



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, Assoc.Prof. Dipl.-Ing. Dr.techn. BSc.

Institute of Software Technology

Graz, October 12, 2024

Abstract

Argumentation frameworks (AFs) are a fundamental approach to formalize argumentative reasoning within artificial intelligence (AI). However, as the complexity of the argumentation frameworks increases, analyzing their structures and concluding something from them becomes challenging. Clustering is a method to decrease complexity by grouping multiple arguments into so-called clusters and simplifying the argumentation framework without drawing erroneous conclusions.

Since clustering argumentation frameworks is a relatively new concept, algorithmic approaches were not studied thoroughly yet. This thesis investigates the application of Boolean Satisfiability in creating and transforming clustered argumentation frameworks. We develop algorithms for determining if a given clustering is faithful, i.e., does not lead to erroneous conclusions, and to compute faithful clusterings by concretizations (extraction from clusters) until faithfulness is reached. We considered the argumentation semantics of stable semantics and the concepts of conflict-free sets and admissible sets. Moreover, we studied variants based on Breadth-First-Search (BFS) and Depth-First-Search (DFS) and refinements of our algorithms.

We provide an empirical evaluation of an implementation of our algorithms. The results indicate that depending on the structure and size of the argumentation framework, one procedure dominates the other. While the refinements do not show improvement for admissible and stable semantics, for conflict-free sets our refinement improves performance. For the two procedures of determining faithfulness or spuriousness, BFS and DFS, find their application in all of the examined semantics. While BFS is mostly more efficient on small argumentation frameworks (with a size of up to 10 arguments), DFS outperforms BFS on more complex argumentation frameworks.

Keywords: Artificial Intelligende, Argumentation Frameworks, Clustering, Semantics, Faithful, Spurious

Kurzfassung

Deutsche Kurzfassung der Abschlussarbeit

Acknowledgements

Thanks to everyone who made this thesis possible

Contents

1	Introduction	15
2	Background	19
2.1	Argumentation Frameworks	19
2.2	Clustering of Argumentation Frameworks	20
2.3	Boolean Satisfiability	23
3	Algorithm to obtain Faithful Clustering	25
3.1	Encodings for Semantics	25
3.2	Concretizing Singletons	28
3.3	Computation of Concretizer List	33
3.4	Algorithmic Approach to Compute Faithful Clusterings	36
3.5	Heuristics and Refinements	37
3.6	Disproved Assumptions	39
4	Implementations	41
4.1	Creating AFs	41
4.2	BFS and DFS Approach	44
4.3	Usage and Settings	45
5	Experiments	47
5.1	Generation of Input Instances	47
5.2	Setup	47
5.3	Faithful Spurious Check	48
5.3.1	Comparison of BFS and DFS	48
5.3.2	Comparison of Refinement and No-Refinement	53
5.3.3	Comparison of all Test-runs	55
5.4	Concretizing Arguments Program	56
5.4.1	Comparison of BFS and DFS	56
5.4.2	Comparison of Refinement and No-Refinement	62
5.4.3	Comparison of all Test-runs	63
6	Related Works	65
7	Conclusion	67

List of Figures

1.1	Concrete and abstract AF	17
1.2	Concretized AF \hat{G}'	17
2.1	AF G	19
2.2	AF \hat{G} clustered	21
2.3	Concrete and abstract AF	22
2.4	Concretized AF \hat{G}'	23
3.1	Abstract AF \hat{G}	26
3.2	Abstract AF \hat{G}	26
3.3	Abstract AF \hat{G}	27
3.4	Concrete and abstract AF	28
3.5	Concretized AF \hat{F}' after Step 1	29
3.6	Concretized AF \hat{F}' after Step 2	29
3.7	Concretized AF \hat{F}' after Step 3	30
3.8	Concretized AF \hat{F}' after Step 4	31
3.9	Concretized AF \hat{F}' after Step 5	32
3.10	Concrete and abstract AFs	33
3.11	Singletons depth with b as viewpoint	34
3.12	Concrete and abstract AF	36
3.13	Concrete and spurious abstract AF	40
3.14	Concretized AF \hat{G}'	40
4.1	Random-based approach with $Amount = 4$ and $p = 1$	41
4.2	Example AF generated with random-based approach	42
4.3	Grid-based approach with $Amount = 16$ and $p = 1$	42
4.4	Example AF generated with grid-based approach	43
4.5	Level-based approach with $Level = 2$, $Amount = 16$ and $p = 1$	43
4.6	Example AF generated with grid-based approach	43
4.7	BFS visualization	44
4.8	DFS visualization	44
5.1	Runtime of random-based AFs over conflict-free semantics	49
5.2	Runtime of random-based AFs over admissible semantics	49
5.3	Runtime of random-based AFs over stable semantics	49
5.4	Runtime of grid-based AFs over conflict-free semantics	50
5.5	Runtime of grid-based AFs over admissible semantics	50
5.6	Runtime of grid-based AFs over stable semantics	51

5.7	Runtime of level-based AFs over conflict-free semantics	51
5.8	Runtime of level-based AFs over admissible semantics	52
5.9	Runtime of level-based AFs over stable semantics	52
5.10	Runtime of BFS and DFS grouped by generator approach	53
5.11	Runtime of grid-based AFs over conflict-free semantics	54
5.12	Runtime of level-based AFs over conflict-free semantics	54
5.13	Performance comparison of Refinement and No-Refinement by Semantics	55
5.14	Solved test-runs depending on semantics	55
5.15	Runtime of random-based AFs over conflict-free semantics	57
5.16	Runtime of random-based AFs over admissible semantics	57
5.17	Runtime of random-based AFs over stable semantics	58
5.18	Runtime of grid-based AFs over conflict-free semantics	58
5.19	Runtime of grid-based AFs over admissible semantics	59
5.20	Runtime of grid-based AFs over stable semantics	59
5.21	Runtime of level-based AFs over conflict-free semantics	60
5.22	Runtime of level-based AFs over admissible semantics	60
5.23	Runtime of level-based AFs over stable semantics	61
5.24	Runtime of BFS and DFS grouped by generator approach	61
5.25	Runtime of grid-based AFs over conflict-free semantics	62
5.26	Runtime of level-based AFs over conflict-free semantics	63
5.27	Performance comparison of Refinement and No-Refinement by Semantics	63

List of Tables

2.1	Boolean logic operators	23
3.1	Combinations of $\{a, c, d, e\}$	35
5.1	Experiment Setup Specifications	48
5.2	test-runs Statistics spurious check mean runtime	56
5.3	test-runs Statistics concretize arguments mean runtime	64

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 [12]. 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 [11, 27].

Arguments can have many forms [32]. 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 [14]. Now we can define argumentation frameworks (AFs) and use them to evaluate conclusions [16]. 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 [13].

Semantics define subsets of arguments that have a certain relation to each other. Dung defined different semantics [13] 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 definition 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 in 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 [30]. 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 abstract 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 abstract 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 Figure 1.1(a). 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 Figure 1.1(b). 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 $stb(G) = \{\{d, e\}, \{b, d, f\}\}$.



Figure 1.1: Concrete and abstract AF

By computing the stable extensions of the abstract AF \hat{G} $stb(\hat{G}) = \{\{d\}, \{d, \hat{g}\}\}$, 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 Figure 1.2, which has the following stable extensions $stb(\hat{G}') = \{\hat{g}, d\}$. 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.

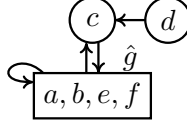


Figure 1.2: Concretized AF \hat{G}'

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 extent, 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 [15]. In fact, the complexity of proving faithfulness or spuriousness of an AF is \prod_2^P hard [30]. 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 abstract 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 abstract AF, respectively, the DFS variant iteratively computes abstract extensions of the abstract AF and verifies (non-)spuriousness of this extension.

- In regard of computing faithfulness or spuriousness, we developed different refinements for the corresponding semantics.
- Towards user-interaction, for a given AF and abstract 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 abstract AF until faithfulness is reached, since the extraction of the user-defined arguments may result in spurious reasoning.
- Furthermore, we provide an algorithm to transform a given spurious abstract AF into a faithful abstract AF by concretizing arguments into singletons.
- We implemented our algorithms in python3 with the SAT-Solver z3, created by Microsoft and provide the implementations in open-source.
- In an experimental evaluation, we tested the efficiency of the tool on the various algorithms and provided a deep insight on the impact of different approaches (BFS, DFS, refinements and no-refinements).

We provide an open-source implementation of the previously listed tools [7].

<https://github.com/p4s3r0/argumentation-framework-clustering>

Our tool can still solve most of the argumentation frameworks with a total argument size of 30. We can even go further, but it is highly dependent on the structure of the argumentation framework and the semantic.

2 Background

In this chapter we recall the required background of the thesis. We start with basic definitions of argumentation frameworks (AFs) in Section 2.1. Next we treat the clustering of AFs in Section 2.2 and finally we provide a brief overview of SAT-Solvers in Section 2.3.

2.1 Argumentation Frameworks

Argumentation frameworks were first formally described by Dung in 1995 [14]. 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 [18, 31]. But do also find their applications in other fields like natural language processing [5], trust and reputation systems [21], legal and medical reasoning [2], and even in game theory and strategic reasoning [26].

AFs are represented by a directed graph, where the nodes are an abstraction of the arguments A , and the arrows represent the attacks R .

Example 1. Let us define an AF $G = (A, R)$ with the arguments $A = \{a, b, c, d, e\}$ and the attacks $R = \{(a, b), (b, b), (a, c), (c, a), (c, d), (d, e), (e, d)\}$.

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



Figure 2.1: AF G

To be able to conclude something, out of an abstract AF, we need to define semantics. Semantics define subsets of arguments that satisfy semantic-specific rules. Dung already defined different semantics [13] 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. We define the set of conflict-free sets of an AF G by $cf(G)$.

Definition 1 ([13]). *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 2. *The conflict-free sets of the AF G in Figure 2.1 are \emptyset , $\{a\}$, $\{c\}$, $\{d\}$, $\{e\}$, $\{a, d\}$, $\{a, e\}$ and $\{c, e\}$.*

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

Definition 2 ([13]). *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 3. *The corresponding admissible sets of the AF G in Figure 2.1 are \emptyset , $\{a\}$, $\{c\}$, $\{e\}$, $\{a, d\}$, $\{a, e\}$ and $\{c, e\}$.*

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 ([13]). *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 $(a, b) \in R$.*

Example 4. *The stable extensions of the AF G in Figure 2.1 are $\{a, d\}$ and $\{a, e\}$.*

The specific rules of a semantics can also be defined via a Boolean formula [3]. Which then can be used to encode the AFs to be solvable with different solvers like Answer Set Programming (ASP) [6] 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 there exists, e.g., an admissible set containing a queried argument) under a specific argumentation semantics [15].

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*. These clustered AFs (i.e. abstract AFs) have semantics extensions, just as non-clustered AFs (i.e. concrete AFs), which we call abstract semantics extensions. 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 $\hat{R} = \{(\hat{h}, d), (d, e), (e, d), (\hat{h}, \hat{h})\}$. This AF can be represented as a directed graph as shown in Figure 2.2.

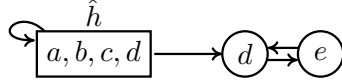


Figure 2.2: AF \hat{G} clustered

Since clusters can not be treated the exact same way as an argument, we need to refine the definitions of the semantics. 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 ([30]). *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 $\hat{a}, \hat{b} \in \text{singleton}(S)$ we have $(\hat{a}, \hat{b}) \notin \hat{R}$.*

Example 5. *The abstractly conflict-free sets of the abstract AF G in Figure 2.2 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 ([30]). *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 $\text{singleton}(\hat{a})$, then there is a $\hat{c} \in \hat{G}$ with $(\hat{c}, \hat{b}) \in \hat{R}$.*

Example 6. *The abstractly admissible sets of the abstract AF G in Figure 2.2 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 ([30]). *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 7. *The abstractly stable extensions of the abstract AF G in Figure 2.2 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: $R = \{(a, b), (b, b), (a, c), (c, a), (c, d), (d, e), (e, d)\}$. This AF can be represented as a directed graph shown in Figure 2.3(a).

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 $\hat{R} = \{(\hat{h}, e), (e, \hat{h}), (\hat{h}, \hat{h})\}$. 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 Figure 2.3(b).



Figure 2.3: Concrete and abstract AF

If we compare the stable extensions of the concrete AF G (e.g. $stb(G) = \{\{a, e\}, \{a, d\}\}$) with the abstractly stable extensions of the abstract clustered AF \hat{G} (e.g. $\hat{stb}(G) = \{\{\hat{h}\}, \{e\}, \{e, \hat{h}\}\}$), 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 $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{a, b\}$, $\{a, c\}$, $\{a, d\}$, $\{b, c\}$, $\{b, d\}$, $\{c, d\}$, $\{a, b, c\}$, $\{a, b, d\}$, $\{a, c, d\}$, $\{b, c, d\}$ and $\{a, b, c, d\}$. 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 corresponding attacks are $\hat{R}' = \{(d, \hat{h}), (d, e), (e, d), (\hat{h}, \hat{h})\}$.

With this definition we can build the concretized abstract graph \hat{G}' in Figure 2.4.

Every abstractly stable extension in Figure 2.4 (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.



Figure 2.4: Concretized AF \hat{G}'

2.3 Boolean Satisfiability

A SAT(isfability)-Solver is used to compute Boolean formulas in an efficient way [4]. 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. A model in this context is an assignment of truth values to the variables that satisfies a given Boolean formula. The Boolean formula is composed of different logical operators, i.e. negation (\neg), conjunction (\wedge), disjunction (\vee) and implication (\rightarrow). The corresponding truth tables are defined in Table 2.1.

Table 2.1: Boolean logic operators

(a) Negation

x	\neg
0	1
1	0

(b) Conjunction

x	y	\wedge
0	0	0
0	1	0
1	0	0
1	1	1

(c) Disjunction

x	y	\vee
0	0	0
0	1	1
1	0	1
1	1	1

(d) Implication

x	y	\rightarrow
0	0	1
0	1	1
1	0	0
1	1	1

SAT-Solvers do find there applications in various domains, e.g. in verification and validation of software and hardware [17, 28]. But also in AI and machine learning [19] and even in security [8, 20].

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 [9]. 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 [22]. 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 [24].

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 Section 3.1, where we define the Boolean encodings for the semantics. In Section 3.2 we explain, the concretization of a clustered argument. Next, in Section 3.3 we explain how the concretizer list (a list of clustered arguments which are transformed to singletons) is computed. The approach to compute faithful clustered AFs is then described in Section 3.4. And finally we state the heuristics and refinements in Section 3.5.

3.1 Encodings for Semantics

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 $\{a \in \hat{A} \mid \tau(a) = 1\}$, where 1 is the numerical equivalent to the Boolean *true*.

We further define a subset of arguments, denoted $\hat{A}_{Single} \subset \hat{A}$ consisting of the the singletons arguments of the the AF G , which effectively excludes each cluster.

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. We can now define the following formula.*

$$\varphi_{cf}(\hat{G}) = \bigwedge_{a \in \hat{A}_{Single}} \left(\bigwedge_{b: (b,a) \in \hat{R}, b \in \hat{A}_{Single}} \neg(a \wedge b) \right)$$

Example 8. *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 Figure 3.1. With the arguments $\hat{A} = \{a, b, c, d\}$ and the attacks $\hat{R} = \{(a, a), (a, b), (b, c), (d, a), (d, b), (d, c)\}$.*

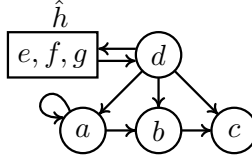


Figure 3.1: Abstract AF \hat{G}

With Definition 7 we obtain the following Boolean formula.

$$\begin{aligned} \hat{cf}(\hat{G}) = & \neg(b \wedge a) \wedge \neg(b \wedge d) \wedge \neg(a \wedge d) \wedge \neg(a \wedge a) \wedge \\ & \neg(c \wedge b) \wedge \neg(c \wedge d) \end{aligned}$$

The models (i.e sets of variables assigned to true) of $\hat{cf}(\hat{G})$ in this example are $\{\emptyset, \{b\}, \{c\}, \{d\}, \{\hat{h}\}, \{b, \hat{h}\}, \{c, \hat{h}\}, \{d, \hat{h}\}\}$.

If we compare the models of $\hat{cf}(\hat{G})$ with the conflict-free sets of \hat{G} , we can observe that they are equal. In fact, Definition 7 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 at least 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. We can now define the following formula.

$$\varphi_{adm}(\hat{G}) = \varphi_{cf}(\hat{G}) \wedge \bigwedge_{a \in \hat{A}_{Single}} (a \rightarrow \bigwedge_{b: (b,a) \in \hat{R}} (\bigvee_{c: (c,b) \in \hat{R}} c))$$

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 Figure 3.2. Where the arguments are $\hat{A} = \{a, b, c, d, e\}$ and the attacks $\hat{R} = \{(a, a), (a, b), (b, c), (d, a), (d, b), (d, d), (d, e), (e, d)\}$.

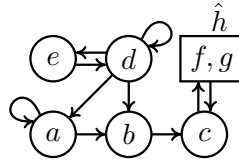


Figure 3.2: Abstract AF \hat{G}

By applying the formula defined in Definition 8 to the AF \hat{G} , we obtain the following Boolean expression.

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

The models of $adm(\hat{G})$ in this example are $\{\{\}, \{c\}, \{e\}, \{\hat{h}\}, \{c, e\}, \{c, \hat{h}\}, \{e, \hat{h}\}, \{c, e, \hat{h}\}\}$.

If we compare the models of $adm(\hat{G})$ with the admissible sets of \hat{G} , we can observe that they are equal. Indeed, Definition 8 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 previous formula for admissible sets.

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 a stable extension except if $\hat{A} = \emptyset$.

Definition 9. Let $\hat{G} = (\hat{A}, \hat{R})$ be an abstract AF. We can now define the following formula.

$$\varphi_{stb}(\hat{G}) = \varphi_{cf}(\hat{G}) \wedge \bigwedge_{a \in \hat{A}} (a \vee \bigvee_{b: (b,a) \in \hat{R}} b) \wedge \bigwedge_{a \in \hat{A}} ((a \wedge \neg b) \rightarrow (\bigwedge_{c: (a,c), c \in \hat{A}_{Single}} \neg c))$$

Example 10. Let us have a look at an example and define the AF $\hat{G} = (\hat{A}, \hat{R})$ to be the AF depicted in Figure 3.3. Where the arguments are $\hat{A} = \{a, b, c, d\}$ and the attacks $\hat{R} = \{(a, a), (a, b), (a, d), (b, a), (b, d), (c, b), (d, a)\}$.

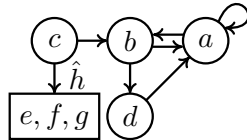


Figure 3.3: Abstract AF \hat{G}

If we apply the encoded formula defined in Definition 9 to the AF \hat{G} , we obtain the following Boolean expression.

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

The models of $\hat{stb}(\hat{G})$ in this example are $\{\{c, d\}, \{c, d, \hat{h}\}\}$.

By comparing the models of $\hat{stb}(\hat{G})$ with the stable sets of \hat{G} , we find that they are identical. Specifically, Definition 9 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 transformation is to extract 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. Since clustering does not preserve all the informations, i.e. the attacks directed towards and originating from the clustered arguments, the abstraction does lead to a loss of details. To be able to concretize argument, this needs to be compensated by providing additionally the concrete AF. With the concrete AF we can reconstruct the attacks of the arguments and transform the AF in such a way, as if it was primarily abstracted the same manner, but without the arguments intended to be concretized.

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 Figure 3.4, where we concretize the arguments a and b .



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}' with no attacks towards or originating from the singleton. If an argument is not part of a cluster, we ignore it and remove it from the list of arguments to be concretized. We do not consider attacks in this step since they depend on the concrete- and abstract AFs which we will consider on the next step. The resulting AF is depicted in Figure 3.5 and the pseudo-code in Algorithm 1.

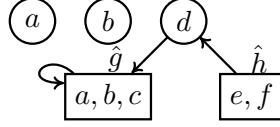


Figure 3.5: Concretized AF \hat{F}' after Step 1

Algorithm 1 Concretizing Singletons Pseudocode Step 1

Require: $A : AF(a_1, r_1)$ \triangleright Abstract Clustered AF
Require: $e : list(Arguments)$ \triangleright Concretizer List
1: $N \leftarrow A$ $\triangleright N =$ 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 Figure 3.6 and the pseudo-code in Algorithm 2. The pseudo code iterates over all the arguments to be concretized and adds the missing attacks from the clusters to the arguments and vice versa.

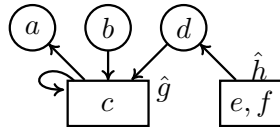


Figure 3.6: Concretized AF \hat{F}' after Step 2

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                                ▷ Add attacks (argument, cluster)
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                                ▷ Add attacks (cluster, argument)
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 Figure 3.7 and the pseudo-code in Algorithm 3.

To add further explanation of the pseudo-code, we split the code into three parts. All of them are encased in a iteration over all the attacks of the abstract AF. In each step we cover a different case of the entities from the current attack (a, b) . In the first part, we check if both entities a and b are clusters. If so, we check if any clustered argument from the attacker cluster a attacks any clustered argument of the cluster b in the concrete AF. If no attack is found, we remove the current attack (a, b) from the concretized AF. The second part covers the case, when the attacker a is a singleton and the defender b is a cluster. In this case, we check if a attacks any of the clustered arguments from b in the concrete AF. If this does not hold, we remove the attack (a, b) from the concretized AF. And finally, the third part covers the case, if the attacker a is a cluster and the defender b is a singleton. Here we do the same as in the second part, but with a and b reversed.

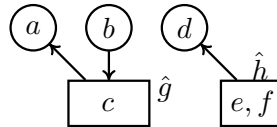


Figure 3.7: Concretized AF \hat{F}' after Step 3

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

```
1: for  $r_i$  in  $A_{r_1}$  do
2:   if  $r_i.defender$  is cluster and  $r_i.attacker$  is cluster then ▷ Part 1
3:     for  $a_i$  in  $A[r_i.attacker]$  do
4:       if  $!(a_i$  attacks any of  $C[r_i].defender)$  then
5:          $N.removeAttack(r_i)$ 
6:         break
7:       end if
8:     end for
9:   end if
10:  if  $r_i.defender$  is cluster then ▷ Part 2
11:    if  $!(r_i.attacker$  attacks any of  $C[r_i].defender)$  then
12:       $N.removeAttack(r_i)$ 
13:      continue
14:    end if
15:  end if
16:  if  $r_i.attacker$  is cluster then ▷ Part 3
17:    if  $!(r_i.defender$  defends against any  $C[r_i].attacker)$  then
18:       $N.removeAttack(r_i)$ 
19:      continue
20:    end if
21:  end if
22: 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 towards or originating 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 Figure 3.8 and the pseudo-code in Algorithm 4. To explain the pseudo-code a bit further, we iterate over all the arguments which need to be concretized and check their direct defender and attacker from the concrete AF and then add the missing attacks to the concretized AF.

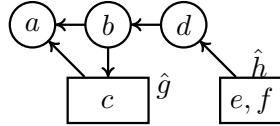


Figure 3.8: Concretized AF \hat{F}' after Step 4

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

```
1: for  $e_i$  in  $e$  do
2:   for  $a_i$  in  $C[e_i].attacks$  do
3:     if  $a_i$  is singleton and  $(e_i, a_i)$  not in  $r_2$  then
4:        $N.addAttack((e_i, a_i))$ 
5:     end if
6:   end for
7:   for  $a_i$  in  $C[e_i].defends$  do
8:     if  $a_i$  is singleton and  $(a_i, e_i)$  not in  $r_2$  then
9:        $N.addAttack((a_i, e_i))$ 
10:    end if
11:  end for
12: 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 Figure 3.9 and the pseudo-code in Algorithm 5.

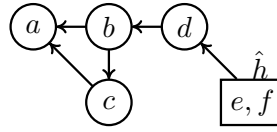


Figure 3.9: Concretized AF \hat{F}' after Step 5

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

```
1: for  $c_i$  in  $N.clusters$  do
2:   if  $c_i.argAmount == 1$  then
3:      $c_i \leftarrow Singleton$ 
4:   else if  $c_i.argAmount == 0$  then
5:      $N.remove(c_i)$ 
6:   end if
7: end for
```

3.3 Computation of Concretizer List

When talking about clustering AFs, faithfulness is an important property. If an AF is spurious, we find at least one extension, which cannot be mapped to a concrete extension. Based on the spurious extensions, we try to concretize the arguments of 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 dependent on the size of the set. We use a heuristic 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 extensions. As observed empirically, the computation intensity of the algorithm is highly dependent on the amount of attacks. If the arguments of the spurious set have many attackers or are attacking a lot of other arguments, we observe an exponential growth on the execution time.

Let us have a look at an example to demonstrate how the concretizer list is computed. A concrete AF G is defined in Figure 3.10(a) and a corresponding abstract AF \hat{G} in Figure 3.10(b).

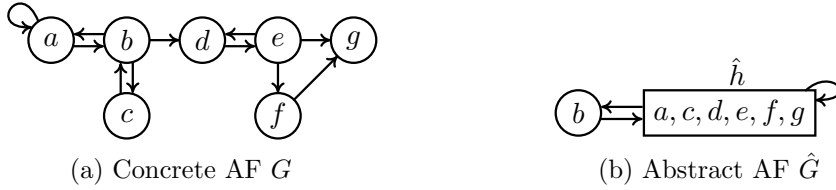


Figure 3.10: Concrete and abstract AFs

If we have a look at the stable extensions of the concrete AF G , i.e. $stb(G) = \{\{b, c\}\}$ and at the abstractly stable extensions of the abstract AF \hat{G} , i.e. $\hat{stb}(\hat{G}) = \{\{b, \hat{h}\}, \{\hat{h}\}, \{b\}\}$, 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 $\{b\}$.

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 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 Figure 3.11 depicted in dotted line style and the attacker with the same depth, depicted in dashed line style. The pseudo-code of this procedure is listed in Algorithm 6. 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), then the lower depth is chosen as the representative.

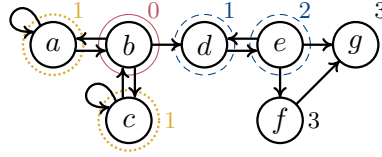


Figure 3.11: Singletons depth with b as viewpoint

Algorithm 6 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	

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.

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

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

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

to remove once again to minimize the memory consumption and reduce the amount of faithful 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 7.

Algorithm 7 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

- 1: $C \leftarrow$ combinations of N with $range(1, len(N) - 1)$ ▷ Combination List
- 2: **for** ca_i in ca **do** ▷ Parameter Arguments to be concretized
- 3: **for** c_i in C **do**
- 4: $c_i.append(ca_i)$
- 5: **end for**
- 6: **end for**
- 7: $C.deduplicate()$
- 8: **for** s_i in C **do** ▷ Remove clusters
- 9: **for** a_i in s_i **do**
- 10: **if** $\hat{G}[a_i]$ is cluster **then**
- 11: $s_i.remove(a_i)$
- 12: **end if**
- 13: **end for**
- 14: **end for**
- 15: **return** $s.sortBySize()$

3.4 Algorithmic Approach to Compute Faithful 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 corresponding concrete AFs. Thus, we need to transform the clustered AF, s.t. only correct solutions can be drawn, 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 Section 3.3. 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, then 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 Algorithm 8.

Example 11. 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 $R = \{(a, a), (a, c), (b, a), (c, b), (c, d), (d, c), (e, b), (b, e), (f, f)\}$, depicted in Figure 3.12(a). When clustering the arguments $\{a, b, f\}$ we obtain the abstract AF $\hat{G} = (\hat{A}, \hat{R})$ depicted in Figure 3.12(b). 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 $\{a, b, f\}$.



Figure 3.12: Concrete and abstract AF

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

Algorithm 8 Compute Faithful Clusters

Require: $G : AF(a_1, r_1)$ ▷ Concrete AF
Require: $\hat{G} : AF(a_2, r_2)$ ▷ Abstract AF

- 1: $s_c \leftarrow \text{computeSemanticsExtension}(G)$
- 2: $s_a \leftarrow \text{computeSemanticsExtension}(\hat{G})$
- 3: $sp \leftarrow \text{computeSpuriousSemantics}(s_c, s_a)$
- 4: **if** $sp.length == 0$ **then**
- 5: **return** \hat{G}
- 6: **end if**
- 7: $conc \leftarrow \text{computeConcretizerList}(sp)$
- 8: **for** set_i in sp **do**
- 9: $\hat{G}' \leftarrow \text{concretize}(\hat{G}, set_i)$
- 10: $s_a \leftarrow \text{computeSemanticsExtension}(\hat{G}')$
- 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 to improve the faithful/spurious check of two AFs for admissible and stable semantics and for conflict-free the general approach of finding semantics sets. 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 a refinement 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 definition of conflict-free sets. 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 . With this adaption we can speed up the process of generating conflict-free sets, which is also used in the faithful/spurious check.

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 spuriousness or faithfulness by adding the negated admissible formula for the concrete AF. This removes the faithful admissible sets, which are composed of only singletons and can therefore 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 at least 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 Definition 8, 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.

$$\begin{aligned} \overline{cf}(F) &= \bigvee_{a \in A_{Single}} \left(\bigvee_{b: (b,a) \in R, b \in A_{Single}} (a \wedge b) \right) \\ \overline{def}(F) &= \left(\bigvee_{a \in A_{Single}} \left(a \wedge \bigvee_{b: (b,a) \in R} \left(\bigwedge_{c: (c,b) \in R} \neg c \right) \right) \right) \end{aligned}$$

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 for showing spuriousness or faithfulness drastically, without losing crucial information. The eliminated abstract extensions are simply the extensions, which can be directly mapped to the concrete stable extensions. The remaining extensions that need to be computed are either spurious extensions, or extensions which have at least 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 Definition 9, 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 imply 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 attacking the argument is not in the abstractly stable extension.

$$\begin{aligned}\overline{cf}(F) &= \bigvee_{a \in A_{Single}} \left(\bigvee_{b: (b,a) \in R, b \in A_{Single}} (a \wedge b) \right) \\ \overline{att}(F) &= \bigvee_{a \in A} (\neg a \bigwedge_{b: (b,a) \in R} \neg b) \\ \overline{con}(F) &= \bigvee_{a \in A} \left((a \bigwedge_{b: (b,a) \in R} \neg b) \wedge \left(\bigvee_{c: (a,c), c \in A_{Single}} c \right) \right)\end{aligned}$$

3.6 Disproved Assumptions

When developing the tool, we came up with different theories. Some theories were discarded immediately, some were implemented (recall Section 3.5) and others had to be proven wrong. This section describes the most promising assumptions 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 $R = \{(b, a), (d, a), (a, d), (a, c), (c, e), (c, f), (d, f)\}$ depicted in Figure 3.13(a). 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 $\hat{R} = \{(\hat{g}, \hat{g}), (\hat{g}, f)\}$ depicted in Figure 3.13(b).

When computing the stable extensions of G , we obtain $\{b, c, d\}$, for the abstract AF \hat{G} we get the abstractly stable extensions $\{\{\hat{g}\}, \{\hat{g}, f\}\}$. Since at least 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 Figure 3.14.



Figure 3.13: Concrete and spurious abstract AF

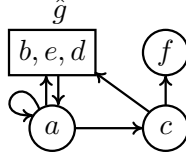


Figure 3.14: Concretized AF \hat{G}'

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.

4 Implementations

In this chapter we dive into the implementation part and discuss different approaches. First, we specify how the AFs we run the experiments on were created in Section 4.1. Here we describe three different methods, with their advantages and disadvantages. Next we explain two different settings to optimize the spurious/faithful check in Section 4.2. In ?? we explain how we generated the semantics extensions and finally, we discuss disproved assumptions in Section 3.6.

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 AFs, which are typically not similar to real-world problems, the grid-based approach has more structure. The level-based approach has even more structure and assures that we can not derive too 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. Basically, if we take a look at Figure 4.5 we can see a graph with potential attacks depicted with dotted arrows. Every potential attack has a probability of `<p>` to be an actual attack of the generated AF.

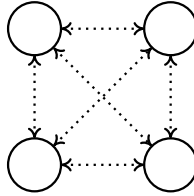


Figure 4.1: Random-based approach with $Amount = 4$ and $p = 1$

Random-based generated AFs have the property (depending on the probability value) of being hard 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 Figure 4.2

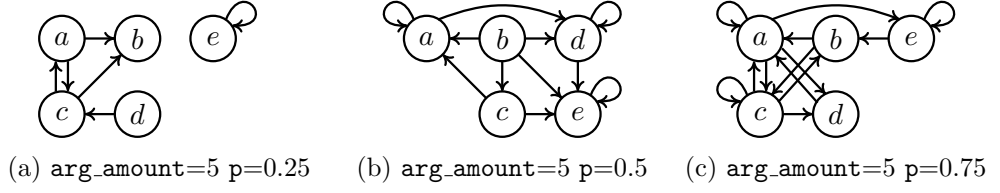


Figure 4.2: Example AF generated with random-based approach

Grid-based Next we are going to discuss the grid-based approach. The arguments for the script are $\langle \text{arg_amount} \rangle$, being the amount of arguments the AF has and $\langle p \rangle$, 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 an $n \times n$ grid, with n being equal to $\lfloor (\sqrt{\langle \text{arg_amount} \rangle}) \rfloor$. An example grid can be seen in Figure 4.3.

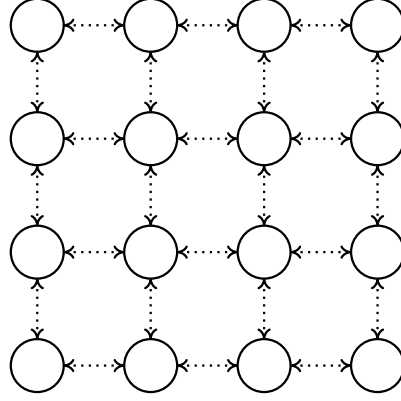


Figure 4.3: Grid-based approach with $\text{Amount} = 16$ and $p = 1$

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 Figure 4.4.

Level-based The last algorithm to create concrete AFs we provide, is the level-based approach. The arguments for this script are $\langle \text{arg_amount} \rangle$, $\langle \text{level} \rangle$ and $\langle p \rangle$. Same as for the grid-based and random-based approach, $\langle \text{arg_amount} \rangle$ defines how many arguments the computed AF has. The $\langle \text{level} \rangle$ argument restricts the height of the grid to the provided value and $\langle p \rangle$ is again the probability that an attack between to

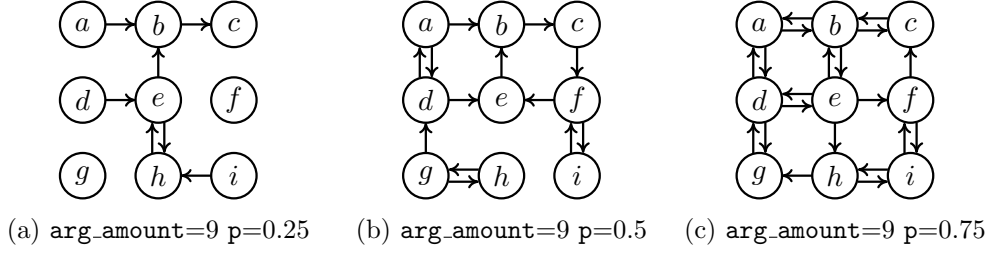


Figure 4.4: Example AF generated with grid-based approach

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 use a $\langle \text{level} \rangle \times n$ grid. In this context, n is equal to $\lceil \langle \text{arg_amount} \rangle / \langle \text{level} \rangle \rceil$. An example grid is depicted in Figure 4.5.

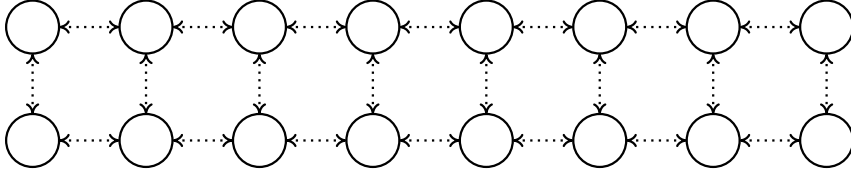


Figure 4.5: Level-based approach with $Level = 2$, $Amount = 16$ and $p = 1$

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 $\min(\langle \text{level} \rangle + 1, 4)$ 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 Figure 4.6

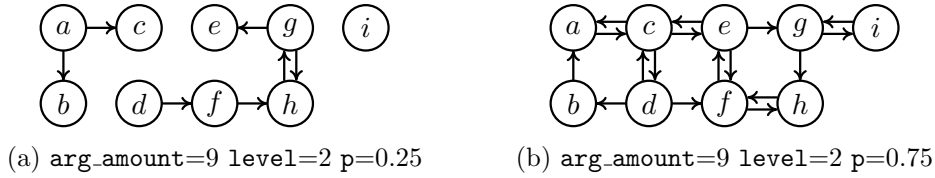


Figure 4.6: Example AF generated with level-based approach

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 it 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.

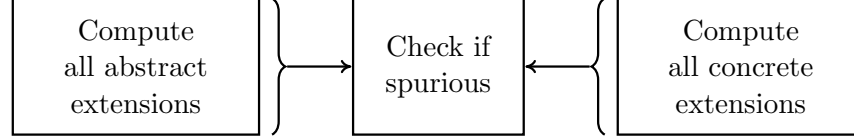


Figure 4.7: BFS visualization

The BFS approach, visualized in Figure 4.7 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 Figure 4.8. 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.

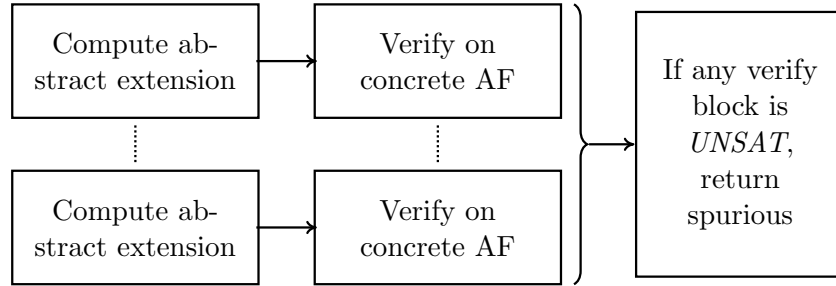


Figure 4.8: DFS visualization

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 Usage and Settings

We implemented the previously mentioned encodings and algorithm into the software called ClustArg. The tool uses Python as primary programming language and the library z3 for SAT-Solving. The software is available at

<https://github.com/p4s3r0/argumentation-framework-clustering>

under the MIT license. We provide four different applications, e.g. SETS, CHECK, CONCRETIZE and FAITHFUL. Additionally, we need to add the corresponding data via flags, with which we can also change the behaviour of the program.

Computing extensions ClustArg can compute all semantics extensions of a given AF. This program can be called with the SETS argument as follows

```
python3 main.py SETS <path_af>
```

The argument <path_af> is the path to the AF, of which the semantics extensions should be computed. To set the semantics of the extension check the additional flags paragraph below.

Spurious/Faithful check To check if an abstract AF is faithful or spurious, the program is called with the CHECK argument. Followed by the abstract AF and the concrete AF.

```
python3 main.py CHECK <path_abs_af> -c <path_con_af>
```

The second argument <path_abs_af> is the path to the abstract AF. Next, we add the path to the concrete AF at the -c <path_con_af> variable. Furthermore, we can specify the semantics and the algorithm (i.e. BFS or DFS) which is explained in the additional flags paragraph.

Compute faithful AF If we have a spurious AF and want to compute a faithful abstract AF with only minor transformations, we want to call the program with the FAITHFUL argument.

```
python3 main.py FAITHFUL <path_abs_af> -c <path_con_af> -s <semantics>
```

The next arguments are again the path to the abstract AF, which is then followed by the path to the concrete AF. Same as for the previously mentioned programs, the semantics is set with the -s <semantics> flag and the algorithmic approach with the -a DFS or -a BFS flag.

Concretizing arguments If we want to have more control over which arguments are concretized to obtain faithfulness, the program needs to be called with the **CONCRETIZE** flag.

```
python3 main.py CONCRETIZE <path_abs_af> -c <path_con_af> -p <conc_args>
```

In addition to the paths of the abstract and concrete AF, we can specify a list of arguments, separated by spaces, to be concretized. The list of arguments **<conc_args>** needs to be directly after the **-p** flag. Additionally we can use positional arguments like **-s <semantics>** to specify the semantics or **-a <alg>** to determine if we want to use BFS or DFS.

Additional Flags To have full control of the inner workings of the program and the output printed to the user, we implemented additional flags. The first flag is the semantics flag.

-s <semantics>

This argument determines the semantics we are operating on. It is optional for every program and has one of the the following values: **CF** for conflict-free semantics which is also the default value, **AD** for admissible semantics or **ST** for stable semantics.

Next, when executing the program to concretize arguments from a cluster, we need a list of the arguments.

-p <conc_args>

The listing of the arguments which need to be concrete is specified after the **-p** tag and is specified in a joint list separated by spaces.

Furthermore we can alter the faithful/spurious check algorithm with the **<alg>** argument.

-a <alg>

Here only two possible values can be selected, i.e. the default value **BFS** and the most of the time more efficient algorithm **DFS**.

If the refinement of the current semantics is not desired, it can be disabled with the following flag.

-noref

We also implemented a visualization of the AFs, which can be triggered by using the the sollowing flag.

-vis

The visualisation will show the abstract AF, the concrete AF if it is part of the program call and finally it will show the resulting AF. To "spectate" the current computation and have more insight to the algorithm, we can add the **-verbose** flag. This will print interim findings to the user directly while calculating.

5 Experiments

In this chapter, we discuss the experiment that was conducted. In Section 5.1, we describe how the AF instances were generated. Then, we describe the setup and environment in which the experiment was executed in Section 5.2. In ??, we show and discuss the faithful/spurious check results according to the two approaches, i.e. BFS and DFS. In ??, we will do the same for the concretizing arguments program. Finally, we investigate the efficiency of the applied refinements for both programs in ??.

5.1 Generation of Input Instances

We generated the experiment instances with the previously mentioned scripts in Section 4.1. To have a large variety of AF inputs, we generated 25 tests per generator approach, i.e., random-based, grid-based, and level-based. The 25 tests per generator approach are grouped into five different sizes, which are 10, 15, 20, 25, and 30. Since the AF generator uses a certain probability to create an attack, it is connected to a certain randomness. This randomness assures the versability of AFs in the same argument amount group. The probability value was set to 0.5, equivalent to a 50% chance that an attack between two arguments might occur. Finally, the abstraction of the concrete AFs was realized with the clustering script described in Section 4.1.

The arguments to be concretized for the concretization algorithm were chosen in the AF generation script. For every instance, we chose a random amount (between 1 and the total amount of arguments in the cluster) of arguments, which had to be concretized. The randomness of generation is essential to ensure that we did not pick AFs that contributed to a misleading runtime by picking favorable AFs.

5.2 Setup

The experiment was conducted in a personal computer environment. The exact specification can be seen in Table 5.1. Generally, we used an AMD CPU with 6 cores and 12 Threads running the *x86_64* architecture. We had a total RAM of 16GB, which sometimes was not enough, and we had to add another 16GB of SWAP memory. Every test run was conducted on the same setup, running Linux as the primary OS. In total, 75 different AFs were tested under various configurations. These included the usage of the BFS and DFS algorithm, the three covered semantics, e.g., conflict-free, admissible, and stable, and running with and without the refinements. A separate Python script managed the tests by running every test instance as a subprocess and setting the timeout to 300s.

Table 5.1: Experiment Setup Specifications

CPU-Model-Name	AMD Ryzen 5 3600
CPU-Cores	6
CPU-Threads	12
CPU-Architecture	x86_64
CPU-Speed	4.2 GHz
RAM	Vengeance PRO 16 GB
SWAP	16 GB
Operating System	Ubuntu 24.04

5.3 Faithful Spurious Check

This section presents the data collected from the program, which checks if an abstract AF is faithful or spurious. We compare BFS and DFS in Section 5.3.1. For these tests, we split plots into three categories, i.e., the three AF generator procedures (random-based, grid-based, and level-based). All three categories are split again on the specific semantics, i.e., conflict-free, admissible and stable. Then, we checked how much impact the refinements had on the corresponding semantics. Finally, we packed all the runtimes into one plot to show the influence of the different generator approaches and created a table to show the impact of the number of arguments from the AF. Next, we did the same for the concretizing arguments program in Section 5.4.

5.3.1 Comparison of BFS and DFS

We implemented two different approaches to check for faithfulness, e.g. BFS and DFS. Both of them were tested and showed different results according to the AFs generation procedure. First of all, let us have a look at the runtime of the random-based generated test-runs depicted in Figure 5.1, Figure 5.2 and Figure 5.3. Note, that the testrun numeration does not begin at 1 because all the test-runs below 40 were trivial and could be solved in under a second. We assume that since the random-based approach has more attacks than the other approaches, the amount of semantics extensions are less. Thus, BFS and DFS do not differ that much.

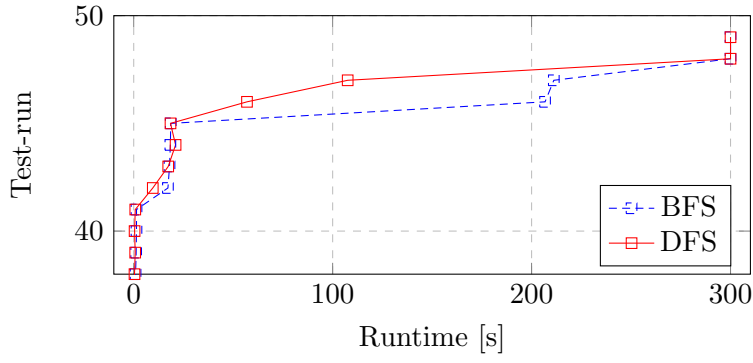


Figure 5.1: Runtime of random-based AFs over conflict-free semantics

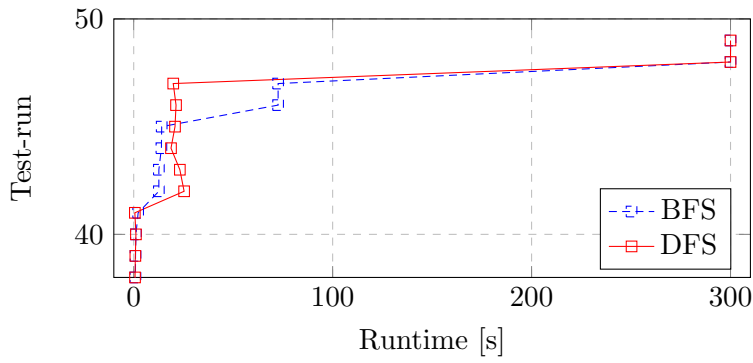


Figure 5.2: Runtime of random-based AFs over admissible semantics

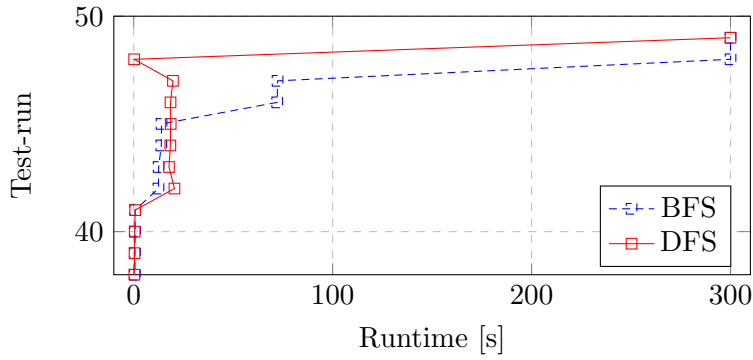


Figure 5.3: Runtime of random-based AFs over stable semantics

The difference of BFS and DFS increases when we change the AF generation approach to the grid-based procedure. In the plots depicted in Figure 5.4, Figure 5.5 and Figure 5.6 we can observe, that DFS outperforms the BFS algorithm. We speculate, that due to the grid-based AFs generation procedure, we created AFs with way more semantics

extensions. Since BFS needs to compute every extension to proof spuriousness and DFS checks after every computed extension for spuriousness, we obtain a better runtime for DFS if the AF is spurious. Since all the testcase instances were computed randomly, the chance of generating a spurious AF is more likely and thus favor the DFS algorithm.

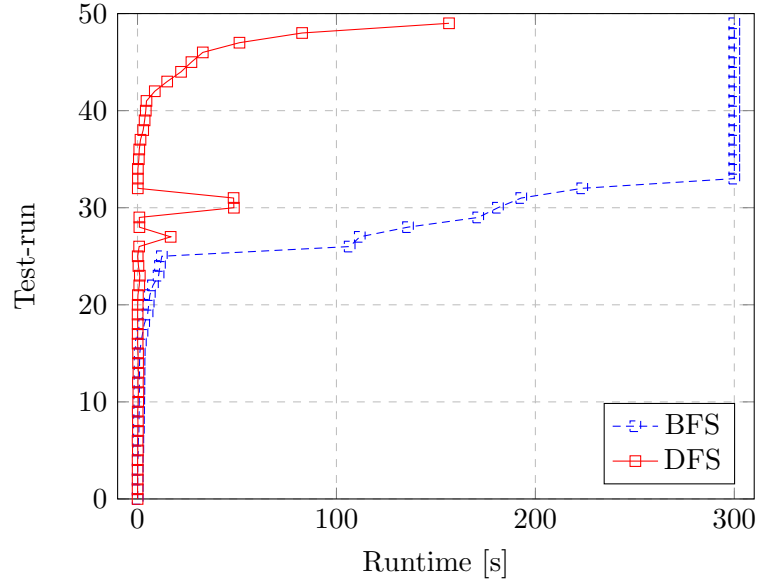


Figure 5.4: Runtime of grid-based AFs over conflict-free semantics

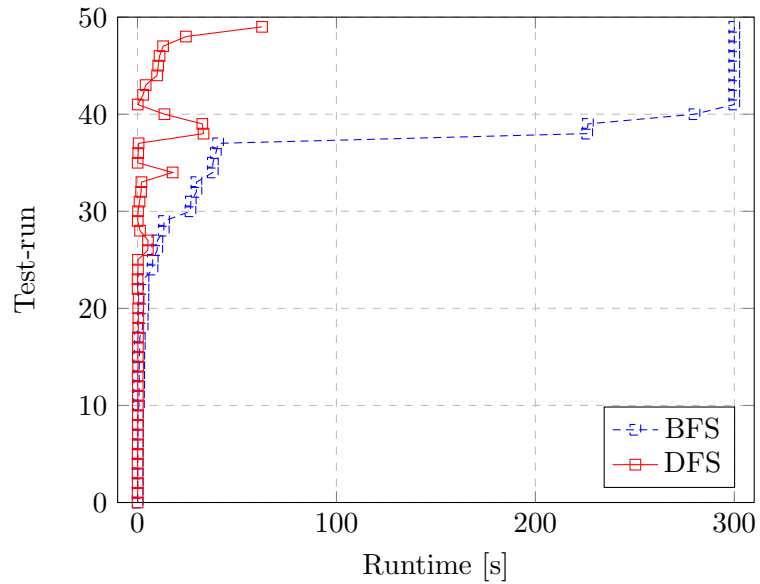


Figure 5.5: Runtime of grid-based AFs over admissible semantics

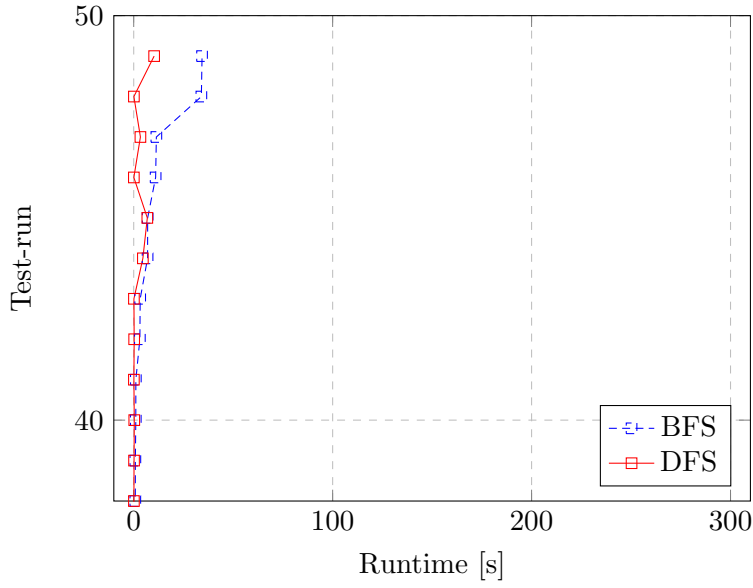


Figure 5.6: Runtime of grid-based AFs over stable semantics

The difference increases even further when changing the AF generator approach to the level-based procedure. As we can see in Figure 5.7, Figure 5.8 and Figure 5.9, DFS dominates BFS especially in the conflict-free and admissible semantics. This is due to the bigger size of the semantics extensions. Recall, that the conflict-free sets are a subset of the admissible sets and the admissible sets are a subset of the stable extensions.

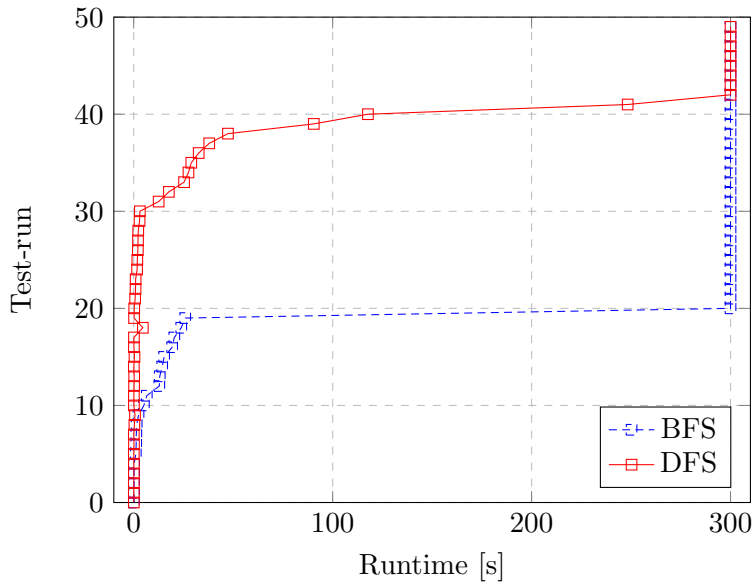


Figure 5.7: Runtime of level-based AFs over conflict-free semantics

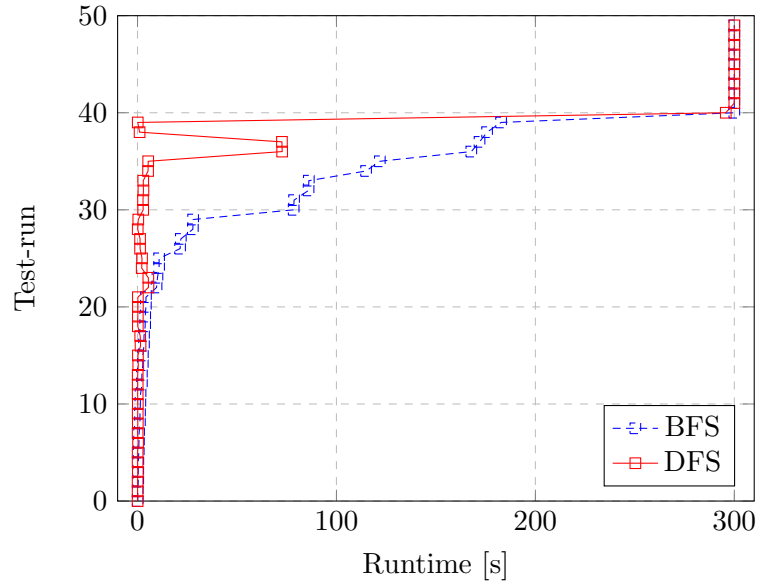


Figure 5.8: Runtime of level-based AFs over admissible semantics

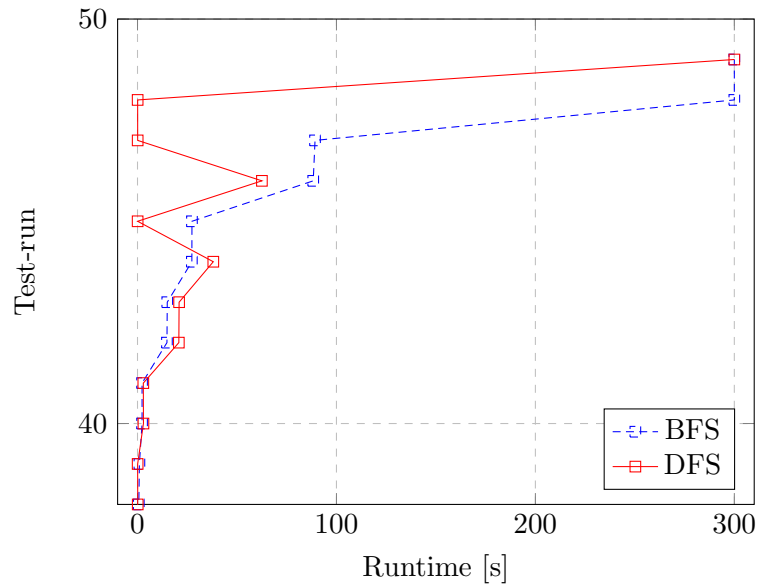


Figure 5.9: Runtime of level-based AFs over stable semantics

Finally, we arranged the data into a single plot from all the generator approaches in Figure 5.10. Here we can see that DFS dominates in almost all of the test-runs the BFS approach and in the few cases where BFS is more efficient, it is by a small factor. Especially with the AFs generated with the level-based procedure we observe that DFS is way more efficient. This is due the fact, that on level-based AFs we have far more

extensions than on random-based and computing all of them with the BFS algorithm leads to a bad runtime. On the other hand, random-based AFs have far less extensions due to fewer attacks. Thus, BFS can be faster than DFS, since it does not have to deal with as many context switches.

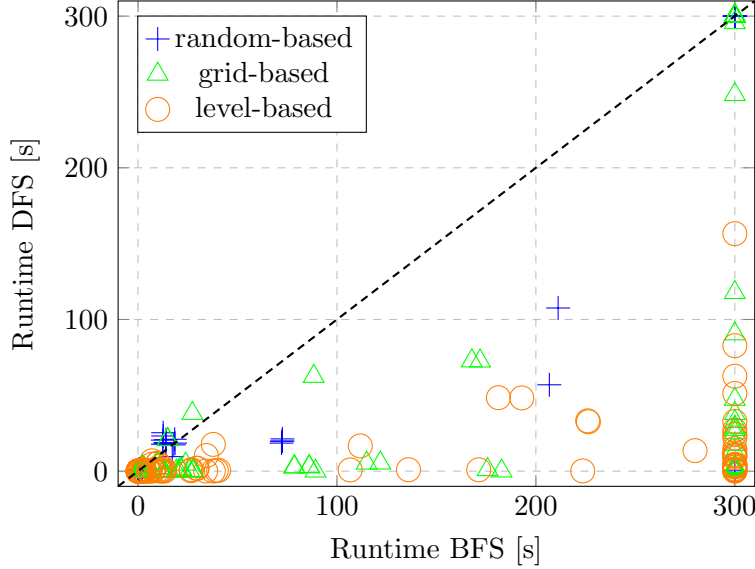


Figure 5.10: Runtime of BFS and DFS grouped by generator approach

5.3.2 Comparison of Refinement and No-Refinement

We implemented different refinements for each semantics. As data has shown, the refinements made for conflict-free semantics have a big impact as shown in Figure 5.11 and Figure 5.12. Especially for AFs with many conflict-free sets, the refinement reduces the runtime significantly. We got some test-runs for the grid-based AFs, where without the refinement we obtain a better result. We hypothesize, that the observed outliers may result from the DFS algorithm finding a spurious extension. Recall, that the refinement for the conflict-free sets is to extract all the subsets from a computed conflict-free set. If all the subsets are faithful, we introduce many faithful/spurious check which lead to an increase of runtime, that the DFS algorithm without the refinement does not face. The same outlier can be seen the other way around in the grid-based plot at test-run 25. Here, the refinement extracted a subset of a computed conflict-free set, which was spurious. Without the refinement, no spurious set was found within the timeout. Besides conflict-free, the refinements made for admissible and stable did only have a minor impact. We speculate, that the SAT-Solver discarded the refinement due to optimization reasons or the overhead of increasing the Boolean formula lead to no significant improvement.

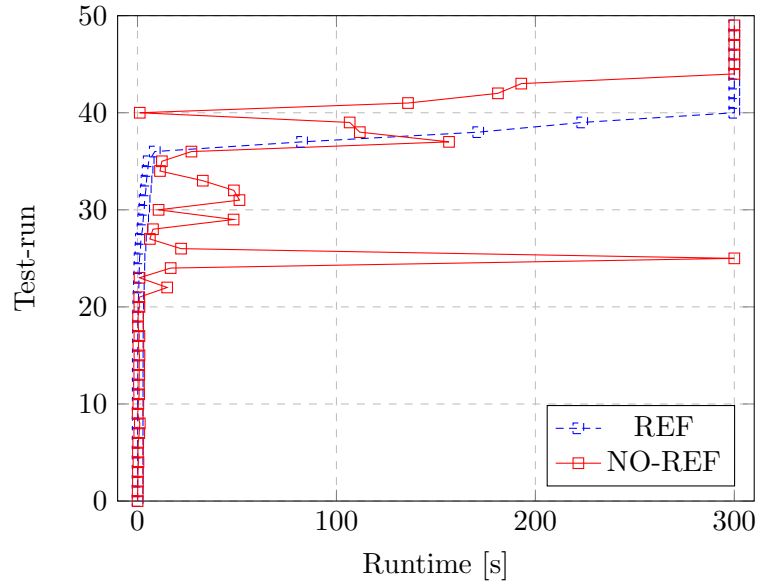


Figure 5.11: Runtime of grid-based AFs over conflict-free semantics

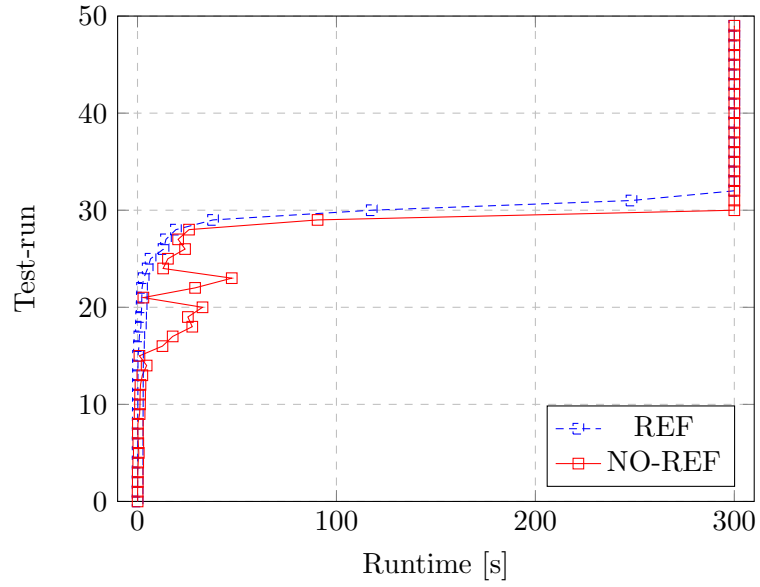


Figure 5.12: Runtime of level-based AFs over conflict-free semantics

In Figure 5.13 we can see the performance of the refinements against no refinements. For admissible and stable, there is no significant difference except some outliers. We hypothesize, that due to the refinement, we pushed the SAT-Solver into a certain direction (like changing the seed), where he found some lucky spurious extensions. Nevertheless, most of the time the refinements made for non-conflict-free semantics had no fundamen-

tal impact on the runtime. On the other hand, the refinement made for conflict-free semantics had a significant improvement on the runtime, besides some outliers.

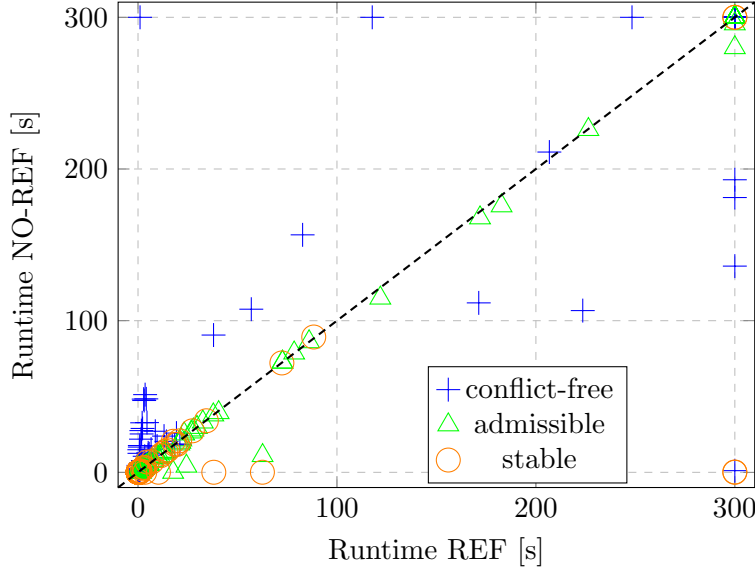


Figure 5.13: Performance comparison of Refinement and No-Refinement by Semantics

5.3.3 Comparison of all Test-runs

In this section we compare the data over all test-run instances, specifically how many test-runs did not run in to a timeout depending on the semantics and the mean runtime of the passed test-cases. In Figure 5.14 we can observe that the most solvable semantics is the stable semantics. Followed by the admissible semantics and the least solved test-runs has the conflict-free semantics. The reason for this is the number of semantic extensions of the AFs. If an AF has many semantic extensions, the runtime of the faithful check increases.

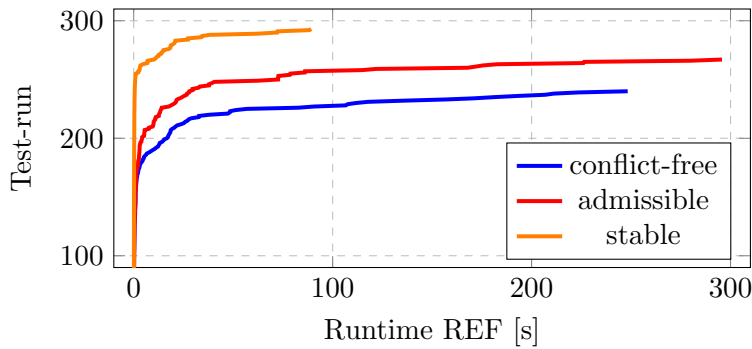


Figure 5.14: Solved test-runs depending on semantics

The data was arranged in a tabular format to enable a more comprehensive comparison from a higher-level viewpoint. We computed the mean value of the test-runs depending on the AFs number of arguments and semantics. Furthermore, we excluded the runtime from the test runs that ran into a timeout and stated the exact amount. In Table 5.2 we can observe that with the increase in size of the AF, the runtime of the faithful/spurious check increases as well. The same holds for the amount of timeouts.

Table 5.2: test-runs Statistics spurious check mean runtime

arguments amount	cf [s] (timeout)	adm [s] (timeout)	stb [s] (timeout)
10	0.29 (0/60)	0.20 (0/60)	0.09 (0/60)
15	3.91 (0/60)	0.81 (0/60)	0.13 (0/60)
20	28.69 (13/60)	6.56 (0/60)	0.34 (0/60)
25	13.85 (20/60)	37.69 (0/60)	4.72 (0/60)
30	35.41 (26/60)	33.01 (32/60)	10.97 (6/60)

5.4 Concretizing Arguments Program

This section presents the data collected from the concretizing arguments program. We again split up the test-runs to be able to compare the impact of BFS and DFS in Section 5.4.1, and the comparison on the runtime of using the semantics-specific refinements is shown in Section 5.4.2. The comparison of BFS and DFS is again split into the three AF generator approaches. Each plot depicts the runtime of BFS and DFS with the generator procedure and the corresponding semantics. The same is done for the refinement comparison. Furthermore, we also show the correlation of execution runtime and the size of the AF with a table and how many instances of test-runs were solvable according to the semantics.

5.4.1 Comparison of BFS and DFS

Since the program for concretizing arguments uses the faithful/spurious check multiple times, we can compare the impact on the runtime of the BFS and DFS approaches. We start with the first AF generator procedure, i.e., the random-based approach. Here we created three plots: in Figure 5.15 the conflict-free semantics is shown, in Figure 5.16 we depicted the admissible semantics and in Figure 5.17 the stable semantics. For these test-cases, the DFS approach dominates the BFS approach in every instance. We hypothesize that this is the repeated check of spurious AFs, which is more favorable for DFS. Furthermore, BFS does not only stand no chance against DFS, but most of the instances do not terminate before the timeout. This is especially critical for the admissible and stable semantics.

For the runtime on the DFS approach, the AFs created with the random-based procedure were trivial to solve, but ran in to a timeout when the complexity of the AFs increased.

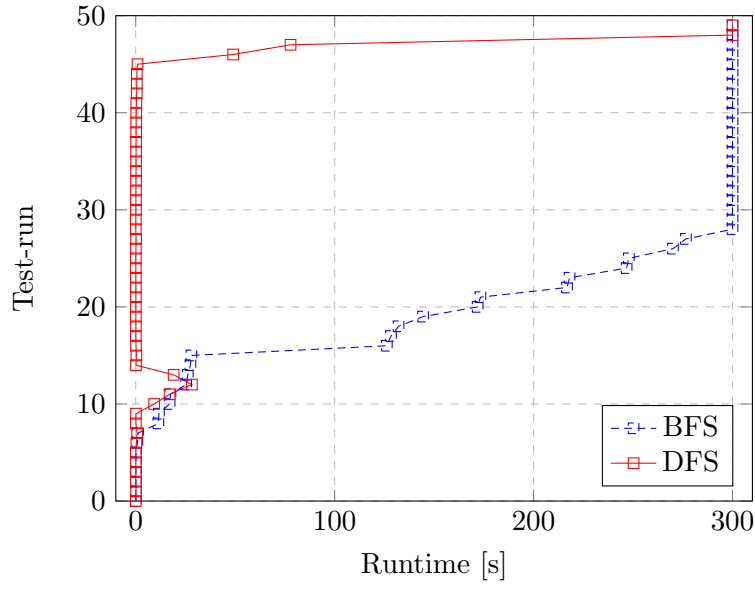


Figure 5.15: Runtime of random-based AFs over conflict-free semantics

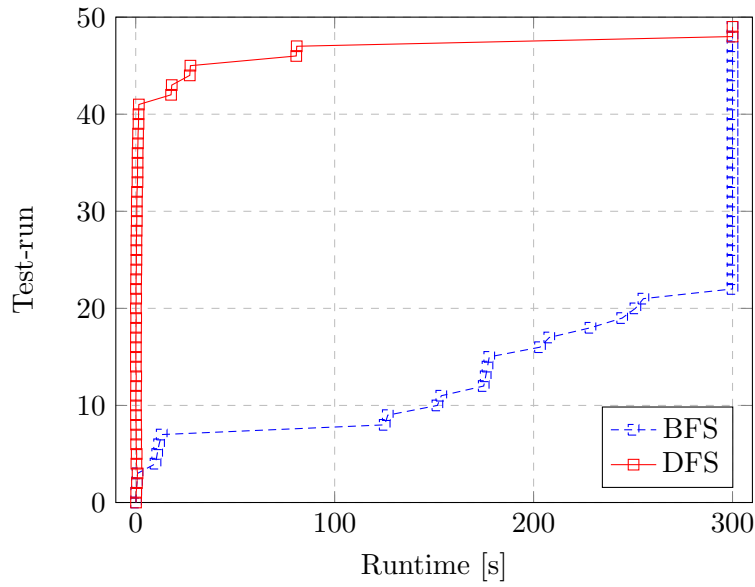


Figure 5.16: Runtime of random-based AFs over admissible semantics

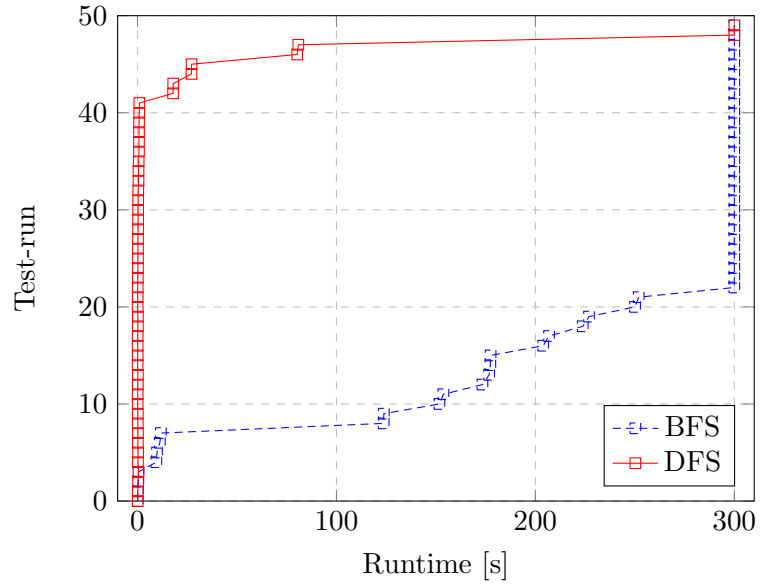


Figure 5.17: Runtime of random-based AFs over stable semantics

The second AFs generator procedure is the grid-based approach. Similar to the random-based approach, DFS dominates the BFS algorithm as shown in Figure 5.18, Figure 5.19 and Figure 5.20. We speculate, that the spikes at the conflict-free plot are edge-cases which produce a large amount of semantics extensions and the algorithm encountered the spurious extension only at a late stage of time.

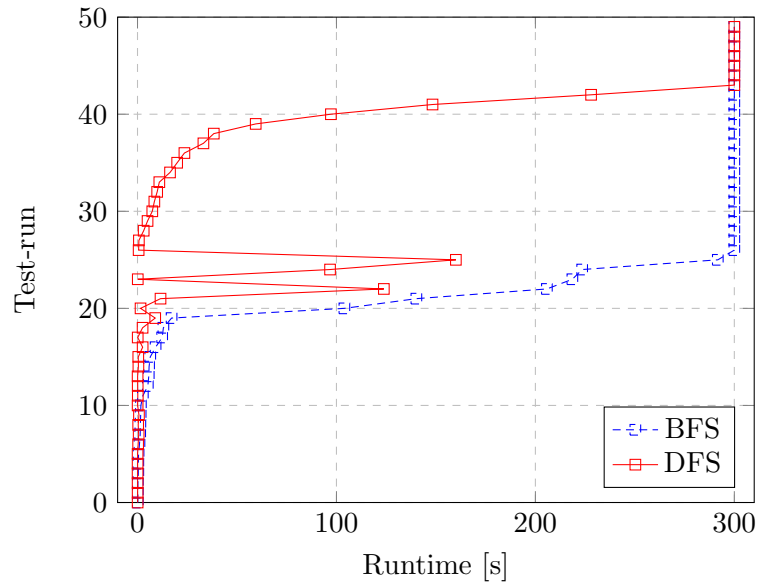


Figure 5.18: Runtime of grid-based AFs over conflict-free semantics

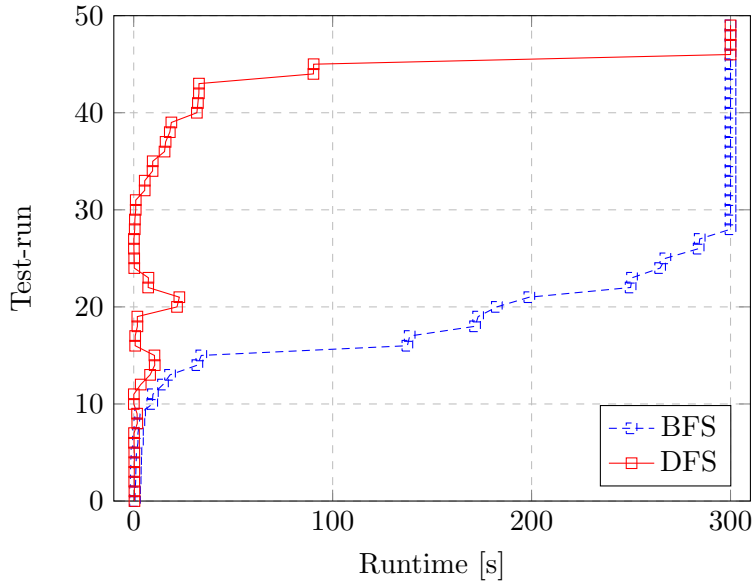


Figure 5.19: Runtime of grid-based AFs over admissible semantics

For stable semantics, the DFS approach managed to solve every test-run instance within the given time frame.

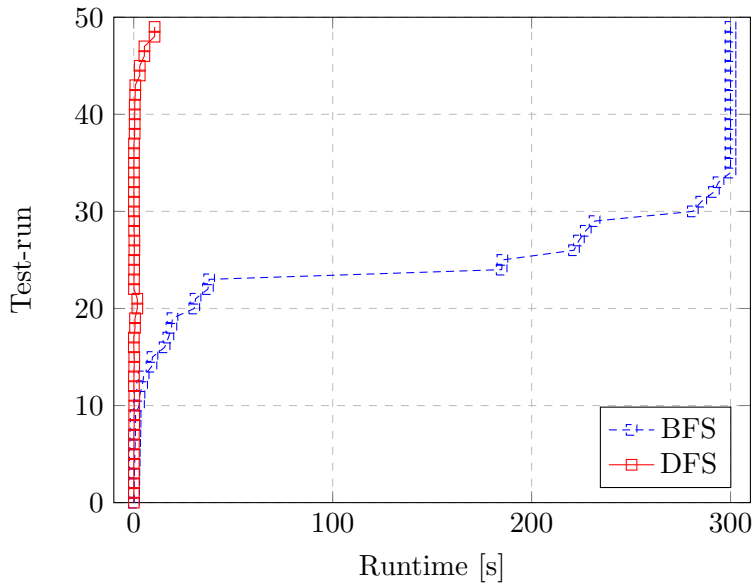


Figure 5.20: Runtime of grid-based AFs over stable semantics

Finally, the remaining AF generator approach is the level-based procedure. We produced three plots for every covered semantics, i.e., conflict-free in Figure 5.21, admissible in Figure 5.22 and stable in Figure 5.23. Also here DFS dominates the BFS algorithm

in all the test-runs. In contrast to the grid-based approach, the level-based approach created AF instances, which were harder to solve for BFS and DFS. Also for the stable semantics, two test-run instances ran into a timeout on the DFS algorithm. We speculate, that due to the AF generation procedure we generated AFs, which even after concretizing arguments remain spurious.

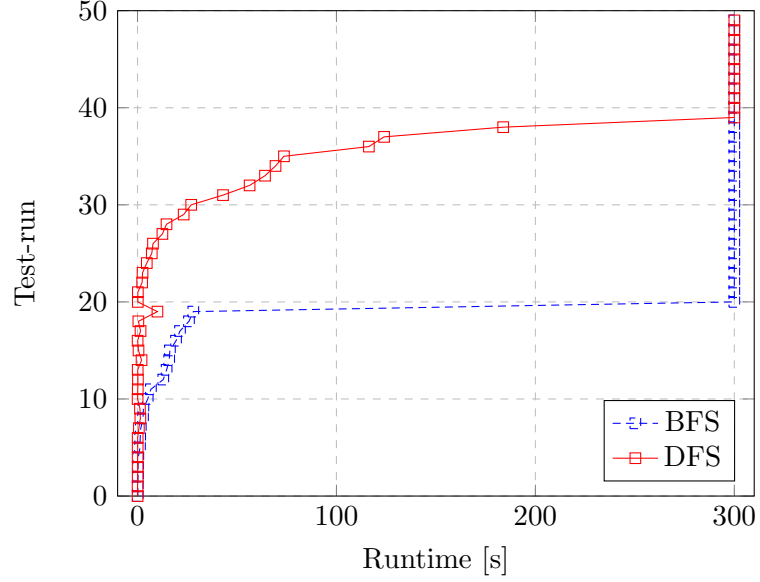


Figure 5.21: Runtime of level-based AFs over conflict-free semantics

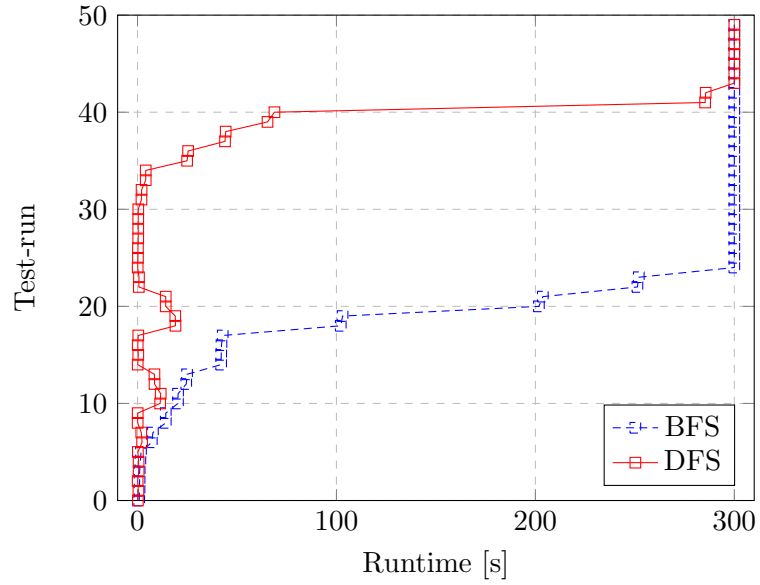


Figure 5.22: Runtime of level-based AFs over admissible semantics

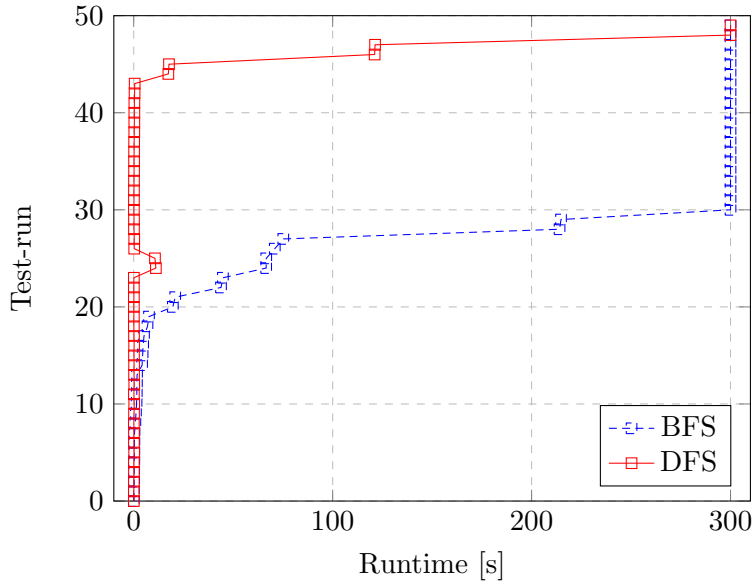


Figure 5.23: Runtime of level-based AFs over stable semantics

Finally, we grouped the data by the AF generator procedure and plugged it into a single plot. In Figure 5.24, we can see how dominant the DFS algorithm is compared to the BFS approach. Every test-run instance is either on the diagonal, i.e., DFS is just as fast as BFS, or below, i.e., DFS is faster than BFS.

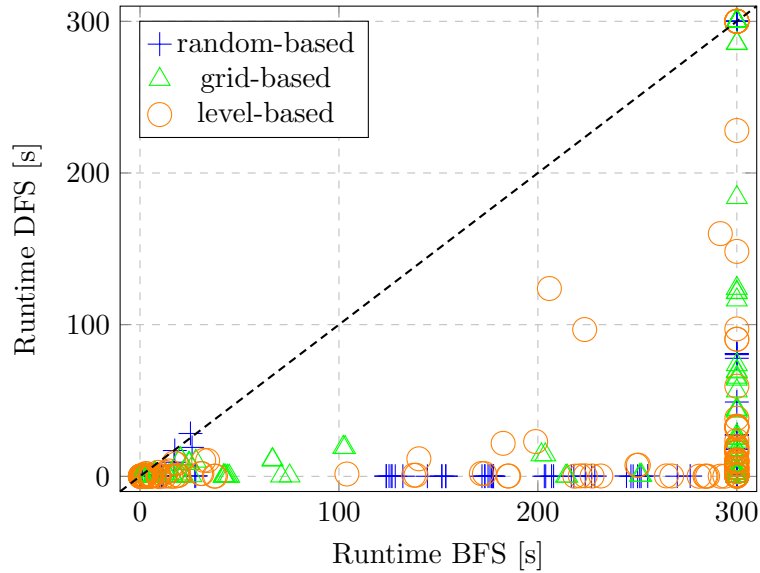


Figure 5.24: Runtime of BFS and DFS grouped by generator approach

5.4.2 Comparison of Refinement and No-Refinement

The results on the impact of the semantic specific refinements on the arguments concretizing program are shown in this section. As previously mentioned in the faithful/spurious check, the refinements made for admissible and stable did not have a significant impact. We hypothesize, that the SAT-Solver has discarded the refinements for optimization reasons for these two semantics. Nevertheless, the refinement made for conflict-free is not a refinement added to the Boolean formula, but is more an iterative improvement added at runtime by adding all the subsets of a computed conflict-free set.

The runtime of the test-run instances for the conflict-free semantics grouped by refinement and no-refinement are shown in Figure 5.25 for the AFs generated with the grid-based procedure and for the AFs generated with the level-based procedure in Figure 5.26. In almost all the test-runs the refinement decreases the runtime of the computation especially for the level-based AFs. Nevertheless, some instances are more efficient without the refinements. We hypothesize that these instances are test-runs that use the DFS algorithm and have a better result since, without the refinement, the algorithm generates the semantic extensions in a different order. The order is crucial when using the DFS algorithm because if the spurious extension is found in an early stage, the algorithm can abort.

For BFS, the refinement for conflict-free semantics improves the runtime for every instance. This is because the subset extraction of a computed semantics extension is always faster than computing the extension with an SAT-Solver. Since BFS computes all the semantics extensions before checking for spuriousness, the refinement decreases the runtime.

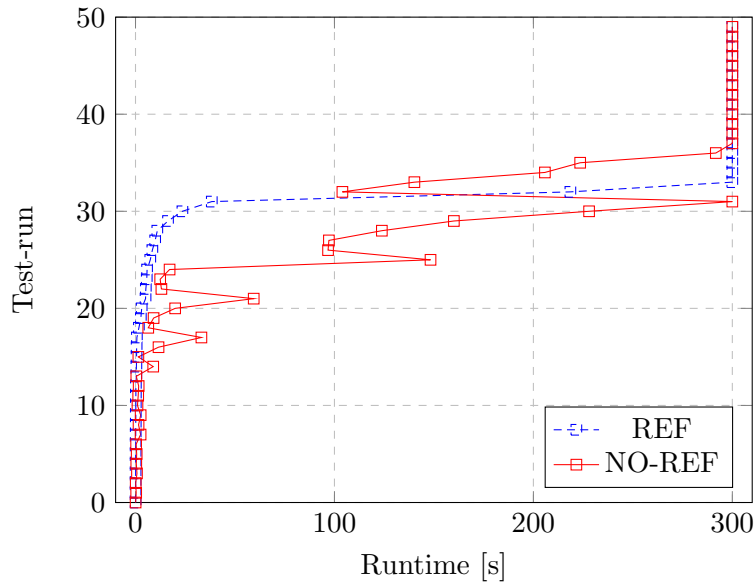


Figure 5.25: Runtime of grid-based AFs over conflict-free semantics

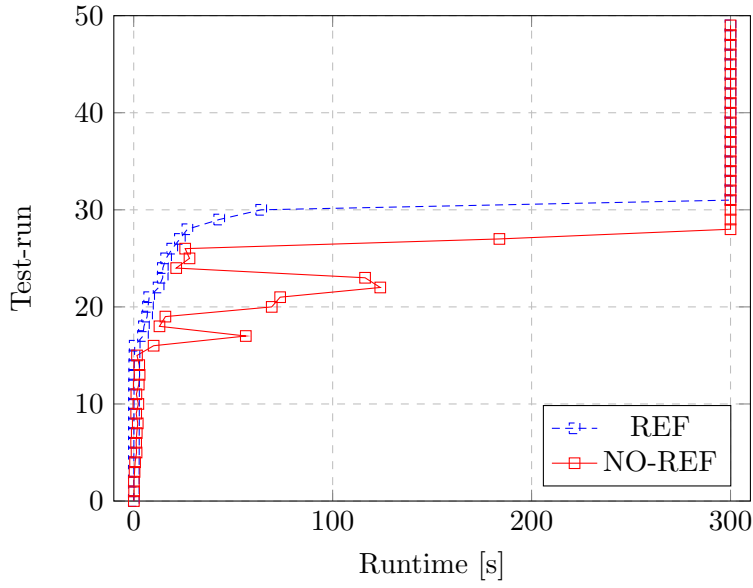


Figure 5.26: Runtime of level-based AFs over conflict-free semantics

5.4.3 Comparison of all Test-runs

In this section we compare the efficiency of the refinements to the corresponding semantics in Figure 5.27. As the plot shows, the refinements for admissible and stable had no significant impact. Nevertheless, for conflict-free the refinement had a big impact. Besides five outliers, the refinement decreased the runtime significantly.

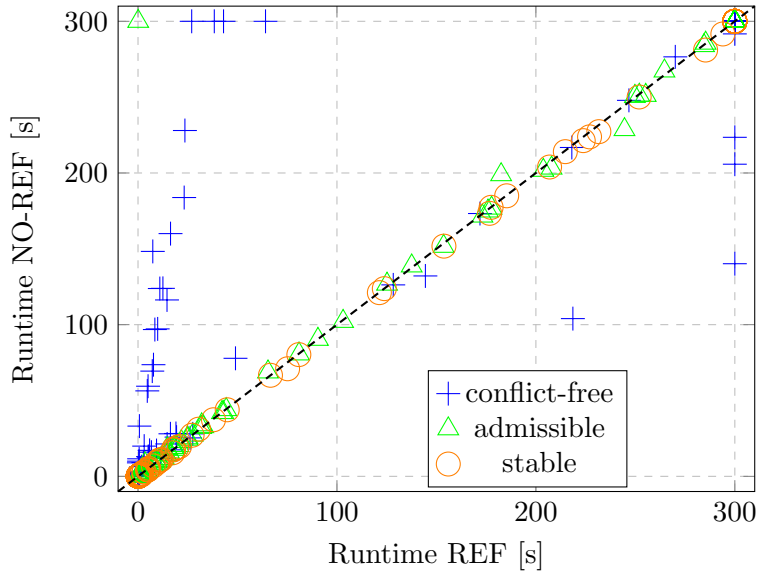


Figure 5.27: Performance comparison of Refinement and No-Refinement by Semantics

In Table 5.3 we grouped the test-run instances by the argument amount of the AFs. We then computed the mean runtime for all the implemented semantics, i.e., conflict-free, admissible and stable. Furthermore, we stated the test-run instances which did not terminate before the timeout of 300s. As the table shows, there is a direct correlation between the increase on arguments of an AF and the amount of timeouted test-runs.

Table 5.3: test-runs Statistics concretize arguments mean runtime

arguments amount	cf [s] (timeout)	adm [s] (timeout)	stb [s] (timeout)
10	18.13 (2/60)	33.07 (0/60)	33.20 (0/60)
15	18.00 (0/60)	59.24 (6/60)	53.36 (6/60)
20	64.17 (20/60)	21.92 (16/60)	1.61 (16/60)
25	31.40 (27/60)	40.55 (24/60)	31.60 (22/60)
30	31.89 (47/60)	59.79 (43/60)	19.33 (24/60)

6 Related Works

In recent years, publications targetting the topic of clustering arguments in AFs have been released. One of the first papers been written in this field was the paper "Existential Abstraction on Argumentation Frameworks via Clustering" from Saribatur and Wallner [30]. Since then, a tool was created (absarg-clustering)[29] to determine faithfulness and spuriousness of an AF and automatically finding non-spurious partitions. Furthermore, every two years the International Competitions on Computational Models of Argumentation (ICCMA) [23] runs a competition, aiming to assess the state of the art in practical systems for reasoning in central argumentation formalisms.

Absarg-clustering TODO

ICCMA Competition The International Competitions on Computational Models of Argumentation (ICCMA) runs a competition designed to foster research and development in implementing computational models of argumentation. The first competition was in 2015 and was associated with the workshop "Theory and Applications of Formal Argument (TAFA'15)". Back then, the covered semantics were *complete*, *preferred*, *grounded*, and *stable*. The main task was to compute semantics extensions and decide whether a given argument is credulously or skeptically inferred. Over the years, the competition evolved, and now there are multiple so-called "tracks", each with a different focus and different problem settings. The credulous/skeptical check is still one of the tracks called "Dynamic Track". After all the competitors have sent in their solutions, benchmark tests are executed for all the submissions, and a final ranking is published afterward.

Since 2015, the competition has been hosted every two years, and researchers world-wide can participate. The last competition hosted by the ICCMA in 2023 was won by the researcher team from the University Artois & CNRS with the name **Crustabri** [10] and second place was awarded to the solver from the University of Helsinki with the name μ -Toksia [25].

7 Conclusion

Future Work TODO

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] Philippe Besnard and Sylvie Doutre. “Checking the acceptability of a set of arguments.” In: Jan. 2004, pp. 59–64.
- [4] 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>.
- [5] 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>.
- [6] 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>.
- [7] Pasero Christian. *argumentation-framework-clustering*. <https://github.com/p4s3r0/argumentation-framework-clustering>. 2024.
- [8] Pasero Christian. *hashfunction-sat-solver-attacks*. <https://github.com/p4s3r0/hashfunction-sat-solver-attacks.git>. 2022.
- [9] 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.
- [10] crillab. *crustabri*. 2023.
- [11] 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>.

- [12] 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>.
- [13] 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.
- [14] 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](https://doi.org/10.1016/0004-3702(94)00041-X). URL: <https://www.sciencedirect.com/science/article/pii/000437029400041X>.
- [15] 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>.
- [16] 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.
- [17] 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/>.
- [18] 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.
- [19] Jia Hui Liang. “Machine Learning for SAT Solvers.” PhD thesis. University of Waterloo, Ontario, Canada, 2018. URL: <https://hdl.handle.net/10012/14207>.
- [20] 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>.
- [21] 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.
- [22] Hans van Maaren and John Franco. *The International SAT Competition Web Page*. <https://satcompetition.github.io/>. 2002.

- [23] Jean-Guy Mailly and Johannes P. Wallner. *International Competition on Computational Models of Argumentation (ICCMA) 2023 - Participating Solvers*. <https://argumentationcompetition.org/2023/solvers.html>. Accessed: 2024-10-09. 2023.
- [24] 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.
- [25] Andreas Niskanen and Matti Järvisalo. “ μ -toksia: An Efficient Abstract Argumentation Reasoner.” In: *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece, September 12-18, 2020*. Ed. by Diego Calvanese, Esra Erdem, and Michael Thielscher. 2020, pp. 800–804. DOI: 10.24963/kr.2020/82. URL: <https://doi.org/10.24963/kr.2020/82>.
- [26] 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>.
- [27] 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>.
- [28] 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>.
- [29] rbankosegger rbanko rbankosegger. *absarg-clustering*. 2023.
- [30] 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 Lake-meyer, and Esra Erdem. 2021, pp. 549–559. DOI: 10.24963/KR.2021/52. URL: <https://doi.org/10.24963/kr.2021/52>.
- [31] Toni and Francesca. “A tutorial on assumption-based argumentation.” In: *Argument and Computation* 5 (Feb. 2014). DOI: 10.1080/19462166.2013.869878.
- [32] Stephen E. Toulmin. *The Uses of Argument*. 2nd ed. Cambridge University Press, 2003.