Christian Pasero, BSc

Computation of Clustered Argumentation Frameworks via Boolean Satisfiability

MASTER'S THESIS

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Johannes P. Wallner, Ass.Prof. Dipl.-Ing. Dr.techn. BSc.
Institute of Software Technology

Graz, August 18, 2024

Abstract

English abstract of your thesis

Kurzfassung

Deutsche Kurzfassung der Abschlussarbeit

Acknowledgements

Thanks to everyone who made this thesis possible

Contents

List of Figures

List of Tables

List of Acronyms and Symbols

AF	Argumentation Framework
AI	Artificial Intelligence
ASP	Answer Set Programming
cf	Conflict-Free
adm	Admissible
stb	Stable
BFS	Breadth First Search
DFS	Depth First Search

1 Introduction

We all encounter arguments in our lives frequently. When talking to friends, listening to political discussions, or even making decisions in our head. These arguments can get heated and complex since humans have different beliefs and motivations. Finding a common ground or a "correct" conclusion is complicated and sometimes impossible. However, these imperfections are what make us humans. Artificial Intelligence (AI), conversely, needs to act precisely and logically [8]. That is why much research is being done on knowledge representation and reasoning [7, 21].

When observing arguments objectively, we can distinguish between facts and conclusions. A fact represents a specific state in the real world. A conclusion on the other hand is a fact claimed by the logical relation of the promises. The relations are opposing facts (f.e. *the square x is red* and *the square x is blue*), which are contradicting each other. While accepting the correctness of facts is very important, refuting facts is even more critical in an argument.

If a fact or, i.e., an argument a is a counterargument of another argument b, we can say that a attacks b. With this generalization, we can abstract our model with directed graphs. The arguments are represented as nodes, and the attacks as directed edges [10]. With this abstraction, we can define AFs and use them to evaluate conclusions [12]. Most of the time, we do not operate on real-world cases, but on abstract examples. This means, that we do not care about the argument which is represented by a specific node. But drawing a conclusion from an AF can be challenging and tears down to the definition of semantics.

A semantic defines a subset of argument sets that satisfy the semantic-specific rules. Dung already defined different semantics [9] like Conflict-Free (cf), Admissible (adm) and Stable (stb). According to Dungs definitions, a set S is cf if there are no attacks between the arguments in S. The conflict-free set is mainly a building block for the other semantics, which means that the conflict-free set is always a superset of admissible and stable. A stable set, is a conflict-free set, if for every argument, which is not in S, has an attacker which is in S. Finally, an admissible set is a conflict-free set, where each argument in S has a defender in S. The specific rules can be defined via a boolean formula. They can be used to encode the AFs to be solvable with different boolean solvers like Answer Set Programming (ASP) [3] or, as in our case, with a Boolean Satisfiability Solver (SAT-Solver) [1]. Unfortunately, drawing a conclusion from an AF can be challenging, e.g., it can be NP-complete and sometimes even be beyond NP to decide whether an argument is acceptable under a specific argumentation semantics [11]. In fact, the complexity of proofing faithfulness or spuriousness of an AF is $\prod_2^P$ [23]. This means, that to obtain a result, multiple instances or calls of a SAT-Solver need to be invoked.

One of the first papers describing the concept was written by Dung [9] in 1995. Since then, there has been more and more interest in AFs due to the artificial intelligence community [6]. The argumentation systems and semantics have been modified and improved over the years, and another abstraction layer has been added. This specific abstraction layer is called *clustering* and generalizes multiple arguments into one bundled cluster [23]. Clustering is a technique to reduce the number of arguments without changing the conclusion, which in this instance would be the sets produced by a specific semantic. When producing a clustered (*abstract*) AF, which produces the same semantic sets as the non-clustered (*concrete*) AF, the abstract AF is defined to be *faithful*. While each concrete semantic set has a directly mapped abstract semantic set, not every abstract semantic set has to have a directly mapped concrete semantic set. If we create an abstract AF that produces a semantic set that cannot be mapped to a concrete semantic set, we call it *spurious*.

When reducing the amount of arguments with clustering we have to pay attention to not abstract crucial facts, and thus, falsify accepted sets from the concrete AF or accept refuted ones. This would lead to a spurious abstraction and the abstract AF would not represent the concrete AF anymore. To preserve the representation of the concrete AF, we need to show faithfulness.

For instance, let us consider a real-world example like the weather. We can define arguments:

- a: The sky is blue
- b: The atmosphere scatters the sunlights and makes the sky appear blue.
- c: There exist photographs of a blue sky.
- d: Photographs can be fake.
- e: At sunrise the sky appears to be orange.
- f: Observations can alter, depending on the time.

With this knowledge basis, we can create a concrete AF $G = (A, R)$. Where we abstract the arguments into nodes and transform the opposing statement into attacks as shown in 1.1. If we apply another layer of abstraction, we obtain, f.e. the abstract AF $G' = (A', R')$ defined in 1.2. Here we created a single Cluster consisting of the singletons {`a, b, c, e, f`}.

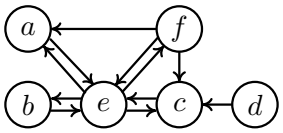

Figure 1.1: AF G concrete

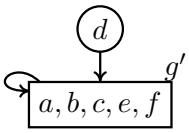

Figure 1.2: AF G' abstract

Now we can compute the sets of the according semantic (cf, adm, stb). To reduce cluttering, we keep this example to the stable semantic. The stable sets of the AF G defined in 1.1 are `stb=[{d, e}, {b, d, f}]`.

By computing the stable semantic sets of the abstract AF G' `stb=[{d}, {d, g'}]`, we can observe that it is spurious due to the extension {`d`}, since it cannot be mapped to one of the concrete stable extensions.

When concretizing the argument {`c`}, we create a new AF $G'' = (A'', R'')$ 1.3, which has the following stable extensions: `stb=[{g', d}]`. This Extension can be mapped to both stable extensions of the concrete AF G, by expanding the cluster `g'` with {`e`} or {`b, f`}. Thus, we created a faithful abstract AF.

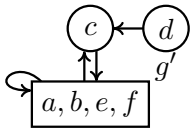

Figure 1.3: AF G"

When producing an AF with multiple layer of abstractions, the concrete problem can be hard to map. To still have an understanding of the structure to some extend, extracting single arguments of the cluster by concretizing them can be helpful. This also allows the user to have a direct impact to the outcome and produce customized faithful AFs.

Creating abstract, faithful AFs can be challenging and is the main focus of this paper. We created one of the first tools [4] to produce an abstract AFs based on a concrete AFs. We cover different setups and usages, including different semantics and base functionalities:

- Generate semantic sets of a concrete- or abstract AF. The sets calculated iteratively or all at once. The covered semantics are cf, adm, and stb, which can be selected throughout the project independently by a parameter in the command line.
- Determine faithfulness or spuriousness by providing two AFs. We provide two approaches, Breadth First Search (BFS) and Depth First Search (DFS), which alter the procedure. While BFS calculates all the semantic sets of the two AFs first and then compares them, DFS calculates iteratively a semantic set of the abstract AF and then verifies it with the concrete AF. The algorithm selection is done via a command line parameter.
- Concretize a set of arguments (i.e., pull out arguments from the cluster) given the concrete AF and an abstract AF (faithful or spurious), and provide faithfulness (by concretizing other arguments not specified in the concretize list as well). The user provides the concretized arguments via a command line parameter.
- ...

TODO: Further contributions
TODO: give pointers to why are non-trivial to obtain
TODO: Choice of methods to obtain results
TODO: How big AFs are still feasible to solve

2 Background

2.1 Argumentation Frameworks

Argumentation frameworks were first formally described by Dung in 1995 [10]. They represent an information state, where various conclusions can be drawen from. An AF $G = (A, R)$ consists of two parameters: a set of arguments A, and a collection of relations R, called attacks which describe the conflicts between the arguments.

They are mostly used in the fields of AI, f.e. in automated reasoning and logic programming [14, 24]. But do also find their applications in other fields like Natural Language Processing [2], Trust and Reputation Systems [17], and even in Game Theory and Strategic Reasoning [20].

AFs are represented by directed graph, where the nodes are an abstraction of the arguments A, and the arrows represent the attacks R. Let us define an AF $G = (A, R)$ with the arguments $\mathtt{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$ and the attacks $\mathtt{R} = [(\mathtt{a}, \mathtt{b}), (\mathtt{b}, \mathtt{b}), (\mathtt{a}, \mathtt{c}), (\mathtt{c}, \mathtt{a}), (\mathtt{c}, \mathtt{d}), (\mathtt{d}, \mathtt{e}), (\mathtt{e}, \mathtt{d})]$.

With the arguments and attacks, we can create a directed graph 2.1.

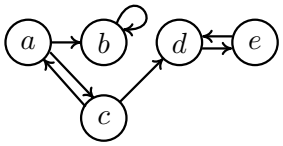

Figure 2.1: AF G

To be able to conclude something, out of an abstract AF, we need to define semantics. A semantic defines a subset of argument sets that satisfy the semantic-specific rules. Dung already defined different semantics [9] like conflict-free, admissible and stable.

conflict-free: According to Dungs definitions, a set S is conflict-free if there are no attacks between the arguments in S. Or, formally:

$$S \in cf(G) \textit{ iff for each } a, b \in S \textit{ we have } (a, b) \notin R$$

The conflict-free set is mainly a building block for the other semantics, which means that the conflict-free set is always a superset of admissible and stable.

In the example 2.1 the conflict-free sets are: $\{\mathtt{a}\}$, $\{\mathtt{c}\}$, $\{\mathtt{d}\}$, $\{\mathtt{e}\}$, $\{\mathtt{a}, \mathtt{d}\}$, $\{\mathtt{a}, \mathtt{e}\}$, $\{\mathtt{c}, \mathtt{e}\}$,

admissible: An admissible set is a conflict-free set, where each argument in S has a defender in S. Or, formally:

$$S \in adm(G) \textit{ iff } S \in cf(G)$$
$$\textit{and if } a \in S \textit{ with } (b, a) \in R,$$
$$\textit{then there is a } c \in G \textit{ with } (c, b) \in R$$

In the example 2.1 the *admissible* sets are: $\{\mathtt{a}\}$, $\{\mathtt{c}\}$, $\{\mathtt{e}\}$, $\{\mathtt{a}, \mathtt{d}\}$, $\{\mathtt{a}, \mathtt{e}\}$, $\{\mathtt{c}, \mathtt{e}\}$

stable: A stable set is a conflict-free set, if for every argument, which is not in S, there exists an attacker in S. Or, formally:

$$S \in stb(G) \textit{ iff } S \in cf(G), b \notin S \textit{ implies}$$
$$\textit{that there is an } a \in S \textit{ with } (a, b) \in R,$$
$$\textit{and if } S \textit{ does not attack an } a \in S \textit{ then } b \notin S$$
$$\textit{whenever } (a, b) \in R \textit{ and } singleton(b)$$

In the example 2.1 the *stable* sets are: $\{\mathtt{a}, \mathtt{d}\}$, $\{\mathtt{a}, \mathtt{e}\}$

The specific semantic rules can also be defined via a boolean formula. Which then can be used to encode the AFs to be solvable with different boolean solvers like ASP [3] or, as in our case, with a SAT-Solver [1]. Unfortunately, drawing a conclusion from an AF can be challenging, e.g., it can be NP-complete and sometimes even be beyond NP to decide whether an argument is acceptable under a specific argumentation semantics [11].

2.2 Clustering of Argumentation Frameworks

When talking about AFs in general, we already have an abstraction layer due to the arguments abstraction. By clustering, we add another layer of abstraction where we combine different arguments into one or multiple so called *clusters*. The arguments which are not clustered are called *singletons*. By definition, a cluster is a single entity (composed of multiple arguments) which can be integrated in an AF to reduce the complexity. While reducing the overall complexity of the AF with clusters, we add a new computation layer: Computing *faithful* clustered AFs. The term *faithful* describes the property of a clustered AF, that every abstract semantic extension can be mapped to a concrete semantic extension. If the clustered AF creates a semantic set which cannot be mapped to a concrete set, we call it *spurious*.

Clustered abstract AFs can also be model with graphs. Where each argument is a node, every attack an arrow and each cluster is represented with a rectangle with every clustered argument inside of it. Let us have a look at an example and define AF $\hat{\mathtt{G}} = (\hat{\mathtt{A}}, \hat{\mathtt{R}})$ with the arguments $\hat{\mathtt{A}} = \{\mathtt{d}, \mathtt{e}, \hat{\mathtt{h}}\}$ and the attacks $[(\hat{\mathtt{h}}, \mathtt{d}), (\mathtt{d}, \mathtt{e}), (\mathtt{e}, \mathtt{d}), (\hat{\mathtt{h}}, \hat{\mathtt{h}})]$. With this definition we can create the directed graph 2.2.

Since clusters can not be treated the exact same way as an argument, we need to refine the semantic definitions. Let us consider a clustered AF $\hat{\mathtt{G}} = \{\hat{\mathtt{A}}, \hat{\mathtt{R}}\}$ and redefine the following semantics:

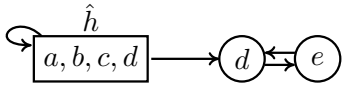

Figure 2.2: AF $\hat{G}$ clustered

conflict-free: A set of arguments is conflict-free, if there is no attack between the singletons of the set. Or, formally, as specified in [23]:

$$\hat{S} \in \hat{cf}(\hat{G}) \textit{ iff for each } \hat{a}, \hat{b} \in \textit{singleton}(\hat{S}) \textit{ we have } (\hat{a}, \hat{b}) \notin \hat{R}.$$

In the example 2.2 the conflict-free sets are: $\{\mathtt{d}\}$, $\{\mathtt{e}\}$, $\{\hat{\mathtt{h}}\}$, $\{\mathtt{e}, \hat{\mathtt{h}}\}$, $\{\mathtt{d}, \hat{\mathtt{h}}\}$

admissible: A set of arguments is admissible, if it is conflict-free and if every singleton which is being attacked, has a defender. Or, formally, as specified in [23]:

$$\begin{gathered}\hat{S} \in \hat{adm}(\hat{G}) \textit{ iff } \hat{S} \in \hat{cf}(\hat{G}) \\ \textit{and if } \hat{a} \in \hat{S} \textit{ with } (\hat{b}, \hat{a}) \in \hat{R} \textit{ with } \textit{singleton}(\hat{a}), \\ \textit{then there is a } \hat{c} \in \hat{G} \textit{ with } (\hat{c}, \hat{b}) \in \hat{R}\end{gathered}$$

In the example 2.2 the admissible sets are: $\{\mathtt{e}\}$, $\{\hat{\mathtt{h}}\}$, $\{\mathtt{e}, \hat{\mathtt{h}}\}$, $\{\mathtt{d}, \hat{\mathtt{h}}\}$

stable: A set of arguments is stable, if it is conflict-free and if an argument is not in the stable set, it implies that an argument in the stable set is attacking it. Furthermore if the stable set is not attacking an argument, then every singleton attacking the argument is not in the stable set. Or, formally, as specified in [23]:

$$\begin{gathered}\hat{S} \in \hat{stb}(\hat{G}) \textit{ iff } \hat{S} \in \hat{cf}(\hat{G}), \hat{b} \notin \hat{S} \textit{ implies} \\ \textit{that there is an } \hat{a} \in \hat{S} \textit{ with } (\hat{a}, \hat{b}) \in \hat{R}, \\ \textit{and if } \hat{S} \textit{ does not attack an } \hat{a} \in \hat{S} \textit{ then } \hat{b} \notin \hat{S} \\ \textit{whenever } (\hat{a}, \hat{b}) \in \hat{R} \textit{ and } \textit{singleton}(\hat{b})\end{gathered}$$

In the example 2.2 the admissible sets are: $\{\mathtt{e}, \hat{\mathtt{h}}\}$, $\{\mathtt{d}, \hat{\mathtt{h}}\}$

Let us have a look at a concrete example to explain faithfulness. The concrete AF $G = (A, R)$ has the following arguments $\mathtt{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$ with these attacks: $\mathtt{R} = [(\mathtt{a}, \mathtt{b})$, $(\mathtt{b}, \mathtt{b})$, $(\mathtt{a}, \mathtt{c})$, $(\mathtt{c}, \mathtt{a})$, $(\mathtt{c}, \mathtt{d})$, $(\mathtt{d}, \mathtt{e})$, $(\mathtt{e}, \mathtt{d})]$.

With this definition we can define the graph G in 2.3.

Now we can group the arguments $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ together into one single cluster $\hat{\mathtt{h}}$. The arguments for the abstract AF $\hat{\mathtt{H}} = (\hat{\mathtt{B}}, \hat{\mathtt{S}})$ would then be $\hat{\mathtt{B}} = \{\mathtt{e}, \hat{\mathtt{h}}\}$ with the according attacks: $\hat{\mathtt{S}} = [(\hat{\mathtt{h}}, \mathtt{e}), (\mathtt{e}, \hat{\mathtt{h}}), (\hat{\mathtt{h}}, \hat{\mathtt{h}})]$

With this defintion we can build the abstract clustered graph $\hat{\mathtt{H}}$ in 2.4

If we compare the stable sets of the concrete AF G (e.g. $\mathtt{stb} = [\{\mathtt{a}, \mathtt{e}\}, \{\mathtt{a}, \mathtt{d}\}]$) with the stable sets of the abstract clustered AF $\hat{H}$ (e.g. $\hat{\mathtt{stb}} = [\{\hat{\mathtt{h}}\}, \{\mathtt{e}\}, \{\mathtt{e}, \hat{\mathtt{h}}\}]$), we see that it is

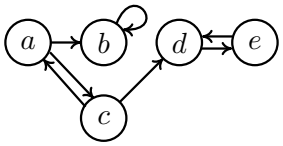

Figure 2.3: AF G

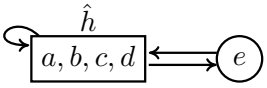

Figure 2.4: AF $\hat{H}$ clustered

spurious due to the stable set $\{\mathtt{e}\}$ which cannot be mapped to one of the concrete stable sets. To create a faithful clustered AF, we need to concretize one or more arguments of the cluster. By concretizing the argument $\{\mathtt{d}\}$, we obtain a new AF $\hat{I} = (\hat{B}, \hat{T})$ with the arguments $\hat{\mathtt{B}} = \{\mathtt{d}, \mathtt{e}, \hat{\mathtt{h}}\}$ and the attacks $\hat{\mathtt{T}} = [(\mathtt{d}, \hat{\mathtt{h}}), (\mathtt{d}, \mathtt{e}), (\mathtt{e}, \mathtt{d}), (\hat{\mathtt{h}}, \hat{\mathtt{h}})]$.

With this defintion we can build the concretized abstract graph $\hat{\mathtt{I}}$ in 2.5

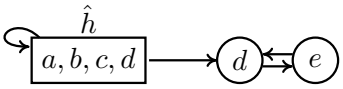

Figure 2.5: AF $\hat{I}$ clustered

Every stable set in 2.5 (e.g. $\{\mathtt{d}, \hat{\mathtt{h}}\}, \{\mathtt{e}, \hat{\mathtt{h}}\}$) can be mapped to one of concrete stable sets of G, which means that the clustered AF $\hat{I}$ is faithful.

2.3 SAT-Solver

A SAT-Solver is used to compute boolean formulas in a rather efficient way. The main purpose is to determine, if a formula is satisfiable (e.g. the variables of the formula can be set to *true* or *false* s.t. the expression evaluates to *true*). If no combination of setting the variables to *true* or *false* s.t. the formula evaluates to *true* is found, we call the boolean expression unsatisfiable. Most of the SAT-Solvers do also provide a model, if a boolean expression is satisfiable.

SAT-Solvers do find there applications in various domains, f.e. in verification and validation of software and hardware [13, 22]. But also in AI and machine learning [15] and even in security [5, 16].

The complexity class of SAT-Solvers lays in NP-complete, and it was the first problem proven to be in in this class. Thus, a lot of other problems could be proven to be in NP-complete due to a reduction to SAT.

Each year further optimizations of the current SAT-Solvers are applied. There are several competitions which are being ran in different classes [18]. Meanwhile, SAT-Solvers are so specialized, that there is no overall best SAT-Solver, but it is dependent on the application field. An overall good performing and easy to implement SAT-Solver, which we also used in this paper is the z3 SAT-Solver [19].

We encoded the semantic rules into boolean formula and used the SAT-Solver to evaluate them. To cover all possibilities of AFs, we generalized the formulas and used

short notation to concatinate the variables. Let us have a look at a concrete example with an abstract clustered AF $\hat{\mathtt{G}} = (\hat{\mathtt{A}}, \hat{\mathtt{R}})$ defined in 2.6.

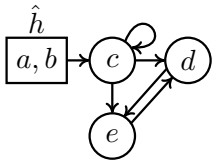

Figure 2.6: AF $\hat{G}$ clustered

For-OR: To concatinate all the singletons of the AF $\hat{G}$, we can use the following notation:

$$\bigvee_{a \in \hat{G}_{SINGLE}} a = c \vee d \vee e$$

For-AND: To concatinate all the singletons of the AF $\hat{G}$, we can use the following notation:

$$\bigwedge_{a \in \hat{G}_{SINGLE}} a = c \wedge d \wedge e$$

For-Attacks: To iterate over the attacks $\hat{R}$ we can extract it from the AF as tuple and address the attacker a and defender b:

$$\bigwedge_{(a,b) \in \hat{R}, a \in \hat{G}_{SINGLE}} (a \vee b) = (c \vee c) \wedge (c \vee d) \wedge (c \vee e) \wedge (e \vee d) \wedge (d \vee e)$$

3 Algorithm

3.1 Concretizing Singletons

Concretizing a list of arguments is done iteratively by deep copying the abstract AF F' to create a new AF F'' and mutating it. The mutation is guided by five steps considering the unchanged abstract AF F' and the concrete AF F. To improve the understanding of each step, we accompany the explanation with the example depicted in 3.3, where we concretize the arguments a and b.

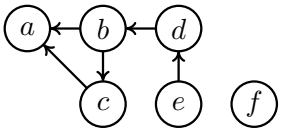

Figure 3.1: Concrete AF F

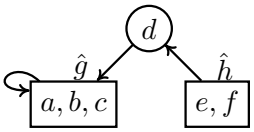

Figure 3.2: Abstract AF F'

Figure 3.3: Example: Concretization of arguments

Step 1: Each argument needing concretization is first removed from the parent cluster and added as a singleton in F''. If an argument is not part of a cluster, we remove it and continue with the filtered list. We do not consider attacks in this step since they depend on the concrete- and abstract AFs.

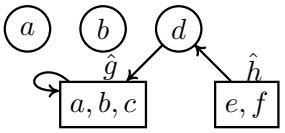

Figure 3.4: Concretized AF F'' after Step 1

Step 2: We add the new attacks from all concretized arguments to the remaining clusters. We must do this after removing the arguments from the clusters because if an argument a attacks argument b in the concrete AF, and b is part of the cluster F' in the abstract AF, by concretizing b, the attack (a, F') would not be persistent anymore.

Step 3: After adding the new attacks, we need to check which attacks from F' are still persistent in F''. If an attack does not persist through the concretization, we remove it

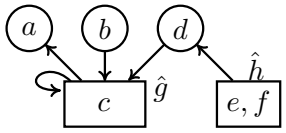

Figure 3.5: Concretized AF F'' after Step 2

in F''. An attack is not persistent anymore; if we remove one of the arguments being attacked or attacked by argument a from the cluster f and no other attack exists, s.t. a is attacked from/attacking an argument within f. Selfattacks of clusters could also change by the concretization of arguments. Therefore, we need to check the clusters from which the arguments are concretized.

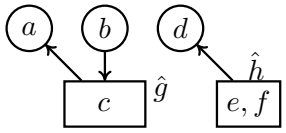

Figure 3.6: Concretized AF F'' after Step 3

Step 4: In this step we add the new attacks between the singletons. Due to the fact, that we copied all the attacks from F', we only have to take into consideration the attacks from or to the concretized singletons. So instead of iterating over all singletons of the AF, we can limit the attack creation to the concretized singletons.

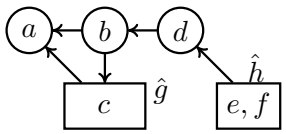

Figure 3.7: Concretized AF F'' after Step 4

Step 5: The last step is to clean up the argumentation framework F'' by removing all empty clusters and mutating the clusters with exactly one singleton to the mentioned singleton.

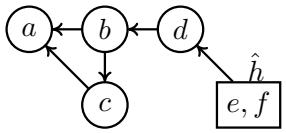

Figure 3.8: Concretized AF F'' after Step 5

The pseudo code is listed in 1.

Algorithm 1 Concretizing Singletons Pseudocode

Require: $A : AF(a_1, r_1)$ ▷ Abstract Clustered AF
Require: $C : AF(a_2, r_2)$ ▷ Abstract Concrete AF
Require: $e : list(Arguments)$ ▷ concretizer list
for e_i in e **do**
  if e_i not in $C \vee e_i$ not in A_C **then** ▷ A_C = Cluster in A
    e.remove(e_i)
  end if
end for
$N \leftarrow A$ ▷ N = Concretized Cluster
N.addSingletons(e) ▷ Step 1
N_C.removeArguments(e)
for e_i in e **do** ▷ Step 2
  for e_i attacks A_c **do**
    $N[e_i].attacks.append(A_c)$
  end for
end for
for r_i in A_r **do** ▷ Step 3
  if r_i not persists in C **then**
    $A_r.remove(r_i)$
  end if
end for
for e_i in e **do** ▷ Step 4
  for e_i attacks C_a **do**
    $N[e_i].attacks.append(C_a)$
  end for
end for
for c_i in N_c **do** ▷ Step 5
  if $c_i.argAmount == 1$ **then**
    $c_i \leftarrow Singleton$
  else if $c_i.argAmount == 0$ **then**
    $N_c.remove(c_i)$
  end if
end for

3.2 Computation of Concretizer List

The concretizer list is a list of sets of clustered arguments. Each set is a unique combination of arguments, which are being concretized to find a faithful AF. All the sets of the concretizer list are attempted iteratively, where the order is dependend on the size of the set. Usually, a minimal deviation of the abstract AF is desired, so small concretizer sets are checked first.

The input to the computation of the concretizer list is a set of the arguments of all the spurious semantic extensions. The size and computation intensity of the concretizer list is highly dependent on the amount of attacks, each argument of the input set and its neighbours with depth 2 have. This is also the critical part of the faithful AF computation and makes some AFs infeasible to solve.

Let us have a look at an example to demonstrate how the concretizer list is computed. The concrete AF G is defined in 3.9 and the according abstract AF $\hat{G}$ in 3.10.

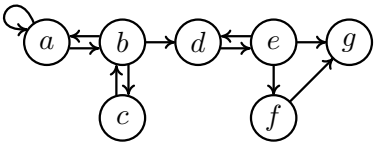

Figure 3.9: Concrete AF G

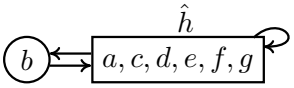

Figure 3.10: Abstract AF $\hat{G}$

If we have a look at the stable extensions of the concrete AF G, e.g. $\texttt{stb} = [\{\texttt{b}, \texttt{c}\}]$ and at the stable extensions of the abstract AF $\hat{G}$, e.g. $\texttt{stb} = [\{\texttt{b}, \hat{\texttt{h}}\}, \{\hat{\texttt{h}}\}, \{\texttt{b}\}]$, we can see that stable extensions $\{\hat{\texttt{h}}\}$ and $\{\texttt{b}\}$ are spurious. The input to the concretizer list computation is a collection of the arguments of all the spurious sets, which in this case is $\{\texttt{b}, \hat{\texttt{h}}\}$.

The first step is to filter out the clusters of the input, since clusters are not present in the concrete AF and therefore do not attack any singletons and are not being attacked. So we reduce the concretizer list from $\{\texttt{b}, \hat{\texttt{h}}\}$ to $\{\texttt{b}\}$.

Next, we have a look at the neighbouring arguments of the current concretizer list. Neighbours in this context are arguments which attack, or are being attacked by an argument. The depth defines how many arguments are between the attacks. A depth of 0 is the actual argument, a depth of 1 represents the direct attacker of the argument and the direct arguments, which are being attacked by the argument. A depth 2 argument is an argument, which has some attack relation (e.g. attacks the argument or is attacked by the argument) with a depth 1 argument.

We used a search depth of 2 in our implementation. So when having a look at our example, we take the defender of depth 1 and 2, in 3.11 depicted in yellow and the attacker with the same depth, depicted in blue. Some arguments can have multiple depths (f.e. argument c. It is a direct attacker of the argument b with depth 0, but also a direct attacker of the argument c with depth 1), than the lower depth is chosen as the representative.

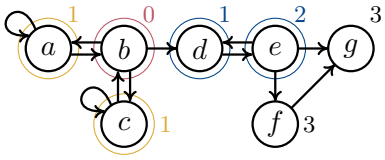

Figure 3.11: Singletons depth with b as viewpoint

Now the concretizer list is expanded with all the possible combinations of the neighbours. The neighbours of the current example are $\{\texttt{a}, \texttt{c}, \texttt{d}, \texttt{e}\}$. When building the combinations, we create the table in 3.1.

size 1	size 2	size 3	size 4
$\{\texttt{a}\}$	$\{\texttt{a}, \texttt{c}\}$	$\{\texttt{a}, \texttt{c}, \texttt{d}\}$	$\{\texttt{a}, \texttt{c}, \texttt{d}, \texttt{e}\}$
$\{\texttt{c}\}$	$\{\texttt{a}, \texttt{d}\}$	$\{\texttt{a}, \texttt{c}, \texttt{e}\}$	
$\{\texttt{d}\}$	$\{\texttt{a}, \texttt{e}\}$	$\{\texttt{a}, \texttt{d}, \texttt{e}\}$	
$\{\texttt{e}\}$	$\{\texttt{c}, \texttt{d}\}$	$\{\texttt{c}, \texttt{d}, \texttt{e}\}$	
	$\{\texttt{c}, \texttt{e}\}$		
	$\{\texttt{d}, \texttt{e}\}$		

Table 3.1: Combinations of $\{\texttt{a}, \texttt{c}, \texttt{d}, \texttt{e}\}$

The combination table grows exponentially to the base of 2. Therefore, the size of the neighbours is crucial. If we have to many neighbours, the computation would need to much memory and turns infeasible to compute.

If the user has provided arguments which have to be concretized as program argument, we add them to each combination set. After adding them, we filter for duplicates to keep the concretizer list size to a minimum.

Next, we need to filter out the arguments, which are not in clusters, since singletons are already concrete. This filtering could lead to some duplicates again, which we need to remove once again to minimize the memory consumption and reduce the amount of faitfhul checks.

Finally, we sort the list by the set size and return it. In the current example we would return the whole table, because no concretizer arguments were provided by the user. So the concretizer list would be $[\{\texttt{a}\}\ \{\texttt{b}\}\ \{\texttt{c}\}\ \{\texttt{d}\}\ \{\texttt{a}, \texttt{c}\}\ \{\texttt{a}, \texttt{d}\}\ \{\texttt{a}, \texttt{e}\}\ \{\texttt{c}, \texttt{d}\}\ \{\texttt{c}, \texttt{e}\}\ \{\texttt{d}, \texttt{e}\}\ \{\texttt{a}, \texttt{c}, \texttt{d}\}\ \{\texttt{a}, \texttt{c}, \texttt{e}\}\ \{\texttt{a}, \texttt{d}, \texttt{e}\}\ \{\texttt{c}, \texttt{d}, \texttt{e}\}\ \{\texttt{a}, \texttt{c}, \texttt{d}, \texttt{e}\}]$.

When concretizing the list, we find a set which leads to a faithful AF: $\{\texttt{a}, \texttt{c}, \texttt{d}\}$ depicted in 3.12. The pseudo code of the computation is listed in 2

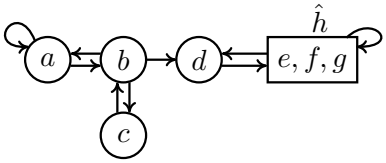

Figure 3.12: Faithful AF $\hat{G}'$

3.3 Algorithmic Approach to Compute Faifthul Clusterings

TODO: Concretize singletons of clustered AF algorithm

3.4 Heuristics and Refinements

TODO: Define every Heuristic and refinement we used for each semantic

Algorithm 2 Computation of Concretizer list Algorithm

Require: $G : AF(a_1, r_1)$ ▷ Concrete AF
Require: $\hat{G} : AF(a_2, r_2)$ ▷ Abstract AF
Require: $s : list(Arguments)$ ▷ spurious arguments
Require: $ca : list(Arguments)$ ▷ Concretize arguments parameter

for s_i in s **do**
  if s_i in $\hat{G}$ is cluster **then**
    s.remove(s_i)
  end if
end for
$N \leftarrow []$ ▷ N = list of neighbours
for s_i in s **do** ▷ Get neighbours
  for $n(1)_i$ attacks s_i **do** ▷ depth 1 attacker
    for $n(2)_i$ attacks $n(1)_i$ **do** ▷ depth 2 attacker
      $N.append(n(2)_i)$
    end for
    for $n(2)_i$ defends $n(1)_i$ **do** ▷ depth 2 defender
      $N.append(n(2)_i)$
    end for
  end for
  for $n(1)_i$ defends s_i **do** ▷ depth 1 defender
    for $n(2)_i$ attacks $n(1)_i$ **do** ▷ depth 2 attacker
      $N.append(n(2)_i)$
    end for
    for $n(2)_i$ defends $n(1)_i$ **do** ▷ depth 2 defender
      $N.append(n(2)_i)$
    end for
  end for
end for
$C \leftarrow$ combinations of N with $range(1, len(N) - 1)$ ▷ Combination List
for ca_i in ca **do** ▷ Parameter Arguments to be concretized
  for c_i in C **do**
    $c_i.append(ca_i)$
  end for
end for
$C.deduplicate()$
for s_i in C **do** ▷ Remove clusters
  for a_i in s_i **do**
    if $\hat{G}[a_i]$ is cluster **then**
      $s_i.remove(a_i)$
    end if
  end for
end for

4 Implementation

4.1 Creating AFs

TODO: Explain AF creation algorithms (Random + Grid-Based)

4.2 BFS and DFS Approach

TODO: BFS and DFS approach in current research + when BFS is better than DFS

4.3 Generating Semantic Sets

TODO: Semantic sets generation algorithm

4.4 Faithful/Spurious Determination

TODO: Determine faithful/spurious algorithm

5 Related Works

6 Conclusion

Bibliography

[1] Leila Amgoud and Caroline Devred. "Argumentation frameworks as constraint satisfaction problems." In: *Ann. Math. Artif. Intell.* 69.1 (2013), pp. 131–148. DOI: 10.1007/S10472-013-9343-0. URL: https://doi.org/10.1007/s10472-013-9343-0.

[2] Elena Cabrio et al. "Natural Language Argumentation: Mining, Processing, and Reasoning over Textual Arguments (Dagstuhl Seminar 16161)." In: *Dagstuhl Reports* 6.4 (2016). Ed. by Elena Cabrio et al., pp. 80–109. ISSN: 2192-5283. DOI: 10.4230/DagRep.6.4.80. URL: https://drops.dagstuhl.de/entities/document/10.4230/DagRep.6.4.80.

[3] Günther Charwat, Johannes Peter Wallner, and Stefan Woltran. "Utilizing ASP for Generating and Visualizing Argumentation Frameworks." In: *CoRR* abs/1301.1388 (2013). arXiv: 1301.1388. URL: http://arxiv.org/abs/1301.1388.

[4] Pasero Christian. *argumentation-framework-clustering*. https://github.com/p4s3r0/argumentation-framework-clustering. 2024.

[5] Pasero Christian. *hashfunction-sat-solver-attacks*. https://github.com/p4s3r0/hashfunction-sat-solver-attacks.git. 2022.

[6] Oana Cocarascu et al. "Mining Property-driven Graphical Explanations for Data-centric AI from Argumentation Frameworks." In: *Human-Like Machine Intelligence*. Ed. by Stephen H. Muggleton and Nicholas Chater. Oxford University Press, 2022, pp. 93–113. DOI: 10.1093/OSO/9780198862536.003.0005. URL: https://doi.org/10.1093/oso/9780198862536.003.0005.

[7] James P. Delgrande et al. "Current and Future Challenges in Knowledge Representation and Reasoning (Dagstuhl Perspectives Workshop 22282)." In: *Dagstuhl Manifestos* 10.1 (2024), pp. 1–61. DOI: 10.4230/DAGMAN.10.1.1. URL: https://doi.org/10.4230/DagMan.10.1.1.

[8] Emmanuelle Dietz, Antonis C. Kakas, and Loizos Michael. "Editorial: Computational argumentation: a foundation for human-centric AI." In: *Frontiers Artif. Intell.* 7 (2024). DOI: 10.3389/FRAI.2024.1382426. URL: https://doi.org/10.3389/frai.2024.1382426.

[9] Phan Minh Dung. "On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and N-Person Games." In: *Artificial Intelligence* 77.2 (1995), pp. 321–357. DOI: 10.1016/0004-3702(94)00041-x.

[10] Phan Minh Dung. "On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games." In: *Artificial Intelligence* 77.2 (1995), pp. 321–357. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(94)00041-X. URL: https://www.sciencedirect.com/science/article/pii/000437029400041X.

[11] Wolfgang Dvorák et al. "The complexity landscape of claim-augmented argumentation frameworks." In: *Artif. Intell.* 317 (2023), p. 103873. DOI: 10.1016/J.ARTINT.2023.103873. URL: https://doi.org/10.1016/j.artint.2023.103873.

[12] Hector Geffner. "A Formal Framework for Clausal Modeling and Argumentation." In: *Practical Reasoning, International Conference on Formal and Applied Practical Reasoning, FAPR '96, Bonn, Germany, June 3-7, 1996, Proceedings*. Ed. by Dov M. Gabbay and Hans Jürgen Ohlbach. Vol. 1085. Lecture Notes in Computer Science. Springer, 1996, pp. 208–222. DOI: 10.1007/3-540-61313-7_74. URL: https://doi.org/10.1007/3-540-61313-7%5C_74.

[13] Martin Gogolla. "Towards Model Validation and Verification with SAT Techniques." In: *Algorithms and Applications for Next Generation SAT Solvers, 08.11. - 13.11.2009*. Ed. by Bernd Becker et al. Vol. 09461. Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2009. URL: http://drops.dagstuhl.de/opus/volltexte/2010/2507/.

[14] Governatori et al. "Argumentation Semantics for Defeasible Logic." In: *Journal of Logic and Computation* 14 (Oct. 2004), pp. 675–702. DOI: 10.1093/logcom/14.5.675.

[15] Jia Hui Liang. "Machine Learning for SAT Solvers." PhD thesis. University of Waterloo, Ontario, Canada, 2018. URL: https://hdl.handle.net/10012/14207.

[16] Da Lin et al. "On the construction of quantum circuits for S-boxes with different criteria based on the SAT solver." In: *IACR Cryptol. ePrint Arch.* (2024), p. 565. URL: https://eprint.iacr.org/2024/565.

[17] Teacy W. T. Luke et al. "TRAVOS: Trust and Reputation in the Context of Inaccurate Information Sources." In: *Autonomous Agents and Multi-Agent Systems* (Mar. 2006). DOI: 10.1007/s10458-006-5952-x.

[18] Hans van Maaren and John Franco. *The International SAT Competition Web Page*. https://satcompetition.github.io/. 2002.

[19] Leonardo de Moura and Nikolaj Bjørner. "Z3: an efficient SMT solver." In: vol. 4963. Apr. 2008, pp. 337–340. ISBN: 978-3-540-78799-0. DOI: 10.1007/978-3-540-78800-3_24.

[20] Simon Parsons, Michael Wooldridge, and Leila Amgoud. "An analysis of formal inter-agent dialogues." In: *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 1*. AAMAS '02. Bologna, Italy: Association for Computing Machinery, 2002, pp. 394–401. ISBN: 1581134800. DOI: 10.1145/544741.544835. URL: https://doi.org/10.1145/544741.544835.

[21] Dragos Constantin Popescu and Ioan Dumitrache. "Knowledge representation and reasoning using interconnected uncertain rules for describing workflows in complex systems." In: *Inf. Fusion* 93 (2023), pp. 412–428. DOI: 10.1016/J.INFFUS.2023.01.007. URL: https://doi.org/10.1016/j.inffus.2023.01.007.

[22] Nils Przigoda et al. *Automated Validation & Verification of UML/OCL Models Using Satisfiability Solvers*. Springer, 2018. ISBN: 978-3-319-72813-1. DOI: 10.1007/978-3-319-72814-8. URL: https://doi.org/10.1007/978-3-319-72814-8.

[23] Zeynep G. Saribatur and Johannes Peter Wallner. "Existential Abstraction on Argumentation Frameworks via Clustering." In: *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021, Online event, November 3-12, 2021*. Ed. by Meghyn Bienvenu, Gerhard Lakemeyer, and Esra Erdem. 2021, pp. 549–559. DOI: 10.24963/KR.2021/52. URL: https://doi.org/10.24963/kr.2021/52.

[24] Toni and Francesca. "A tutorial on assumption-based argumentation." In: *Argument and Computation* 5 (Feb. 2014). DOI: 10.1080/19462166.2013.869878.