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, September 14, 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

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 [**DBLP:journals/frai/DietzKM24**]. To do so, the data needs to be stored and structured in a way, that AI can extract informations from it. This is part of knowledge representation and reasoning and that is why much research is being done in that field [**DBLP:journals/dagstuhl-manifestos/DelgrandeG0TW24**, **DBLP:journals/inffus/PopescuD23**].

Arguments can have many forms [**Toulmin˙2003**]. 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 [**DUNG1995321**]. Now we can define argumentation frameworks (AFs) and use them to evaluate conclusions [**DBLP:conf/fapr/Geffner96**]. 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. One of the first pioneers in the topic of argumentation frameworks was Dung. He defined the structure and functionalities in 1995 [**Dung1995-DUNOTA-2**].

Semantics define subsets of arguments that have a certain relation to each other. Dung defined different semantics [**Dung1995-DUNOTA-2**] 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 defintion of conflict-freeness is mainly a building block for the other semantics. A stable extension, 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.

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 [**DBLP:conf/kr/SaribaturW21**]. In this thesis we refer to the clustered AF as *abstract AF* and the original AF, where no clustering occurs is called *concrete AF*. 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 concrete AF. If no such spurious sets exist, then the clustered AF is said to be faithful, w.r.t. the concrete 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 $G = (A, R)$. Where we abstract the arguments into nodes and transform the opposing statement into attacks as shown in **??**. An opposing statement in this context would be for instance the argument a (i.e. *The sky is blue*) and e (i.e. *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 $\hat{G} = (\hat{A}, \hat{R})$ defined in **??**. 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 $\{a, b, c, e, f\}$.

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

Figure 1.1: Concrete and abstract AF

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

By computing the stable extensions of the abstract AF $\hat{G}$ $\texttt{stb} = \big\{\{d\}, \{d, \hat{g}\}\big\}$, we can observe that the AF is spurious due to the extension $\{d\}$, since the abstract stable extension 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 $\hat{G}' = (\hat{A}', \hat{R}')$ depicted in **??**, which has the following stable extensions $\texttt{stb} = \{\hat{g}, d\}$. This extension can be mapped to both stable extensions of the concrete AF G, by mutating the cluster $\hat{g}$ with $\{e\}$ or $\{b, f\}$. Thus, we created a faithful abstract AF.

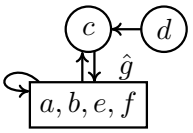


Figure 1.2: Concretized AF $\hat{G}'$

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 [**DBLP:journals/ai/DvorakGRW23**]. In fact, the complexity of proving faithfulness or spuriousness of an AF is $\prod_2^P$ hard [**DBLP:conf/kr/SaribaturW21**]. 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.
- $\cdots$

We provide an open source implementation of the previously listed tools [**Pasero2024-AFClusteri**

`https://github.com/p4s3r0/argumentation-framework-clustering`

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 discuss the background of the thesis. We start with the basic definition of argumentation frameworks (AFs) in **??**. Next we treat the clustering of AFs in **??** and finally we provide a quick overview of SAT-Solvers in **??**.

TODO: update when adding sections

2.1 Argumentation Frameworks

Argumentation frameworks were first formally described by Dung in 1995 [**DUNG1995321**]. They represent an information state, where various conclusions can be drawn 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 Artificial Intelligence (AI), e.g. in automated reasoning and logic programming [**AFINAIARLP**, **AFINAIARLPexample**]. But do also find their applications in other fields like natural language processing [**AFINNLP**], trust and reputation systems [**AFINTaRS**], legal and medical reasoning [**legalAndMedicalReasoning**], and even in game theory and strategic reasoning [**AFinGames**].

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 $A = \{a, b, c, d, e\}$ and the attacks $R = \big\{(a, b), (b, b), (a, c), (c, a), (c, d), (d, e), (e, d)\big\}$.

This AF can be represented as a directed graph as shown in **??**.

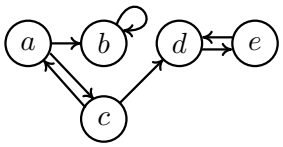


Figure 2.1: Argumentation Framework (AF) G

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 [**Dung1995-DUNOTA-2**] 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. 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.

Definition 1 ([**Dung1995-DUNOTA-2**]). *Let $G = (A, R)$ be an AF. Then a set $S \subseteq A$ is conflict-free in G if and only if for each $a, b \in S$ we have $(a, b) \notin R$.*

Example 1. *The conflict-free sets of the concrete AF G in* **??** *are* $\{\}$, $\{a\}$, $\{c\}$, $\{d\}$, $\{e\}$, $\{a, d\}$, $\{a, e\}$, $\{c, e\}$.

Admissible A set is admissible, if each argument in the admissible set has a defender in the admissible set.

Definition 2 ([**Dung1995-DUNOTA-2**]). *Let $G = (A, R)$ be an AF. Then a set $S \subseteq A$ is admissible in G, if and only if $S \in cf(G)$ and if $a \in S$ with $(b, a) \in R$, then there is a $c \in S$ with $(c, b) \in R$.*

Example 2. *The admissible sets of the concrete AF G in* **??** *are* $\{\}$, $\{a\}$, $\{c\}$, $\{e\}$, $\{a, d\}$, $\{a, e\}$, $\{c, e\}$.

Stable An extension 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.

Definition 3 ([**Dung1995-DUNOTA-2**]). *Let $G = (A, R)$ be an AF. Then a set $S \subseteq A$ is stable in G. If and only if $S \in cf(G), b \notin S$ 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 $(a, b) \in R$.*

Example 3. *The stable extensions of the concrete AF G in* **??** *are* $\{a, d\}$, $\{a, e\}$.

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 Answer Set Programming (ASP) [**DBLP:journals/corr/abs-1301-1388**] or, as in our case, with a Boolean Satisfiability Solver (SAT-Solver) [**DBLP:journals/amai/AmgoudD13**]. 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 [**DBLP:journals/ai/DvorakGRW23**].

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 semantics 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 $\hat{G} = (\hat{A}, \hat{R})$ with the arguments $\hat{A} = \{d, e, \hat{h}\}$, where the cluster $\hat{h}$ contains the arguments $\{a, b, c, d\}$ and the attacks being $\hat{R} = \big\{(\hat{h}, d), (d, e), (e, d), (\hat{h}, \hat{h})\big\}$. This AF can be represented as a directed graph as shown in **??**.

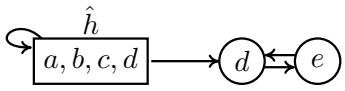


Figure 2.2: AF $\hat{G}$ clustered

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. abstractly conflict-free ($\hat{cf}$), abstractly admissible ($\hat{adm}$) and abstractly stable ($\hat{stb}$)). Let us consider a clustered AF $\hat{G} = \{\hat{A}, \hat{R}\}$ 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.

Definition 4 ([**DBLP:conf/kr/SaribaturW21**])**.** *Let $\hat{G} = (\hat{A}, \hat{R})$ be an AF. Then a set $S \subseteq \hat{A}$ is conflict-free in $\hat{G}$ if and only if for each $\hat{a}, \hat{b} \in$ singleton(S) we have $(\hat{a}, \hat{b}) \notin \hat{R}$.*

Example 4. *The abstractly conflict-free sets of the abstract AF G in* **??** *are* $\{\}$, $\{d\}$, $\{e\}$, $\{\hat{h}\}$, $\{e, \hat{h}\}$, $\{d, \hat{h}\}$.

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.

Definition 5 ([**DBLP:conf/kr/SaribaturW21**])**.** *Let $\hat{G} = (\hat{A}, \hat{R})$ be an AF. Then a set $S \subseteq \hat{A}$ is admissible in $\hat{G}$ if and only if $S \in \hat{cf}(\hat{G})$ and if $\hat{a} \in S$ with $(\hat{b}, \hat{a}) \in \hat{R}$ with singleton($\hat{a}$), then there is a $\hat{c} \in \hat{G}$ with $(\hat{c}, \hat{b}) \in \hat{R}$.*

Example 5. *The abstractly admissible sets of the abstract AF G in* **??** *are* $\{\}$, $\{e\}$, $\{\hat{h}\}$, $\{e, \hat{h}\}$, $\{d, \hat{h}\}$.

Stable A set of arguments is abstractly stable, if it is abstractly conflict-free and if an argument is not in the abstractly stable extension, it implies that an argument in the abstractly stable extension is attacking it. Furthermore if the abstractly stable extension is not attacking an argument, then every singleton attacking the argument is not in the abstractly stable extension.

Definition 6 ([**DBLP:conf/kr/SaribaturW21**])**.** *Let $\hat{G} = (\hat{A}, \hat{R})$ be an AF. Then an extension $S \subseteq \hat{A}$ is stable in $\hat{G}$ if and only if $S \in \hat{cf}(\hat{G}), \hat{b} \notin S$ implies that there is an $\hat{a} \in \hat{S}$ with $(\hat{a}, \hat{b}) \in \hat{R}$, and if S does not attack an $\hat{a} \in S$ then $\hat{b} \notin S$ whenever $(\hat{a}, \hat{b}) \in \hat{R}$ and singleton $(\hat{b})$.*

Example 6. *The abstractly stable extensions of the abstract AF G in* **??** *are $\{e, \hat{h}\}$, $\{d, \hat{h}\}$.*

Let us have a look at a concrete example to explain faithfulness. The concrete AF $G = (A, R)$ has the following arguments $A = \{a, b, c, d, e\}$ with these attacks: $R = \big\{(a, b), (b, b), (a, c), (c, a), (c, d), (d, e), (e, d)\big\}$. This AF can be represented as a directed graph shown in **??**.

Now we can group the arguments $\{a, b, c, d\}$ together into one single cluster $\hat{h}$. The arguments for the abstract AF $\hat{G} = (\hat{A}, \hat{R})$ would then be $\hat{A} = \{e, \hat{h}\}$, where the cluster $\hat{h}$ is composed of $\{a, b, c, d\}$ and the according attacks are $\hat{R} = \big\{(\hat{h}, e), (e, \hat{h}), (\hat{h}, \hat{h})\big\}$. 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 **??**.

(a) Concrete AF G

(b) Abstract AF $\hat{G}$

Figure 2.3: Concrete and abstract AF

If we compare the stable extensions of the concrete AF G (e.g. $\texttt{stb} = \big\{\{a, e\}, \{a, d\}\big\}$) with the abstractly stable extensions of the abstract clustered AF $\hat{G}$ (e.g. $\hat{\texttt{stb}} = \big\{\{\hat{h}\}, \{e\}, \{e, \hat{h}\}\big\}$), we see that it is spurious due to the abstractly stable extension $\{e\}$ which cannot be mapped to one of the concrete stable extensions. The mapping of semantics 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 $\hat{h}$ can mutate to $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{a, b\}$, $\{a, c\}$, $\{a, d\}$, $\{b, c\}$, $\{b, d\}$, $\{c, d\}$, $\{a, b, c\}$, $\{a, b, d\}$, $\{a, c, d\}$, $\{b, c, d\}$, $\{a, b, c, d\}$. To create a faithful clustered AF, we can concretize one or more arguments of the cluster. By concretizing the argument $\{d\}$, we obtain a new AF $\hat{G}' = (\hat{A}', \hat{R}')$ with the arguments $\hat{A}' = \{d, e, \hat{h}\}$, where the

cluster $\hat{h}$ is composed of $\{a, b, c\}$, and the according attacks are $\hat{R}' = \{(d, \hat{h}), (d, e), (e, d), (\hat{h}, \hat{h})\}$.

With this defintion we can build the concretized abstract graph $\hat{G}'$ in **??**.

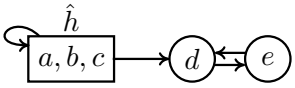


Figure 2.4: Concretized AF $\hat{G}'$

Every abstractly stable extension in **??** (e.g. $\{d, \hat{h}\}, \{e, \hat{h}\}$) can be mapped to one of the concrete stable extensions of G, which means that the clustered AF $\hat{G}'$ is faithful.

2.3 Boolean Satisfiability

A SAT(isfability)-Solver is used to compute Boolean formulas in a rather efficient way [**Biere2009**]. 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, e.g. in verification and validation of software and hardware [**DBLP:conf/dagstuhl/Gogolla09**, **DBLP:books/daglib/0045943**]. But also in AI and machine learning [**DBLP:phd/basesearch/Liang18a**] and even in security [**Pasero2022-SATHashFunctions-Repo**, **DBLP:journals/iacr/LinYXTS24**].

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 [**Cook71**]. 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 [**SAT-Solver-Competition**]. 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 [**z3-SAT-Solver**].

3 Algorithm to obtain Faithful Clustering

In this chapter we have a closer look at the algorithms we designed and how they work. We begin with **??**, where we define the Boolean encodings for the semantics. In **??** we explain, the concretization of a clustered argument. Next, in **??** 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 then described in **??**. And finally we state the heuristics and refinements in **??**.

TODO: Update when adding sections

3.1 Encodings

We previously defined the semantics in mathematical notation. But since we use SAT-Solving to compute extensions under a chosen semantics we develop in this section encodings in Boolean logic of the semantics of clustered AFs. We develop these encodings for (abstract) conflict-free, (abstract) admissible, and (abstract) stable extensions. For each semantics we present the corresponding Boolean encoding and an example.

In general, given an AF $\hat{G} = (\hat{A}, \hat{R})$, we will use Boolean variables corresponding to arguments. That is, if $a \in \hat{A}$ is an argument, then a is also a Boolean variable. The intended meaning is then that in a truth-value assignment a variable a is *true*, then there is a corresponding set of arguments with a in the set. Formally, if τ is a truth-value assignment on the variables in $\hat{A}$, then the corresponding set of arguments is defined as $\{a \in A \mid \tau(a) = 1\}$.

Conflict-Free We begin with (abstract) conflict-free sets. Recall that a set of (clustered) arguments is conflict-free if there is no attack between singletons in the set. Thus, the empty set is always part of the conflict-free sets.

Definition 7. *Let $\hat{G} = (\hat{A}, \hat{R})$ be an abstract AF. Where $\hat{A}$ are the representations of the arguments as Boolean variables and $\hat{R}$ are the the attacks between the arguments.*

$$\hat{cf}(\hat{G}) = \bigwedge_{a \in A_{SINGLE}} \Big(\bigwedge_{b:(b,a) \in R, b \in A_{SINGLE}} \neg(a \wedge b) \Big)$$

Example 7. *Let us have a look at an example and define the abstract AF $\hat{G} = (\hat{A}, \hat{R})$ to be the AF depicted in **??**. With the arguments $\hat{A} = \{a, b, c, d\}$ and the attacks $\hat{R} = \big\{(a, a), (a, b), (b, c), (d, a), (d, b), (d, c)\big\}$.*

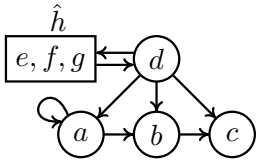


Figure 3.1: Abstract AF $\hat{G}$

By applying the formula from **??** *to the AF* $\hat{G}$*, we obtain the following Boolean expression.*

$$\psi = \neg(b \wedge a) \wedge \neg(b \wedge d) \wedge \neg(a \wedge d) \wedge \neg(a \wedge a) \wedge \\ \neg(c \wedge b) \wedge \neg(c \wedge d)$$

The models of ψ *in this example are* $\big\{\{\}\ \{b\}, \{c\}, \{d\}, \{\hat{h}\}, \{b, \hat{h}\}, \{c, \hat{h}\}, \{d, \hat{h}\}\big\}$.

If we compare the models of ψ with the conflict-free sets of $\hat{G}$, we can observe that they are equal. In fact, **??** is a Boolean representation to obtain conflict-free sets from an AF. The formula can be applied to concrete and abstract AFs, because if no cluster is present, the formula corresponds to the previously defined conflict-free formula and only adds the layer of abstraction if atleast one cluster is present.

Admissible Next we continue with (abstract) admissible sets. Recall that a set of arguments is abstractly admissible, if it is abstractly conflict-free and if every singleton which is being attacked, has a defender. For admissibility, the empty set does always satisfy the criteria and therefore is part of the admissible sets.

Definition 8. *Let* $\hat{G} = (\hat{A}, \hat{R})$ *be an abstract AF. Here the arguments as Boolean variables denote to* $\hat{A}$ *and the attacks of* G *denote to* $\hat{R}$*.*

$$\hat{adm}(\hat{G}) = \hat{cf}(\hat{G}) \wedge \bigwedge_{a \in A_{SINGLE}} (a \rightarrow \bigwedge_{b:(b,a) \in R} (\bigvee_{c:(c,b) \in R} c))$$

Example 8. *Let us have a look at an example and define the AF* $\hat{G} = (\hat{A}, \hat{R})$ *to be the AF depicted in* **??**. *Where the arguments are* $\hat{A} = \{a, b, c, d, e\}$ *and the attacks* $\hat{R} = \big\{(a, a), (a, b), (b, c), (d, a), (d, b), (d, d), (d, e), (e, d)\big\}$.

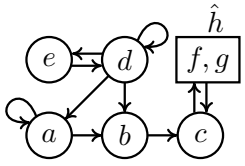


Figure 3.2: Abstract AF $\hat{G}$

By applying the formula defined in **??** *to the AF* $\hat{G}$*, we obtain the following Boolean expression.*

$$\begin{aligned}\psi = \big(&\neg(a \wedge a) \wedge \neg(a \wedge d) \wedge \neg(b \wedge d) \wedge \neg(b \wedge c) \wedge \neg(b \wedge d) \wedge \neg(c \wedge b) \wedge \neg(d \wedge e) \\ &\wedge \neg(d \wedge d) \wedge \neg(e \wedge d)\big) \\ &\wedge \big(a \rightarrow ((a \vee d) \wedge (e \vee d)) \wedge \big(b \rightarrow ((a \vee d) \wedge (b) \wedge (e \vee d))\big)\big) \\ &\wedge \big(c \rightarrow ((a \vee c \vee d) \wedge (\hat{h}))\big) \wedge \big(d \rightarrow (d) \wedge (e \vee d)\big) \wedge \big(e \rightarrow (e \vee d)\big)\end{aligned}$$

The models of ψ *in this example are* $\big\{\{\}, \{c\}, \{e\}, \{\hat{h}\}, \{c, e\}, \{c, \hat{h}\}, \{e, \hat{h}\}, \{c, e, \hat{h}\}\big\}$.

If we compare the models of ψ with the admissible sets of $\hat{G}$, we can observe that they are equal. Indeed, **??** provides a Boolean representation that allows the derivation of admissible sets from an AF. If the AF $\hat{G}$ has no cluster, the formula reduces itself to the previously defined admissible formula.

Stable Finally we take a look at the (abstract) stable extensions. Recall, that a set of arguments is abstractly stable, if it is abstractly conflict-free and if an argument is not in the abstractly stable extension, it implies that an argument in the abstractly stable extension is attacking it. Furthermore if the abstractly stable extension is not attacking an argument, then every singleton attacking the argument is not in the abstractly stable extension. Other than conflict-free and admissible, the empty set is not part of the stable extension.

Definition 9. *Let* $\hat{G} = (\hat{A}, \hat{R})$ *be an abstract AF. In this case,* $\hat{A}$ *represents the arguments as Boolean variables, and* $\hat{R}$ *represents the attacks of* $\hat{G}$*.*

$$\hat{stb}(\hat{G}) = \hat{cf}(\hat{G}) \wedge \bigwedge_{a \in A} \big(a \bigvee_{b:(b,a) \in R} b\big) \wedge \bigwedge_{a \in A} \big(\big(a \bigwedge_{b:(b,a) \in R} \neg b\big) \rightarrow \big(\bigwedge_{c:(a,c), c \in A_{SINGLE}} \neg c\big)\big)$$

Example 9. *Let us have a look at an example and define the AF* $\hat{G} = (\hat{A}, \hat{R})$ *to be the AF depicted in* **??**. *Where the arguments are* $\hat{A} = \{a, b, c, d\}$ *and the attacks* $\hat{R} = \big\{(a, a), (a, b), (a, d), (b, a), (b, d), (c, b), (d, a)\big\}$.

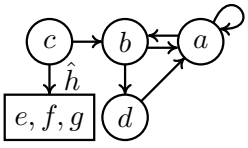


Figure 3.3: Abstract AF $\hat{G}$

If we apply the encoded formula defined in **??** *to the AF* $\hat{G}$*, we obtain the following Boolean expression.*

$$\begin{aligned}\psi = \big((\neg(a \wedge a)) \wedge (\neg(a \wedge b)) \wedge (\neg(b \wedge a)) \wedge (\neg(b \wedge c)) \wedge (\neg(d \wedge b))\big) \\ \wedge \big((a \vee a \vee b \vee d) \wedge (b \vee a \vee c) \wedge (d \vee b) \wedge (\hat{h} \vee c)\big) \\ \wedge \big(((a \wedge \neg a \wedge \neg b \wedge \neg d) \rightarrow (\neg a \wedge \neg b)\big) \\ \wedge \big((b \wedge \neg a \wedge \neg c) \rightarrow (\neg a \wedge \neg d)) \wedge ((d \wedge \neg b) \rightarrow \neg a)\big)\end{aligned}$$

The models of ψ *in this example are* $\big\{\{c, d\},\ \{c, d, \hat{h}\}\big\}$.

By comparing the models of ψ with the stable sets of $\hat{G}$, we find that they are identical. Specifically, **??** is a Boolean expression to derive admissible sets from an AF. If the AF has no clusters, the Boolean expression reduces to the admissible formula defined earlier.

3.2 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 $\hat{F}$ to create a new AF $\hat{F}'$ and transforming it. The transformation is guided by five steps considering the unchanged abstract AF $\hat{F}$ and the concrete AF F. To improve the understanding of each step, we accompany the explanation with the example depicted in **??**, where we concretize the arguments a and b.

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

Figure 3.4: Concrete and abstract AF

Step 1: Each argument needing concretization is first removed from the parent cluster and added as a singleton in $\hat{F}'$. 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 **??** and the pseudo-code in **??**.

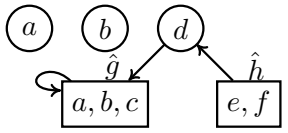


Figure 3.5: Concretized AF $\hat{F}'$ after Step 1

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 $\hat{g}$ in the abstract AF $\hat{F}$, by concretizing b, the attack $(a, \hat{g})$ would not be present anymore. The resulting AF $\hat{F}'$ is depicted in **??** and the pseudo-code in **??**

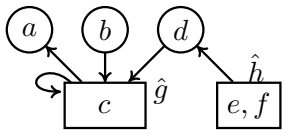


Figure 3.6: Concretized AF $\hat{F}'$ after Step 2

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

```
1: for e_i in e do
2:     for att_i in C[e_i].attacks do
3:         if att_i to cluster c_i then
4:             N.addAttack((e_i, c_i))
5:         end if
6:     end for
7:     for def_i in C[e_i].defends do
8:         if def_i from cluster c_i then
9:             N.addAttack((c_i, e_i))
10:        end if
11:    end for
12: end for
```

Step 3: After adding the new attacks, we need to check which attacks from $\hat{F}$ are still present in $\hat{F}'$. If an attack does not persist through the concretization, we remove it in $\hat{F}'$. An attack is not present anymore; if we remove one of the arguments being attacked or attacked by argument a from a cluster $\hat{f}$ and no other attack exists, s.t. a is attacked from/attacking an argument within $\hat{f}$. 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 **??** and the pseudo-code in **??**.

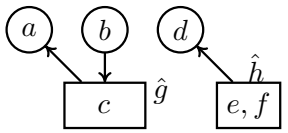


Figure 3.7: Concretized AF $\hat{F}'$ after Step 3

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 $\hat{F}$, 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 $\hat{F}'$, we can limit the attack creation to the concretized singletons. The resulting AF is depicted in **??** and the pseudo-code in **??**.

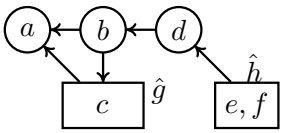


Figure 3.8: Concretized AF $\hat{F}'$ after Step 4

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 $\hat{F}'$ by removing all empty clusters and mutating the clusters with exactly one singleton to the mentioned singleton. The resulting AF $\hat{F}'$ is depicted in **??** and the pseudo-code in **??**.

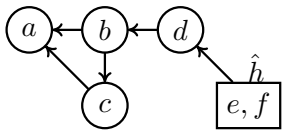


Figure 3.9: Concretized AF $\hat{F}'$ after Step 5

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
```

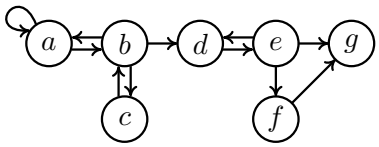
3.3 Computation of Concretizer List

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.

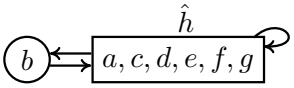
The concretizer list is a list of sets of clustered arguments. Each set is a unique combination of arguments, which are being concretized to find a faithful AF. All the sets of the concretizer list are attempted iteratively, where the order is dependend on the size of the set. We use a heuristicical 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 semantics 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 **??** and the according abstract AF $\hat{G}$ in **??**.



(a) Concrete AF G



(b) Abstract AF $\hat{G}$

Figure 3.10: Concrete and abstract AF

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

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 $\{b, \hat{h}\}$ to $\{b\}$. The pseudo-code is listed in **??**.

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 **??** depicted in yellow and the attacker with the same depth, depicted in blue. The pseudo-code of this procedure is listed in **??**. 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.

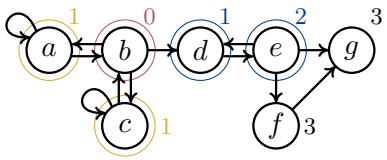


Figure 3.11: Singletons depth with b as viewpoint

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

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

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. In our example, we remove the set $\{b\}$.

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 $\{\{a\}\ \{c\}\ \{d\}\ \{a, c\}\ \{a, d\}\ \{c, d\}\ \{d, e\}\ \{a, c, d\}\ \{a, d, e\}\ \{c, d, e\}\ \{a, c, d, e\}\}$. The pseudo-code for the last step is stated in **??**.

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 $s.sortBySize()$

3.4 Algorithmic Approach to Compute Faifthul Clusterings

When clustering AFs, information gets lost. If crucial information is abstracted in a way, s.t. we can draw erroneous conclusions, we have a spurious AF. Spurious abstract AFs do not represent the according concrete AFs. Thus, we need to mutate the clustered AF, s.t. only correct solutions can be drawen, which we then call faithful. The algorithm we designed, takes as input a concrete AF denoted to G, and an abstract AF $\hat{G}$. First we determine, if the abstract AF $\hat{G}$ is spurious, by calculating the semantics extensions and attempting to find spurious set. If no spurious set can be found, the AF is faithful and no further mutations are needed.

But if we find spurious extensions, we build the concretizer list as specified previously in **??**. Recall, that the concretizer list operates with the depth of neighbours of 2. This means, that depending on the AF, we can not guarantee to find a faithful AF. If the spurious extension of an AF G has an argument a with depth 3 to an argument b, and the only faithful AF is the concrete one, than we will not find the faithful AF.

Finally, we concretize the concretizer list set after set and check for faithfulness. The algorithm is listed in **??**.

Example 10. *Let us have a look at an example and define an AF* $G = (A, R)$ *with the arguments* $A = \{a, b, c, d, e, f\}$ *and the attacks* $R = \big\{(a, a), (a, c), (b, a), (c, b), (c, d), (d, c), (e, b), (b, e), (f, f)\big\}$*, depicted in* **??**. *When clustering the arguments* $\{a, b, f\}$ *we obtain the abstract AF* $\hat{G} = (\hat{A}, \hat{R})$ *depicted in* **??**. *Where the arguments are* $\hat{A} =$

$\{c, d, e, \hat{h}\}$. *The attacks of the abstract AF are* $\hat{R} = \{(\hat{h}, \hat{h}), (\hat{h}, c), (c, d), (d, c), (\hat{h}, e), (e, \hat{h})\}$ *and the cluster* $\hat{h}$ *contains the arguments* $\{a, b, f\}$.

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

Figure 3.12: Concrete and abstract AF

When taking a look at the conflict-free sets of the concrete AF G (e.g. $\big\{\{\}, \{b\}, \{c\}, \{d\}, \{e\}, \{b, d\}, \{c, e\}, \{d, e\}\big\}$) and the abstractly conflict-free sets of the abstract AF $\hat{G}$ (e.g. $\big\{\{\}, \{c\}, \{d\}, \{e\}, \{\hat{h}\}, \{c, e\}, \{c, \hat{h}\}, \{d, e\}, \{d, \hat{h}\}, \{e, \hat{h}\}, \{c, e, \hat{h}\}, \{e, d, \hat{h}\}\big\}$) we can observe, that the abstract AF $\hat{G}$ is spurious due to the sets $\big\{\{\hat{h}, d, e\}, \{\hat{h}, e\}, \{c, \hat{h}, e\}, \{c, \hat{h}\}\big\}$. Now we compute the concretizer list and obtain the sets $\big\{\{a\}, \{b\}, \{a, b\}\big\}$. In this example, the only faithful solution we can obtain by concretizing is the concrete AF G. Thus, the set $\{a, b\}$ leads to a faithful AF.

Algorithm 9 Compute Faithful Clusters

Require: $G : AF(a_1, r_1)$ ▷ Concrete AF
Require: $\hat{G} : AF(a_2, r_2)$ ▷ Abstract AF
1: $s_c \leftarrow computeSemanticsExtension(G)$
2: $s_a \leftarrow computeSemanticsExtension(\hat{G})$
3: $sp \leftarrow computeSpuriousSemantics(s_c, s_a)$
4: **if** $sp.length == 0$ **then**
5: **return** $\hat{G}$
6: **end if**
7: $conc \leftarrow computeConcretizerList(sp)$
8: **for** set_i in sp **do**
9: $\hat{G}' \leftarrow concretize(\hat{G}, set_i)$
10: $s_a \leftarrow computeSemanticsExtension(\hat{G}')$
11: **if** $sp.length == 0$ **then**
12: **return** $\hat{G}'$
13: **end if**
14: **end for**

3.5 Heuristics and Refinements

To speed up the process of computing semantics extensions, we came up with heuristics and refinements for specific scenarios. Some heuristics were already mentioned previously, like sorting the concretizer list by size to obtain a rather small mutation of the spurious AF. But we also implemented shortcuts and refinements, specific for every semantics.

conflict-free Let us begin with the refinement made for the conflict-free sets computation. When computing a single conflict-free set, with the amount of arguments greater than 1, we extract every subset, which is conflict-free as well. Formally, if X is a conflict-free set and $|X| \geq 2$, then $\forall S \subseteq X$, S is conflict-free. This property can be derived from the defintion of conflict-free. Recall that a set of (clustered) arguments is conflict-free if there is no attack between singletons in the set. Thus, if there are no attacks between the arguments in X, and $S \subseteq X$, we can conclude that there are not attacks between the argument in S.

admissible Next, we will define the refinements made for the admissible semantics. Other than conflict-free sets, for (abstract) admissible sets we cannot derive directly information from the subsets. But we can decrease the computation time of showing faithfulness by adding the negated admissible formula for the concrete AF. This removes the faithful admissible sets which can be directly mapped from the abstract admissible sets to the concrete ones. The remaining sets to be computed are the spurious sets, or abstract admissible sets containing atleast one cluster (since the concrete AF does not possess clusters and are therefore not ignored).

Recall, that a set of arguments is abstractly admissible, if it is abstractly conflict-free and if every singleton which is being attacked, has a defender. We refine the formula from **??**, denoted as φ_{ADM} with the refinement denoted as η_{ADM}.

$$\hat{adm} = \ \varphi_{ADM}(\hat{F}) \wedge \eta_{ADM}(F)$$

Where $\eta_{ADM}(F)$ is defined as the negated admissible formula of the concrete AF.

$$\eta_{ADM}(F) = \ \overline{cf}(F) \wedge \overline{def}(F)$$

The two new introduced components (i.e. $\overline{cf}$ and $\overline{def}$) define together the refinement. Here, $\overline{cf}$ stands for the negated part of the conflict-free sets from the concrete AF and $\overline{def}$ defines that we want to ignore all the sets from the concrete AF which are defending themselves from an attack.

$$\overline{cf}(F) = \bigvee_{a \in A_{SINGLE}} \Big(\bigvee_{b:(b,a)\in R, b\in A_{SINGLE}} (a \wedge b) \Big)$$

$$\overline{def}(F) = \Big(\bigvee_{a \in A_{SINGLE}} \big(a \wedge \bigvee_{b:(b,a)\in R} \big(\bigwedge_{c:(c,b)\in R} \neg c \big) \big) \Big)$$

stable For the stable semantics we apply the same principle as for abstract admissibility. Since we cannot conclude anything from the subsets of the computed (abstract) semantics, we need to add the negation of the stable formula for the concrete AF. With this, we reduce the semantics extensions which need to be computed drastically without losing crucial information. The eliminated abstract extensions are simply the extensions, which can be directly mapped to the concrete stable extensions, which have no impact on the spuriousness of the AF. The remaining extensions that need to be computed are either spurious extensions, or extensions which have atleast one cluster and need to be expanded and checked separately for spuriousness. Recall, that a set of arguments is abstractly stable, if it is abstractly conflict-free and if an argument is not in the abstractly stable extension, it implies that an argument in the abstractly stable extension is attacking it. Furthermore if the abstractly stable extension is not attacking an argument, then every singleton attacking the argument is not in the abstractly stable extension. We refine this formula from **??**, denoted as φ_{STB} with the refinement denoted as η_{STB}.

$$\hat{stb} = \varphi_{STB}(\hat{F}) \wedge \eta_{STB}(F)$$

Here η_{STB} is defined as the negated stable formula of the concrete AF.

$$\eta_{STB}(F) = \overline{cf}(F) \wedge \overline{att}(F) \wedge \overline{con}(F)$$

The three components (i.e. $\overline{cf}$, $\overline{att}$, $\overline{con}$) represent three different conditions which together define the refinement. Here, $\overline{cf}$ is the negated part of conflict-freeness of the concrete AF, $\overline{att}$ describes, that we want to ignore the concrete stable sets which impliy that if and argument being outside the stable set, has an attacker inside the stable set. Finally, $\overline{con}$ is the negated form of if the stable extension is not attacking an argument, then every singleton atacking the argument is not in the abstractly stable extension.

$$\overline{cf}(F) = \bigvee_{a \in A_{SINGLE}} \Big(\bigvee_{b:(b,a)\in R, b\in A_{SINGLE}} (a \wedge b) \Big)$$

$$\overline{att}(F) = \bigvee_{a \in A} \big(\neg a \bigwedge_{b:(b,a)\in R} \neg b\big)$$

$$\overline{con}(F) = \bigvee_{a \in A} \big((a \bigwedge_{b:(b,a)\in R} \neg b) \wedge \big(\bigvee_{c:(a,c), c\in A_{SINGLE}} c\big)\big)$$

4 Implementations

In this chapter we dive into the implementation part and discuss different approaches. First, we specify on how the AFs we run the experiments on were created in **??**. Here we describe three different methods, with their advantages and disadvantages. Next we explain two different settings to optimize the spurious/faithful check in **??**. In **??** is the explanation on how we generated the semantics extensions and finally, we will tackle refuted theories in **??**.

4.1 Creating AFs

We created three different approaches to generate AFs. Each of them has a different idea and generates AFs with different properties. While the random-based approach generates chaotic AFs, which are typically not similar to real-world problems, the grid-based approach has structure and is therefore more related to real-world problems. The level-based approach has even more structure and assures that we can not derive to many neighbours from a problematic argument. For each approach, we provide an additional figure for better visualization and example AFs generated with the algorithm.

random-based Let us begin with the random-based approach. The arguments of the script are `<arg_amount>` and `<p>`. The `<arg_amount>` specifies how many arguments the AF has and the argument `<p>` defines the probability of an attack between two arguments. This approach creates chaotic AFs with no structure. Basically, if we take a look at **??** we can see a graph with potential attacks depicted with dotted arrows. Every potential attack has a a probability of `<p>` to be an actual attack of the generated AF.

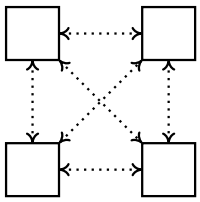
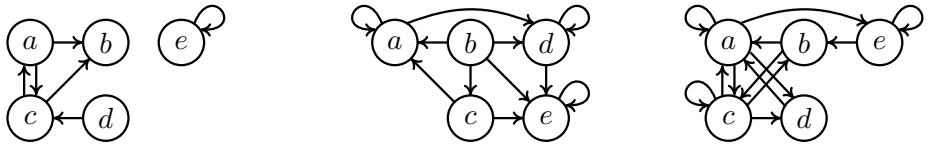


Figure 4.1: Random-Based Approach. $Amount = 4$

Random-based generated AFs have the property (depending on the probability value) of being hard to to predict on how good the AF is solvable. This is due the fact, that the neighbours of each argument are highly dependent on the amount of attacks and randomness (since an argument can attack every other arguments). Example AFs generated with the random-based approach can be seen in **??**



(a) `arg_amount`=5 `p`=0.25 (b) `arg_amount`=5 `p`=0.5 (c) `arg_amount`=5 `p`=0.75

Figure 4.2: Example AF generated with random-based approach

grid-based Next we are going to discuss the grid-based approach. The arguments for the script are `<arg_amount>`, being the amount of arguments the AF has and `<p>`, which is the probability that an attack between two arguments occurs. Different to the random-based approach, attacks can only happen between the direct neighbours of the grid (i.e. top, bottom, right, left). The grid is a $n \times n$ grid, with n being equal to $\lfloor(\sqrt{\texttt{<arg\_amount>}})\rfloor$. An example grid can be seen in **??**.

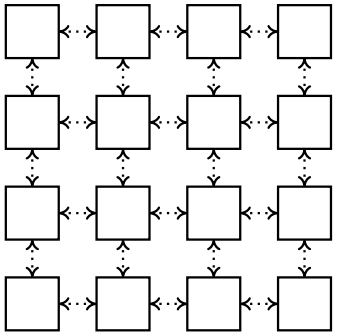
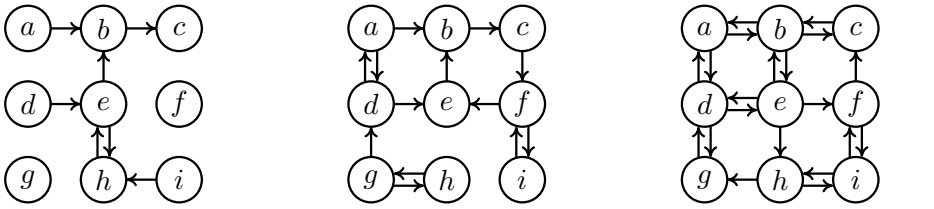


Figure 4.3: Grid-Based Approach. $Amount = 16$

With the grid-based approach, we obtain a more structured AF. Structured in this context means, that the attacks between the arguments are restricted to locality. Due to this restriction, we reduce the amount of neighbours drastically in comparison to the random-based approach. Since we have less neighbours, we decrease the computation time and increase the chance to find a faithful AF. Example AF created with the grid-based approach can be seen in **??**.

level-based The last algorithm to create concrete AFs we provide, is the level-based approach. The arguments for this script are `<arg_amount>`, `<level>` and `<p>`. Same as for the grid-based and random-based approach, `<arg_amount>` defines how many arguments the computed AF has. The `<level>` argument restricts the height of the grid to the provided value and `<p>` is again the probability that an attack between to arguments occurs. The difference to the grid-based approach is the dimension of the grid. While the grid-based approach uses a $n \times n$ grid, in the level-based approach we



(a) `arg_amount`=9 `p`=0.25 (b) `arg_amount`=9 `p`=0.5 (c) `arg_amount`=9 `p`=0.75

Figure 4.4: Example AF generated with grid-based approach

use a `<level>` $\times n$ grid. In this context, n is equal to $\lceil$`<arg_amount>`$/$`<level>`$\rceil$. An example grid is depicted in **??**.

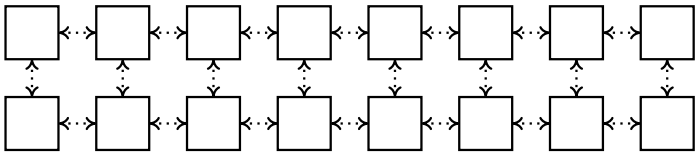


Figure 4.5: Level-Based Approach. $Level = 2$ and $Amount = 16$

With the level-based approach, we obtain the same structured AF as for the grid-based, but with less neighbours. Every argument can only have $min($`<level>` $+ 1, 4)$ amount of direct neighbours. This reduces the neighbours even further and thus, decreases the overall computation effort. Example AFs created with the level-based script can be seen in **??**

(a) `arg_amount`=9 `level`=2 `p`=0.25 (b) `arg_amount`=9 `level`=2 `p`=0.75

Figure 4.6: Example AF generated with grid-based approach

clustering The random-based, grid-based and level-based only generate concrete AFs. To be able to generate the abstract AFs based on the concrete ones, we created another independent script. Since clusters make a lot more sense when respecting the locality of the arguments, we cluster arguments which are next to each other (e.g. direct neighbours). The size of the cluster is determined on runtime and is a random value between 2 and the amount of arguments present in the concrete AF.

4.2 BFS and DFS Approach

Breadth-First-Search (BFS) and Depth-First-Search (DFS) are usually used in algorithms to traverse graphs. Where BFS visits each node in order of distance to the start, DFS follows a direct path and only backtracks, if is has to. We, however, are not using the two approaches to traverse a graph, but we are using a similar principle to compute faithfulness of an abstract AF. The BFS approach in our implementation first computes all the semantics extensions of the abstract AF, then computes all the semantics extensions of the concrete AF and finally compares them to check for spuriousness.

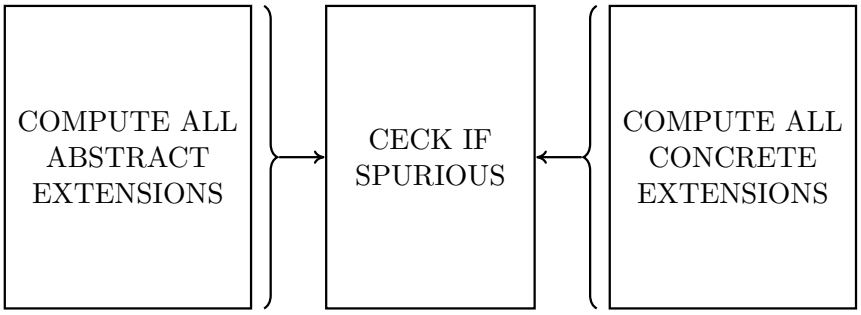


Figure 4.7: BFS visualization

The BFS approach, visualized in **??** is practical for AFs which do not have many semantics extensions. If the semantics sets take too long to compute, we will run into a timeout no matter at which seed the SAT-Solver is operating on. BFS is a solid and robust approach, nevertheless, there is no space of randomness and lucky early terminations.

On the other hand, we have the DFS approach depicted in **??**. When using DFS, instead of calculating all the abstract extensions at once, we verify each computed set directly. Verifying in this context means, to check if the extension can be mapped to one of the concrete extensions. But instead of computing every possible concrete set, and checking for a valid mapping, we encode the extensions directly, add a new context (using incremental solving), and check for satisfiability. If the SAT-Solver returns *UNSAT*, we found a spurious extension and, therefore, showed that the abstract AF is spurious.

DFS has some overhead, due to the context switches resulting with a longer computation power for faithful AFs. Nevertheless, depending on the seed of the SAT-Solver, we can obtain a result much faster than with BFS. If the first computed semantics extension is already spurious, we save a lot of computation power and can even solve AFs, which are not feasible for the BFS approach.

4.3 Generating Semantics Sets

To be able to create semantics extensions with a SAT-Solver, we need to encode the semantics (previously defined in **??** and the according refinements in **??**) into Boolean

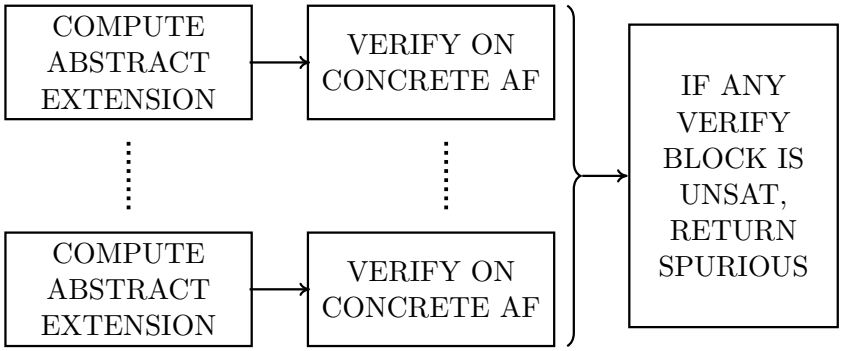


Figure 4.8: DFS visualization

logic. Since the semantics definitions are written in an generelazide form, we need to unpack the n-ary con-/disjunction and encode them appropriately.

Example 11. *Let us have a look at a concrete example with an abstract clustered AF* $\hat{G} = (\hat{A}, \hat{R})$ *defined in* **??**. *The arguments of the AF are* $\hat{A} = \{c, d, e, \hat{h}\}$, *with the cluster* $\hat{h}$ *containing the arguments* $\{a, b\}$ *and the rules being* $\hat{R} = \big\{(\hat{h}, c),\ (c, c),\ (c, d),\ (c, e),\ (e, d),\ (d, e)\big\}$.

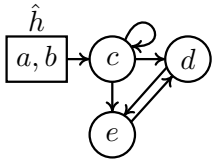


Figure 4.9: Abstract AF $\hat{G}$

n-ary conjunction *To concatinate all the arguments with a Boolean AND of the AF* $\hat{G}$, *we use the n-ary conjunction operator. We select an argument* a *from the singletons of the AF* $\hat{G}$ *and concatinate them conjunctively. Note, that with the subscript* SINGLE *we only include the singletons of the AF.*

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

n-ary disjunction *To concatinate all the arguments of the AF* $\hat{G}$ *with a Boolean OR, we use the n-ary disjunction operator. By iterating over all the arguments of the AF* $\hat{G}$ *and concatinating them disjunctively we generate the Boolean formula for the SAT-Solver. Note, that this time we do not have the subscript* SINGLE, *so we include the clusters as well.*

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

extracting attacks *We can also iterate over every attack of the AF $\hat{G}$. We can extract both arguments* a *(being the attacker) and* b *(being the defender) from the attacks by selecting the tuples from $\hat{R}$.*

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

With this encoding, we can compute a model with the SAT-Solver and extract a semantics set. If we try to compute another extension, we first have to add a new Boolean rule. This rule removes the previously found model from the satisfiable solutions by adding the negated disjunction of the arguments from the extension. Or formally, if S is the previously found extension, we add the following formula conjunctively to the previous rule set.

$$\bigvee_{a \in S} \neg a$$

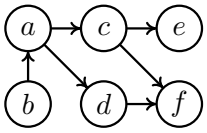
4.4 Refuted Theories

When developing the tool, we came up with different theories. Some theories were discarded immediately, some were implemented (recall **??**) and others had to be proven wrong. This section describes the most promising theory which turned out to be wrong.

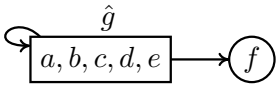
We came up with the theory, that every subset of a stable spurious extension, is spurious as well. Formally, if S is a spurious stable set of the abstract AF $\hat{G}$. Then for every subset $T \subseteq S$, T is spurious as well. This would reduce the computation time of finding spurious sets drastically. Unfortunately, we can disprove this theory with a counter-example.

Example 12. *Let us define $G = (A, R)$ to be a concrete AF, with the arguments $A = \{a, b, c, d, e, f\}$ and the attacks $R = \big\{(b, a), (d, a), (a, d), (a, c), (c, e), (c, f), (d, f)\big\}$ depicted in* **??**. *The spurious abstract AF $\hat{G} = (\hat{A}, \hat{R})$ is defined with the arguments $\hat{A} = \{\hat{g}, f\}$, with the cluster $\hat{g}$ containing the arguments $\{a, b, c, d, e\}$ and the attacks being $\hat{R} = \big\{(\hat{g}, \hat{g}), (\hat{g}, f)\big\}$ depicted in* **??**.

When computing the stable extensions of G, we obtain $\big\{b, c, d\big\}$, for the abstract AF $\hat{G}$ we get the abstractly stable extensions $\big\{\{\hat{g}\}, \{\hat{g}, f\}\big\}$. Since atleast one of the abstractly stable extensions are spurious, we have a spurious abstract AF. Now, we can have a look at the subsets of the spurious extensions and check if every subset is spurious as well. When concretizing the subset a subset of $\{\hat{g}\}$, e.g. $\{a, c\}$ we obtain the AF $\hat{G}'$ depicted in **??**.



(a) Concrete AF



(b) Spurious abstract AF

Figure 4.10: Concrete and spurious abstract AF

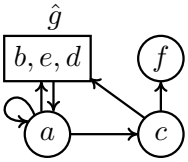


Figure 4.11: Concretized AF $\hat{G}'$

Now we compute the abstractly stable extensions of $\hat{G}'$ and obtain $\{\hat{g}, c\}$, which after expanding the cluster $\hat{g} = \{b, d\}$ we can map the abstract extension to the concrete extension. Thus, we found a faithful AF by concretizing a subset of a spurious stable extension which disproves the theory.

5 Experiments

5.1 Usage

TODO: Explain how program is called

5.2 Setup

TODO: My PC specs

5.3 DFS and BFS comparison

TODO: Compare DFS with BFS

5.4 Different AF types comparison

TODO: Compare Grid Based, Random, Level Approach

6 Related Works

7 Conclusion