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 27, 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

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 [9]. To do so, the data needs to be stored and structured in a way, that AI can extract informations from it. That is why much research is being done in that field of knowledge representation and reasoning [8, 22].

Arguments can have many forms [26]. For instance, arguments can be seen as derivations of conclusions, based on assumptions or premises. Such premises can be facts or defeasible assumptions. Relations among arguments are key for driving (automated) argumentative reasoning. A prominent relation between arguments is that of an attack relation, or counter-argument relation. For instance, an argument might attack another. As an example, one argument might conclude that a square is red, while another is concluding that a square is blue. These two arguments are conflicting, and mutually attack each other. Another example would be that an argument is based on a witness statement, while a counter-argument to this one claims that the witness is not truthful, leading to a one-directional attack.

If an argument `a` is a counterargument of another argument `b`, we can say that `a` attacks `b`. With this abstraction, we can abstract our model with directed graphs. The arguments are represented as nodes, and the attacks as directed edges [11]. Now we can define Argumentation Frameworks (AFs) and use them to evaluate conclusions [13]. In many cases, viewing arguments as abstract entities is sufficient to carry out argumentative reasoning. Such reasoning is defined via so-called argumentation semantics, which define criteria which (sets of) arguments are deemed jointly acceptable.

Semantics define a subset of argument sets that have a certain relation to each other. Dung defined different semantics [10] like conflict-free (cf), admissible (adm) and stable (stb). To be precise, conflict-free and admissible are semantical properties but we will treat them as semantics. According to Dung's definitions, a set S is conflict-free if there are no attacks between the arguments in S. The conflict-free set is mainly a building block for the other semantics. A stable set, is a conflict-free set, if 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. A defender is this context means an argument which attacks an attacker of an argument 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 solvers like Answer Set Programming (ASP) [5] or, as in our case, with a Boolean Satisfiability Solver (SAT-Solver) [1].

Since AFs can get very big and complicated, another layer of abstraction can be added. This abstraction layer is called *clustering* and generalizes multiple arguments into one bundled cluster [24]. Clustering of arguments is a technique to reduce the number of arguments and to provide a high-level view of a given AF. Here, clustering means that arguments can be clustered together in clusters (or clustered arguments). In general, as is the case with many abstraction techniques, clustering can change conclusions that can be drawn from an abstracted formalism. A clustering is said to be *faithful* if no erroneous conclusions can be drawn that is not part of the original, non-abstracted, structure. Otherwise, if conclusions can be drawn that are not there on the original structure, we say that these are *spurious* conclusions.

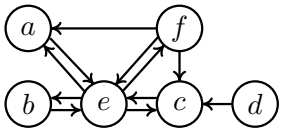
In our case, the semantics of conflict-free, admissibility, and stable semantics were "lifted" to the case with clustered arguments. That is, a clustered (abstracted) version of conflict-free sets, admissible sets, and stable extensions was defined on clustered AFs. These semantics respect the clustering of arguments. Then, e.g., an abstract admissible set is spurious if there is no (concrete, non-abstract) admissible set matching this one in the original AF. If no such spurious sets exist, then the clustered AF is said to be faithful, w.r.t. the original AF.

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

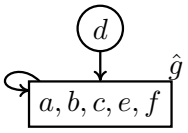
- a: The sky is blue.
- b: The atmosphere scatters the sunlights and makes the sky appear blue.
- c: There exist photographs of a blue sky.
- d: Photographs can be fake.
- e: At sunrise the sky appears to be orange.
- f: Observations can alter, depending on the time.

With this knowledge basis, we can create a concrete AF $\mathtt{G} = (\mathtt{A}, \mathtt{R})$. Where we abstract the arguments into nodes and transform the opposing statement into attacks as shown in Example 1.1. An opposing statement in this context would be for instance the argument `a` (i.d. *The sky is blue*) and `e` (i.d. *At sunrise the sky appears to be orange*). Since both statements can not be true at the same time, they are contradicting, or in other words, attacking each other. If we apply another layer of abstraction, we obtain, e.g. the abstract AF $\hat{\mathtt{G}} = (\hat{\mathtt{A}}, \hat{\mathtt{R}})$ defined in Example 1.2. We call arguments which are clustered, *clustered arguments* and arguments which are not in clusters *singletons*. Here, we created a single cluster consisting of the arguments {`a, b, c, e, f`}.

Now we can compute the sets of the according semantics (cf, adm, stb). The definitions of the semantics are defined in the Chapter 2. To reduce cluttering, we keep this example to the stable semantics. The stable extensions of the AF G are `stb=`{{`d, e`}, {`b, d, f`}}.



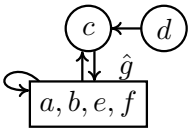
Example 1.1: Concrete AF G



Example 1.2: Abstract AF Ĝ

By computing the stable semantics sets of the abstract AF Ĝ 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 (i.e. removing an argument from the cluster) the argument c, we create a new AF $\hat{\mathtt{G}}' = (\hat{\mathtt{A}}', \hat{\mathtt{R}}')$ depicted in Example 1.3, which has the following stable extensions $\mathtt{stb} = \{\hat{\mathtt{g}}, \mathtt{d}\}$. This extension can be mapped to both stable extensions of the concrete AF G, by expanding the cluster $\hat{\mathtt{g}}$ with {e} or {b, f}. Thus, we created a faithful abstract AF.



Example 1.3: Concretized AF $\hat{\mathtt{G}}'$

When producing an AF with multiple layer of abstractions, we obtain a high-level view of the concrete AF. This simplification has the drawback to lose some details. To still have a deep 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 thesis. 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 [12]. In fact, the complexity of proving faithfulness or spuriousness of an AF is $\prod_2^P$ hard [24]. In practice, this means, that to obtain a result, multiple instances or calls of a SAT-Solver need to be invoked.

We created one of the first tools to produce an abstract AFs based on a concrete AFs. We cover different setups and usages, including different semantics and base functionalities. The main contributions of this thesis are as follows.

- We provide algorithms for computing abstract semantics of a given clustered AF. That is, our algorithms are capable of computing or enumerating all extensions under abstract conflict-free, admissible, and stable semantics.

- Based on our algorithms for computing abstract semantics, we provide algorithmic solutions for checking faithfulness of a given clustered AF. We develop two approaches in this regard: (i) one of based on breadth-first-search (BFS) and (ii) one based on depth-first search (DFS). While the algorithm based on BFS first

calculates all original extensions and abstract extensions of a given AF and clustered AF, respectively, the DFS variant iteratively computes abstract extensions of the clustered AF and verifies (non-)spuriousness of this extension.

- Towards user-interaction, for a given AF and clustered AF, we provide an algorithm for concretization, by which we mean that a user can select arguments inside clusters to be made concrete (singletons). We then refine the clustered AF until faithfulness is reached, since the extraction of the user-defined arguments may result in spurious reasoning.
- We implemented our algorithms in TODO how? and provide the implementations in open-source.
- In an experimental evaluation, $\cdots$ TODO results.
- $\cdots$

TODO: Further contributions
TODO: Choice of methods to obtain results
TODO: How big AFs are still feasible to solve

2 Background

In this chapter we will discuss the background of the thesis. We will start with the basic definition of argumentation frameworks (AFs) in Section 2.1. Next we will treat the clustering of AFs in Section 2.2 and finally we will have a look at SAT-Solver in Section 2.3.

TODO: update when adding sections

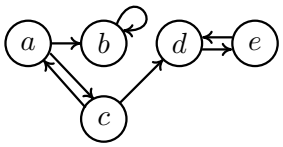
2.1 Argumentation Frameworks

Argumentation frameworks were first formally described by Dung in 1995 [11]. They represent an information state, where various conclusions can be drawn from. An AF $\mathtt{G} = (\mathtt{A}, \mathtt{R})$ consists of two parameters: a set of arguments $\mathtt{A}$, and a collection of relations $\mathtt{R}$, called attacks which describe the conflicts between the arguments.

They are mostly used in the fields of Artificial Intelligence (AI), e.g. in automated reasoning and logic programming [15, 25]. But do also find their applications in other fields like Natural Language Processing [4], Trust and Reputation Systems [18], Legal and Medical Reasoning [2], and even in Game Theory and Strategic Reasoning [21].

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

This AF can be represented as a directed graph as shown in Example 2.1.



Example 2.1: AF $\mathtt{G}$

To be able to conclude something, out of an abstract AF, we need to define semantics. Semantics define a subset of argument sets that satisfy the semantic-specific rules. Dung already defined different semantics [10] like conflict-free, admissible and stable.

Conflict-Free According to Dung's definitions, a set is conflict-free if there are no attacks between the arguments in the conflict-free set. Or, formally:

Definition 1. *Let* $\mathtt{G} = (\mathtt{A}, \mathtt{R})$ *be an AF. Then a set* $S \subseteq A$ *is conflict-free in* G *iff for each* $a, b \in S$ *we have* $(a, b) \notin R$.

A conflict-free set is mainly a building block for the other semantics, which means here that each admissible set or stable extension is conflict-free.

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 According to Dung's definitions, a set is admissible, if each argument in the admissible set has a defender in the admissible set. Or, formally:

Definition 2. *Let* $\mathtt{G} = (\mathtt{A}, \mathtt{R})$ *be an AF. Then a set* $S \subseteq A$ *is admissible in* G*, iff* $S \in cf(G)$ *and if* $a \in S$ *with* $(b, a) \in R$*, then there is a* $c \in S$ *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 According to Dung's definitions, a set is stable, if it is conflict-free, and if for every argument, which is not in the stable extension, there exists an attacker in the stable extension. Or, formally:

Definition 3. *Let* $\mathtt{G} = (\mathtt{A}, \mathtt{R})$ *be an AF. Then a set* $S \subseteq A$ *is stable in* G*. iff* $S \in cf(G), b \notin S$ *implies that there is an* $a \in S$ *with* $(a, b) \in R$*, and if* S *does not attack an* $a \in S$ *then* $b \notin S$ *whenever* $(a, b) \in R$.

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

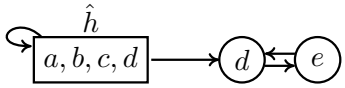
The specific semantics rules can also be defined via a Boolean formula. Which then can be used to encode the AFs to be solvable with different solvers like ASP [5] or, as in our case, with a SAT-Solver [1]. Unfortunately, drawing a conclusion from an AF can be challenging, e.g., it can be NP-complete and sometimes even be beyond NP to decide whether an argument is acceptable (whether an argument can be defended successfully against attacks within the AF) under a specific argumentation semantics [12].

2.2 Clustering of Argumentation Frameworks

When talking about AFs in general, we already have an abstraction layer. 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*. The name is derived from set theory, where singleton refers to a set containing exactly one argument. 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 semantics extension can be mapped to a concrete semantics 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 modelled with graphs. One can represent each singleton argument as a node, attacks as arrows, and each cluster can be represented by 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 $\hat{\mathtt{A}} = \{\mathtt{d}, \mathtt{e}, \hat{\mathtt{h}}\}$, where the cluster $\hat{\mathtt{h}}$ contains the arguments $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ and the attacks being $\{(\hat{\mathtt{h}}, \mathtt{d}), (\mathtt{d}, \mathtt{e}), (\mathtt{e}, \mathtt{d}), (\hat{\mathtt{h}}, \hat{\mathtt{h}})\}$. This AF can be represented as a directed graph as shown in Example 2.2.



Example 2.2: AF $\hat{\mathtt{G}}$ clustered

Since clusters can not be treated the exact same way as an argument, we need to refine the semantics definitions. When referring to the alternative semantics used in clustered AFs, we call them *abstract* semantics (e.g. abstract conflict-free ($\hat{cf}$), abstract admissible ($\hat{adm}$) and abstract stable ($\hat{stb}$)). Let us consider a clustered AF $\hat{\mathtt{G}} = \{\hat{\mathtt{A}}, \hat{\mathtt{R}}\}$ and redefine the semantics.

Conflict-Free A set of arguments is abstractly conflict-free, if there is no attack between the singletons of the set. Or, formally, as specified in [24]:

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

In the Example 2.2 the abstract 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 abstractly admissible, if it is abstractly conflict-free and if every singleton which is being attacked, has a defender. Or, formally, as specified in [24]:

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

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

Stable A set of arguments is abstractly stable, if it is abstractly conflict-free and if an argument is not in the abstractly stable set, it implies that an argument in the abstractly stable set is attacking it. Furthermore if the abstractly stable set is not attacking an argument, then every singleton attacking the argument is not in the abstractly stable set. Or, formally, as specified in [24]:

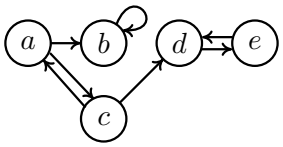
$$\hat{S} \in \hat{stb}(\hat{G}) \textit{ iff } \hat{S} \in \hat{cf}(\hat{G}), \hat{b} \notin \hat{S} \textit{ implies}$$
$$\textit{that there is an } \hat{a} \in \hat{S} \textit{ with } (\hat{a}, \hat{b}) \in \hat{R},$$
$$\textit{and if } \hat{S} \textit{ does not attack an } \hat{a} \in \hat{S} \textit{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

In the Example 2.2 the abstractly stable sets are $\{\mathtt{e}, \hat{\mathtt{h}}\}$, $\{\mathtt{d}, \hat{\mathtt{h}}\}$.

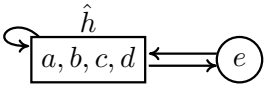
Let us have a look at a concrete example to explain faithfulness. The concrete AF $\mathtt{G} = (\mathtt{A}, \mathtt{R})$ has the following arguments $\mathtt{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$ with these attacks: $\mathtt{R} = \{(\mathtt{a}, \mathtt{b}), (\mathtt{b},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

This AF can be represented as a directed graph shown in Example 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 $\hat{\mathtt{G}} = (\hat{\mathtt{A}}, \hat{\mathtt{R}})$ would then be $\hat{\mathtt{A}} = \{\mathtt{e}, \hat{\mathtt{h}}\}$, where the cluster $\hat{\mathtt{h}}$ is composed of $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ and the according attacks are $\hat{\mathtt{R}} = \{(\hat{\mathtt{h}}, \mathtt{e}), (\mathtt{e}, \hat{\mathtt{h}}), (\hat{\mathtt{h}}, \hat{\mathtt{h}})\}$. The attacks are directly derived from the concrete AF. If an argument is clustered, the cluster inherits the attacks from the argument. The emerging AF can be represented as a directed graph shown in Example 2.4.



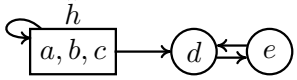
Example 2.3: Concrete AF $\mathtt{G}$



Example 2.4: Abstract AF $\hat{\mathtt{G}}$

If we compare the stable sets of the concrete AF $\mathtt{G}$ (e.g. $\mathtt{stb} = \{\{\mathtt{a}, \mathtt{e}\}, \{\mathtt{a}, \mathtt{d}\}\}$) with the abstractly stable sets of the abstract clustered AF $\hat{\mathtt{G}}$ (e.g. $\hat{\mathtt{stb}} = \{\{\hat{\mathtt{h}}\}, \{\mathtt{e}\}, \{\mathtt{e}, \hat{\mathtt{h}}\}\}$), we see that it is spurious due to the abstractly stable set $\{\mathtt{e}\}$ which cannot be mapped to one of the concrete stable sets. The mapping of semantic extensions with clustered AFs is done the same way as for concrete AFs, except that clusters can mutate to every possible combination of the clustered Arguments. In our example, the cluster $\hat{\mathtt{h}}$ can mutate to $\{\mathtt{a}\}$, $\{\mathtt{b}\}$, $\{\mathtt{c}\}$, $\{\mathtt{d}\}$, $\{\mathtt{a}, \mathtt{b}\}$, $\{\mathtt{a}, \mathtt{c}\}$, $\{\mathtt{a}, \mathtt{d}\}$, $\{\mathtt{b}, \mathtt{c}\}$, $\{\mathtt{b}, \mathtt{d}\}$, $\{\mathtt{c}, \mathtt{d}\}$, $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $\{\mathtt{a}, \mathtt{b}, \mathtt{d}\}$, $\{\mathtt{a}, \mathtt{c}, \mathtt{d}\}$, $\{\mathtt{b}, \mathtt{c}, \mathtt{d}\}$, $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. To create a faithful clustered AF, we can concretize one or more arguments of the cluster. By concretizing the argument $\{\mathtt{d}\}$, we obtain a new AF $\hat{\mathtt{G}}' = (\hat{\mathtt{A}}', \hat{\mathtt{R}}')$ with the arguments $\hat{\mathtt{A}}' = \{\mathtt{d}, \mathtt{e}, \hat{\mathtt{h}}\}$, where the cluster $\hat{\mathtt{h}}$ is composed of $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, and the according attacks are $\hat{\mathtt{R}}' = \{(\mathtt{d}, \hat{\mathtt{h}}), (\mathtt{d}, \mathtt{e}), (\mathtt{e}, \mathtt{d}), (\hat{\mathtt{h}}, \hat{\mathtt{h}})\}$.

With this defintion we can build the concretized abstract graph $\hat{\mathtt{G}}'$ in Example 2.5.



Example 2.5: Concretized AF $\hat{\mathtt{G}}'$

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

2.3 Boolean Satisfiability

A SAT(isfability)-Solver is used to compute Boolean formulas in a rather efficient way [3]. 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 UNSAT(isfiable). 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 [14, 23]. But also in AI and machine learning [16] and even in security [6, 17].

The decision problem of deciding whether a Boolean formula is satisfiable (SAT) is NP-complete, and it was the first problem to be shown to be NP-complete [7]. Subsequently, many other problems were shown to be NP-hard, due to a reduction from SAT.

Each year further optimizations of SAT-solvers are developed. There are several competitions which are being ran in different classes [19]. 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 thesis is the z3 SAT-Solver [20].

3 Algorithm

In this chapter we have a closer look at the algorithms we designed and how they work. We provide an explanation, accompanied with an example and pseudo-code. In Section 3.1 we explain, the concretization of a clustered argument. Next, in Section 3.2 we explain how the concretizer list (a list of clustered arguments which are mutated to singletons) is computed. The approach to compute faithful clustered AFs is described in Section 3.3. And finally we state the heuristics and refinements in Section 3.4.

TODO: Update when adding sections

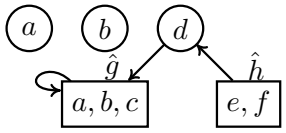
3.1 Concretizing Singletons

When operating on clustered AFs, a crucial mutation is to extract clustered arguments from a cluster and transform it to a singleton. This is called concretizing. When clustering singletons, the cluster inherits the attacks of the argument, concretizing is the inverse operation. This means, that it needs to revert the changes done by the clustering. Concretizing a list of arguments is done iteratively by duplicating the abstract AF $\hat{\mathtt{F}}$ to create a new AF $\hat{\mathtt{F}}'$ and transforming it. The transformation is guided by five steps considering the unchanged abstract AF $\hat{\mathtt{F}}$ and the concrete AF $\mathtt{F}$. To improve the understanding of each step, we accompany the explanation with the example depicted in Example 3.1, where we concretize the arguments `a` and `b`.

(a) Concrete AF $\mathtt{F}$ (b) Abstract AF $\hat{\mathtt{F}}$

Example 3.1: Example: Concretization of arguments

Step 1: Each argument needing concretization is first removed from the parent cluster and added as a singleton in $\hat{\mathtt{F}}'$. If an argument is not part of a cluster, we ignore it. We do not consider attacks in this step since they depend on the concrete- and abstract AFs. The resulting AF is depicted in Example 3.2 and the pseudo-code in Algorithm 1.

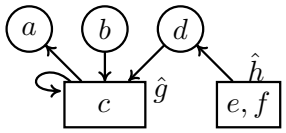


Example 3.2: Concretized AF $\hat{\mathtt{F}}'$ after Step 1

Algorithm 1 Concretizing Singletons Pseudocode Step 1

Require: $A : AF(a_1, r_1)$ ▷ Abstract Clustered AF
Require: $e : list(Arguments)$ ▷ Concretizer List
1: $N \leftarrow A$ ▷ N = Concretized Cluster
2: **for** a_i in e **do**
3: **for** c_i in $A.clusters$ **do**
4: **if** a_i in c_i **then**
5: $c_i.remove(a_i)$
6: **end if**
7: **end for**
8: $N.addSingleton(a_i)$
9: **end for**

Step 2: We add the new attacks from all concretized arguments to the remaining clusters and vice versa. We must do this after removing the arguments from the clusters because if an argument `a` attacks argument `b` in the concrete AF `F`, and `b` is part of the cluster $\hat{\mathtt{g}}$ in the abstract AF $\hat{\mathtt{F}}$, by concretizing `b`, the attack $(\mathtt{a}, \hat{\mathtt{g}})$ would not be present anymore. The resulting AF $\hat{\mathtt{F}}'$ is depicted in Example 3.3 and the pseudo-code in Algorithm 2



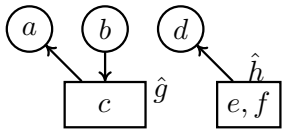
Example 3.3: Concretized AF $\hat{\mathtt{F}}'$ after Step 2

Algorithm 2 Concretizing Singletons Pseudocode Step 2

Require: $C : AF(a_2, r_2)$ ▷ Concrete AF
Require: $e : list(Arguments)$ ▷ Concretizer List
Require: $N : AF(a_3, r_3)$ ▷ Concretized Cluster

```
for e_i in e do
    for att_i in C[e_i].attacks do
        if att_i to cluster c_i then
            N.addAttack((e_i, c_i))
        end if
    end for
    for def_i in C[e_i].defends do
        if def_i from cluster c_i then
            N.addAttack((c_i, e_i))
        end if
    end for
end for
```

Step 3: After adding the new attacks, we need to check which attacks from $\hat{\mathtt{F}}$ are still present in $\hat{\mathtt{F}}'$. If an attack does not persist through the concretization, we remove it in $\hat{\mathtt{F}}'$. An attack is not present anymore; if we remove one of the arguments being attacked or attacked by argument $\mathtt{a}$ from a cluster $\hat{\mathtt{f}}$ and no other attack exists, s.t. $\mathtt{a}$ is attacked from/attacking an argument within $\hat{\mathtt{f}}$. Selfattacks of clusters could also change by the concretization of arguments. Therefore, we need to check the clusters from which the arguments are concretized. The resulting AF is depicted in Example 3.4 and the pseudo-code in Algorithm 3.



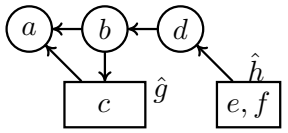
Example 3.4: Concretized AF $\hat{\mathtt{F}}'$ after Step 3

Algorithm 3 Concretizing Singletons Pseudocode Step 3

Require: $A : AF(a_1, r_1)$ ▷ Abstract Clustered AF
Require: $C : AF(a_2, r_2)$ ▷ Concrete AF
Require: $N : AF(a_3, r_3)$ ▷ Concretized Cluster

```
for r_i in A_{r_1} do
    if r_i.defender is cluster and r_i.attacker is cluster then
        if !(r_i.attacker attacks any of C[r_i].defender) then
            N.removeAttack(r_i)
            continue
        end if
    end if
    if r_i.defender is cluster then
        if !(r_i.attacker attacks any of C[r_i].defender) then
            N.removeAttack(r_i)
            continue
        end if
    end if
    if r_i.attacker is cluster then
        if !(r_i.defender defends against any C[r_i].attacker) then
            N.removeAttack(r_i)
            continue
        end if
    end if
end for
```

Step 4: In this step we add the new attacks between the singletons. Due to the fact, that we copied all the attacks from $\hat{\mathtt{F}}$, 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 $\hat{\mathtt{F}}'$, we can limit the attack creation to the concretized singletons. The resulting AF is depicted in Example 3.5 and the pseudo-code in Algorithm 4.



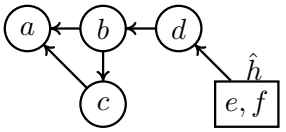
Example 3.5: Concretized AF $\hat{\mathtt{F}}'$ after Step 4

Algorithm 4 Concretizing Singletons Pseudocode Step 4

Require: $A : AF(a_1, r_1)$ ▷ Abstract Clustered AF
Require: $C : AF(a_2, r_2)$ ▷ Concrete AF
Require: $e : list(Arguments)$ ▷ Concretizer List
Require: $N : AF(a_3, r_3)$ ▷ Concretized Cluster

```
for e_i in e do
    for a_i in C[e_i].attacks do
        if a_i is singleton and (e_i, a_i) not in r_2 then
            N.addAttack((e_i, a_i))
        end if
    end for
    for a_i in C[e_i].defends do
        if a_i is singleton and (a_i, e_i) not in r_2 then
            N.addAttack((a_i, e_i))
        end if
    end for
end for
```

Step 5: The last step is to clean up the argumentation framework $\hat{\mathtt{F}}'$ by removing all empty clusters and mutating the clusters with exactly one singleton to the mentioned singleton. The resulting AF $\hat{\mathtt{F}}'$ is depicted in Example 3.6 and the pseudo-code in Algorithm 5.



Example 3.6: Concretized AF $\hat{\mathtt{F}}'$ after Step 5

Algorithm 5 Concretizing Singletons Pseudocode Step 5

Require: $A : AF(a_1, r_1)$ ▷ Abstract Clustered AF
Require: $C : AF(a_2, r_2)$ ▷ Concrete AF
Require: $e : list(Arguments)$ ▷ Concretizer List
Require: $N : AF(a_3, r_3)$ ▷ Concretized Cluster

```
for c_i in N.clusters do
    if c_i.argAmount == 1 then
        c_i ← Singleton
    else if c_i.argAmount == 0 then
        N.remove(c_i)
    end if
end for
```

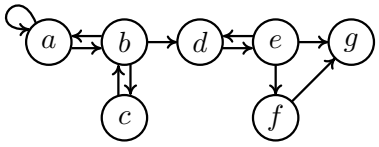
3.2 Computation of Concretizer List

When talking about clustering AFs, faithfulness is an important property. If an AF is spurious, we found atleast one semantics extension, which cannot be mapped to a concrete extension. Based on the spurious extensions, we try to mutate the clustered AF, to obtain faithfulness. This mutation is realized through the concretizer list.

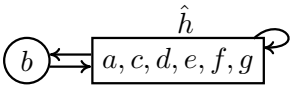
The concretizer list is a list of sets of clustered arguments. Each set is a unique combination of arguments, which are being concretized to find a faithful AF. All the sets of the concretizer list are attempted iteratively, where the order is dependend on the size of the set. We use a heuristicical approach, putting the main focus on local changes. Here we operate directly on the arguments and its attackers which make a set spurious, instead of applying global changes to the AF. Further, a minimal deviation of the abstract AF is usually 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 $\mathtt{G}$ is defined in Example 3.7 and the according abstract AF $\hat{\mathtt{G}}$ in Example 3.8.



Example 3.7: Concrete AF $\mathtt{G}$



Example 3.8: Abstract AF $\hat{\mathtt{G}}$

If we have a look at the stable extensions of the concrete AF $\mathtt{G}$, e.g. $\mathtt{stb} = [\{\mathtt{b}, \mathtt{c}\}]$ and at the abstractly stable extensions of the abstract AF $\hat{\mathtt{G}}$, e.g. $\hat{\mathtt{stb}} = [\{\mathtt{b}, \hat{\mathtt{h}}\}, \{\hat{\mathtt{h}}\}, \{\mathtt{b}\}]$, we can see that the abstractly stable extensions $\{\hat{\mathtt{h}}\}$ and $\{\mathtt{b}\}$ 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 $\{\mathtt{b}, \hat{\mathtt{h}}\}$.

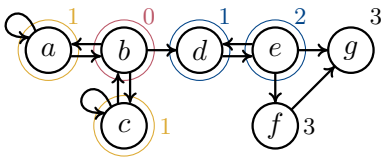
The first step is to filter out the clusters of the input, since clusters are not present in the concrete AF and therefore do not attack any singletons and are not being attacked. So we reduce the concretizer list from $\{\mathtt{b}, \hat{\mathtt{h}}\}$ to $\{\mathtt{b}\}$. The pseudo-code is listed in Algorithm 6.

Algorithm 6 Computation of Concretizer list Algorithm: Prefiltering

Require: $\hat{G} : AF(a_2, r_2)$ ▷ Abstract AF
Require: $s : list(Arguments)$ ▷ spurious arguments
1: **for** s_i in s **do**
2: **if** s_i in $\hat{G}$ is cluster **then**
3: s.remove(s_i)
4: **end if**
5: **end for**

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 Example 3.9 depicted in yellow and the attacker with the same depth, depicted in blue. The pseudo-code of this procedure is listed in Algorithm 7. Some arguments can have multiple depths (e.g. 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.



Example 3.9: 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 {`a`, `c`, `d`, `e`}. When building the combinations, we create the table defined in Table 3.1.

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

Table 3.1: Combinations of {`a`, `c`, `d`, `e`}

Algorithm 7 Computation of Concretizer list Algorithm: Neighbours

Require: $G : AF(a_1, r_1)$ ▷ Concrete AF
Require: $s : list(Arguments)$ ▷ Working List
1: $N \leftarrow []$ ▷ N = list of neighbours
2: **for** s_i in s **do** ▷ Get neighbours
3: **for** $n(1)_i$ neighbour of s_i **do** ▷ depth 1 attacker
4: **for** $n(2)_i$ neighbour of $n(1)_i$ **do** ▷ depth 2 attacker
5: $N.append(n(1)_i)$
6: $N.append(n(2)_i)$
7: **end for**
8: **end for**
9: **end for**

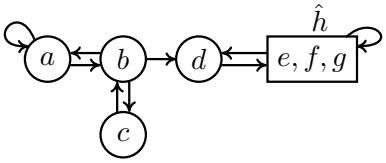
The combination table grows exponentially to the base of 2. Therefore, the size of the neighbours is crucial. If we have too many neighbours, the computation would need too much memory and turns infeasible to compute.

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 [{a} {b} {c} {d} {a, c} {a, d} {a, e} {c, d} {c, e} {d, e} {a, c, d} {a, c, e} {a, d, e} {c, d, e} {a, c, d, e}]. The pseudo-code for the last step is stated in Algorithm 8.

When concretizing the list, we find a set which leads to a faithful AF: {a, c, d} depicted in Example 3.10.



Example 3.10: Faithful AF $\hat{\mathtt{G}}'$

Algorithm 8 Computation of Concretizer list Algorithm: Combinations and Cleanup

Require: $G : AF(a_1, r_1)$ ▷ Concrete AF
Require: $\hat{G} : AF(a_2, r_2)$ ▷ Abstract AF
Require: $s : list(Arguments)$ ▷ Working List
Require: $ca : list(Arguments)$ ▷ Concretize arguments parameter
$C \leftarrow$ 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
return $s.sortBySize()$

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

4 Implementations

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

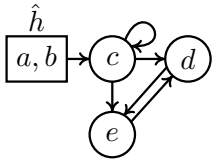
TODO: Semantic sets generation algorithm

4.4 Faithful/Spurious Determination

TODO: Determine faithful/spurious algorithm

REUSE THIS SOMEWHERE:

We encoded the semantics 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 Example 4.1.



Example 4.1: Abstract AF $\hat{\mathtt{G}}$

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:

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

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] Katie Atkinson et al. "Toward Artificial Argumentation." In: *AI Magazine* 38.3 (2017), pp. 25–36. DOI: https://doi.org/10.1609/aimag.v38i3.2704. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1609/aimag.v38i3.2704. URL: https://onlinelibrary.wiley.com/doi/abs/10.1609/aimag.v38i3.2704.

[3] Armin Biere et al., eds. *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009. ISBN: 978-1-58603-929-5. URL: http://dblp.uni-trier.de/db/series/faia/faia185.html.

[4] 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.

[5] 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.

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

[7] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures." In: *Proc. STOC*. Ed. by Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman. ACM, 1971, pp. 151–158.

[8] 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.

[9] 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.

[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. DOI: 10.1016/0004-3702(94)00041-x.

[11] 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.

[12] 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.

[13] 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.

[14] 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/.

[15] 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.

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

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] Stephen E. Toulmin. *The Uses of Argument*. 2nd ed. Cambridge University Press, 2003.