal of extracting meaningful information. Finally, it would be interesting to conduct a comparative study with existing approaches for the computation of argumentation semantics in order to understand the connections between the causal relationship we identify and the notion of acceptability.

# References

- C. Dutilh Novaes, Argument and Argumentation, in: E. N. Zalta (Ed.), The Stanford Encyclopedia of Philosophy, Fall 2022 ed., Metaphysics Research Lab, Stanford University, 2022.
- [2] P. M. Dung, On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games, Artif. Intell. 77 (1995) 321–358.
- [3] H. Prakken, M. D. Winter, Abstraction in argumentation: Necessary but dangerous, in: COMMA, volume 305 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2018, pp. 85–96.
- [4] P. Baroni, M. Caminada, M. Giacomin, An introduction to argumentation semantics, Knowl. Eng. Rev. 26 (2011) 365–410.
- [5] M. Caminada, On the issue of reinstatement in argumentation, in: JELIA, volume 4160 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 111–123.
- [6] S. Bistarelli, C. Taticchi, A concurrent language for argumentation, in: AI<sup>3</sup>@AI\*IA, volume 2777 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2020, pp. 75–89.
- [7] S. Bistarelli, C. Taticchi, Introducing a tool for concurrent argumentation, in: JELIA, volume 12678 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 18–24.
- [8] S. Bistarelli, C. Taticchi, A concurrent language for modelling arguing agents, Submitted to Argument & Computation (2022).
- [9] M. Sharir, A strong-connectivity algorithm and its applications in data flow analysis, Computers & Mathematics with Applications 7 (1981) 67–72.