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 23, 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 consider the stable argumentation semantics and the concepts of conflict-free sets and admissible sets. Moreover, we study 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 performs better than 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, both find applicability in terms of runtime performance for all of the examined semantics. While BFS is more efficient on small argumentation frameworks (with a size of up to 10 arguments), DFS outperforms BFS on more complex argumentation frameworks.

Kurzfassung

Argumentationssysteme sind ein grundlegender Bestandteil zur Formalisierung von argumentativen Szenarien im Forschungsgebiet der Künstlichen Intelligenz. Eine große Problemstellung stellt dabei die Skalierung der Systeme dar. Durch die Zunahme der Komplexität ist das Analysieren von Strukturen und Mustern in einem Argumentationssystem eine Herausforderung. Diese Herausforderung kann durch Clustering vereinfacht werden. Clustering ist eine Verfahren bei dem Argumente in sogenannten "Clustern" gruppiert werden, sodass keine falschen Schlussfolgerungen daraus impliziert werden können.

Da das Clustering von Argumentationssystemen ein relativ neues Konzept ist, wurde dahingehend noch nicht ausreichend geforscht. Diese Arbeit beschäftigt sich mit der Anwendung der Erfüllbarkeit von boolschen Formeln im Zusammenhang mit der Transformation von geclusterten Argumentationssystemen. Dabei wurden Algorithmen zur Feststellung von *faithfulness* (ein geclustertes Argumentationssystem ist faithful, wenn keine falschen Schlussfolgerungen daraus geschlossen werden können) entwickelt. Zusätzlich wurde ein weiterer Algorithmus entwickelt, der ein nicht-faithful Argumentationssystem so transformiert, dass es faithful wird. Dabei haben wir drei Semantiken berücksichtigt: conflict-free, admissible und stable. Zusätzlich haben wir verschiedene Algorithmen, wie die Breitensuche und die Tiefensuche und noch weitere Verfeinerungen/Optimierungen je nach Semantik, welche die Effizienz der Algorithmen steigern, implementiert.

In dieser Arbeit, zeigen wir eine empirische Auswertung unserer Implementation der Algorithmen. Die Auswertung zeigt, dass je nach Größe und Struktur des Argumentationssystems ein Algorithmus effizienter ist als der andere. Die Optimierungen haben sehr gute Verbesserungen vor allem bei der conflict-free Semantik gezeigt, hatten allerdings bei admissible und stable keinen signifikanten Unterschied. Die Breitensuche und die Tiefensuche haben beide Vor- und Nachteile. Die Breitensuche ist effizienter als die Tiefensuche bei kleinen Argumentationssystemen (bis zu einer Größe von 10 Argumenten) und die Tiefensuche ist effizienter bei komplexeren Systemen.

Acknowledgements

I would like to express my deepest gratitude to everyone who supported me throughout the process of writing this thesis.

First and foremost, I would like to thank my supervisor, Johannes P. Wallner, for his invaluable guidance and for all the time he dedicated to supporting me throughout the process of completing this thesis. Thank you for all our insightful discussions in your office and for making it so easy for me to reach out and ask for your valuable feedback. Furthermore, I would like to thank Maria Eichlseder for the mentorship of my bachelor thesis and for all the valuable feedback you provided.

Thanks to all my friends I met at university, especially Simon, Martin, Lukas, and Timo, who were always there for productive discussions (sometimes even until late at night for some last-minute pre-deadline code changes).

I extend my appreciation to my friends in Graz besides university. Especially to my best-friend/roommate/uncarriable-teammate Felix, who was always there to distract me from what was important. I would also like to thank my friends Susanne and Katja for the time we spent together in Graz, where I was able to disconnect from my computer and enjoy time besides university.

Also, thank you to all my friends at home, especially Michael, for all the mathematical support and the countless chess games played together; Eva, for all the moral and medical support; and Mirjam, for always being there when I'm in need.

Finally, I would like to express my deepest gratitude to my beloved family. To my grandmothers, Nonna Gemma - *Grazie per avermi sempre sostenuto e per tutte le candeline che mi hai acceso per ogni esame*; Oma Gertrud - *Danke, dass du immer da warst und für all die leckeren Mahlzeiten.*

I would also like to thank my sister Sabrina for always being by my side and being the kindest person I know. Thank you for all the time we spent together; Having a lifelong sister and friend by my side has made my life so much easier.

Last but not least, I would like to thank my parents, my father - *Grazie per avermi sostenuto e per darmi sempre la certezza di poter contare sul tuo supporto*; and my mother - *Danke, dass du immer an mich geglaubt hast und mich an jeden meiner Schritte ermutigt und unterstützt hast.*

Without these great people, I wouldn't be the person who I am today. Thank You.

Contents

List of Figures

List of Tables

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. A cornerstone of Artificial Intelligence (AI), conversely, is that of rationality, based on precise and logical foundations [25]. To do so, the data needs to be stored and structured in a way that AI can extract information from it. This is part of knowledge representation and reasoning and that is why much research is being done in that field [19].

Arguments can have many forms [29]. 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, a so-called argumentation framework (AF) [11]. In many cases, viewing arguments as abstract entities is sufficient to carry out argumentative reasoning [4, 15]. Such reasoning is defined via so-called argumentation semantics [3], 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 [11].

Semantics define subsets of arguments that have a certain relation to each other. Dung defined different semantics [11] 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 and 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 [27]. 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 the following 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 $stb(G) = \big\{\{d, e\}, \{b, d, f\}\big\}$.

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

Figure 1.1: Concrete and abstract AF

By computing the stable extensions of the abstract AF $\hat{G}$ $stb(G) = \big\{\{d\}, \{d, \hat{g}\}\big\}$, we can observe that the AF is spurious due to the extension $\{d\}$, since the abstract stable extension cannot be mapped to one of the concrete stable extensions.

When concretizing (i.e. removing an argument from the cluster) the argument c, we create a new AF $\hat{G}' = (\hat{A}', \hat{R}')$ depicted in 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 considering the parts $\{e\}$ or $\{b, f\}$ of cluster $\hat{g}$. Thus, we created a faithful abstract AF.

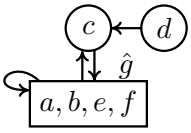


Figure 1.2: Concretized AF $\hat{G}'$

When producing an AF with multiple layers 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 [12]. In fact, the complexity of proving faithfulness or spuriousness of an AF is $\prod_2^P$-hard [27]. In practice this means, in order to obtain a result, multiple instances or calls of efficient procedures capable of solving NP-hard problems, such as SAT-Solvers [6], need to be invoked.

We created one of the first tools to produce an abstract AF based on a concrete AF. 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 abstract 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 keep concretizing arguments of the abstract AF until faithfulness is reached (if possible according to our specifications), since the extraction of the user-defined arguments may result in spurious reasoning. Furthermore, we extended the algorithm by making the concretizer list optional. In this case, the user has no control over which arguments are concretized and we cannot guarantee that the program finds a solution for every AF.
- We implemented our algorithms in Python3 with the SAT-Solver z3 [21] and provide the implementations in open-source.
- In an experimental evaluation, we tested the efficiency of the tool on the various algorithms and provide insights on the impact of different approaches (BFS, DFS, refinements and no-refinements). The experimental evaluation showed that our tool solves most of the instances with at most 30 arguments. For AFs beyond this size, the performance is dependent on the structure of the AF and the chosen semantics. Furthermore, DFS is more efficient than BFS and the refinements have a significant impact on the conflict-free semantics. For admissible and stable, the refinements do not decrease the runtime significantly. Due the usage of heuristics, the algorithm cannot guarantee it will find a solution, where the goal is to create a new faithful abstract AF based on a spurious AF.

Our open-source implementation can be found at:

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

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 [11]. 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, 28]. But do also find their applications in other fields like natural language processing [8], legal and medical reasoning [1], and even in game theory and strategic reasoning [23].

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 $R = \big\{(a, b), (b, b), (a, c), (c, a), (c, d), (d, e), (e, d)\big\}$.*

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

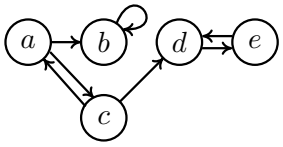


Figure 2.1: AF G

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 [11] 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 denote the set of conflict-free sets of an AF G by $cf(G)$.

Definition 1 ([11]). *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. The admissible sets of an AF G are represented as $adm(G)$.

Definition 2 ([11]). *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. We sometimes call a stable extensions also a stable set. The stable sets (or extensions) of an AF G are denoted as $stb(G)$.

Definition 3 ([11]). *Let $G = (A, R)$ be an AF. Then a set $S \subseteq A$ is stable in G if and only if $S \in cf(G)$, and $b \notin S$ 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 [5]. Which then can be used to encode the AFs to be solvable with different solvers like a solver for Answer Set Programming (ASP) [7, 13] or, as in our case, with a SAT-Solver [2]. 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 [12].

2.2 Clustering of Argumentation Frameworks

When talking about AFs in general, we already have an abstraction layer. By clustering, we add another layer of abstraction where we combine different arguments into one or multiple so called *clusters*. These clustered AFs (i.e. abstract AFs) have extensions, just as non-clustered AFs (i.e. concrete AFs), which we call abstract 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* abstract AFs. The term *faithful* describes the property of an abstract AF, that every abstract extension can be mapped to a concrete extension. If the abstract AF creates a set X under an abstract semantics which cannot be mapped to a set under the corresponding concrete semantics, we call the set X *spurious*.

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 the abstract AF $\hat{G} = (\hat{A}, \hat{R})$ with the arguments $\hat{A} = \{d, e, \hat{h}\}$, where the cluster $\hat{h}$ contains the arguments $\{a, b, c, d\}$ and the attacks being $\hat{R} = \big\{(\hat{h}, d),\ (d, e),\ (e, d),\ (\hat{h}, \hat{h})\big\}$. This AF can be represented as a directed graph as shown in Figure 2.2.

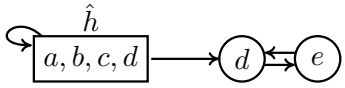


Figure 2.2: Abstract AF $\hat{G}$

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 abstract 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 an abstract AF $\hat{G} = \{\hat{A}, \hat{R}\}$ and redefine the semantics. To extract the singletons from a set S (composed of singletons and clusters), we make use of the $singleton(S)$ function.

Conflict-Free A set of arguments is abstractly conflict-free, if there is no attack between the singletons of the set.

Definition 4 ([27]). *Let $\hat{G} = (\hat{A}, \hat{R})$ be an AF. Then a set $S \subseteq \hat{A}$ is conflict-free in $\hat{G}$ if and only if for each $\hat{a}, \hat{b} \in singleton(S)$ we have $(\hat{a}, \hat{b}) \notin \hat{R}$.*

Example 5. *The abstractly conflict-free sets of the abstract AF G in Figure 2.2 are $\emptyset$, $\{d\}$, $\{e\}$, $\{\hat{h}\}$, $\{e, \hat{h}\}$ and $\{d, \hat{h}\}$.*

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 ([27]). *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 $\hat{a} \in singleton(\hat{G})$, then there is a $\hat{c} \in \hat{A}$ with $(\hat{c}, \hat{b}) \in \hat{R}$.*

Example 6. *The abstractly admissible sets of the abstract AF G in Figure 2.2 are $\emptyset$, $\{e\}$, $\{\hat{h}\}$, $\{e, \hat{h}\}$, $\{d, \hat{h}\}$.*

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 ([27]). *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 $\hat{b} \in singleton(\hat{A})$.*

Example 7. *The abstractly stable extensions of the abstract AF G in Figure 2.2 are $\{e, \hat{h}\}$ and $\{d, \hat{h}\}$.*

Let us have a look at a concrete example to explain faithfulness. The concrete AF $G = (A, R)$ has the following arguments $A = \{a, b, c, d, e\}$ with these attacks: $R = \big\{(a, b), (b, b), (a, c), (c, a), (c, d), (d, e), (e, d)\big\}$. This AF can be represented as a directed graph shown in 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 corresponding attacks are $\hat{R} = \big\{(\hat{h}, e), (e, \hat{h}), (\hat{h}, \hat{h})\big\}$. The attacks are directly derived from the concrete AF. If an argument is clustered, the cluster inherits the attacks from the argument. The emerging AF can be represented as a directed graph shown in Figure 2.3(b).

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

Figure 2.3: Concrete and abstract AF

If we compare the stable extensions of the concrete AF G (i.e. $stb(G) = \big\{\{a, e\}, \{a, d\}\big\}$) with the abstractly stable extensions of the abstract AF $\hat{G}$ (i.e. $\hat{stb}(G) = \big\{\ \{\hat{h}\}, \{e\}, \{e, \hat{h}\}\big\}$), we see that it is spurious due to the abstractly stable extension $\{e\}$ which cannot be mapped to one of the concrete stable extensions. The mapping of semantics extensions with abstract AFs is done the same way as for concrete AFs, except that clusters can mutate to every possible combination of the abstract 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 abstract 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 $\hat{R}' = \{(d, \hat{h}), (d, e), (e, d), (\hat{h}, \hat{h})\}$.

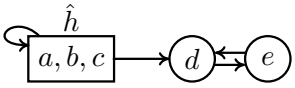


Figure 2.4: Concretized AF $\hat{G}'$

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 abstract AF $\hat{G}'$ is faithful.

2.3 Boolean Satisfiability

A SAT(isfability)-Solver is used to compute Boolean formulas in an efficient way [6]. The main purpose is to determine if a formula is satisfiable (i.e. 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, 24]. But also in AI and machine learning [20] and even in security [30].

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 [10]. 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 [14]. 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 [21].

3 Algorithmic Approaches 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. Next, we described in Section 3.2 how we decide if an AF is spurious or faithful. In Section 3.3 we explain the concretization of a clustered argument. Next, in Section 3.4 we explain how the concretizer list (a list of clustered arguments which are transformed to singletons) is computed. The approach to compute faithful abstract AFs is then described in Section 3.5. Then we state the heuristics and refinements in Section 3.6 and finally we discuss a specific relation between spurious sets in Section 3.7.

3.1 Boolean Encodings for Abstract 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 abstract AFs. We develop these encodings for (abstract) conflict-free, (abstract) admissible, and (abstract) stable extensions. For each semantics we present the corresponding Boolean encoding and an example.

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

We further define a subset of arguments, denoted as $\hat{A}_{Single} \subseteq \hat{A}$ consisting of the singletons arguments of 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. Define the following formula.*

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

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* $\hat{R} = \big\{(a,a),\ (a,b),\ (b,c),\ (d,a),\ (d,b),\ (d,c)\big\}$.

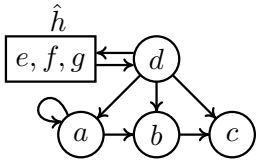


Figure 3.1: Abstract AF $\hat{G}$

With Definition 7 we obtain the following Boolean formula.

$$\varphi_{cf}(\hat{G}) = \neg(b \wedge a) \wedge \neg(b \wedge d) \wedge \neg(a \wedge d) \wedge \neg(a \wedge a) \wedge \neg(c \wedge b) \wedge \neg(c \wedge d)$$

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

If we compare the models of $\varphi_{cf}(\hat{G})$ 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 [5].

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. Define the following formula.*

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

Example 9. *Let us have a look at an example and define the AF* $\hat{G} = (\hat{A}, \hat{R})$ *to be the AF depicted in Figure 3.2. Where the arguments are* $\hat{A} = \{a, b, c, d, e\}$ *and the attacks* $\hat{R} = \big\{(a,a), (a,b), (b,c), (d,a), (d,b), (d,d), (d,e), (e,d)\big\}$.

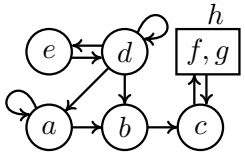


Figure 3.2: Abstract AF $\hat{G}$

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

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

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

If we compare the models of $\varphi_{adm}(\hat{G})$ 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 a previously defined formula for admissible sets [5].

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 attacked by 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. Define the following formula.*

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

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

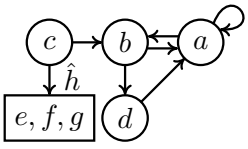


Figure 3.3: Abstract AF $\hat{G}$

If we apply the encoded formula defined in Definition 9 to the AF $\hat{G}$, we obtain the

following Boolean expression.

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

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

By comparing the models of $\varphi_{stb}(\hat{G})$ with the stable sets of $\hat{G}$, we find that they are identical. Specifically, Definition 9 is a Boolean expression to derive stable sets from an AF. If the AF has no clusters, the Boolean expression reduces to the stable formula defined earlier.

3.2 Checking Faithfulness

In the context of abstract AFs, faithfulness is an important property. To be able to determine if an AF is faithful or spurious we develop two different approaches, i.e., BFS and DFS wich are described in Section 3.2.1. To be able to show faithfulness or spuriousness for a specific abstract extension, we developed again two different procedures, i.e., list comparison and the SAT-based check described in Section 3.2.2.

3.2.1 BFS and DFS

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 on the computation of semantics.

BFS The BFS approach in our implementation, depicted in Figure 3.4, first computes all the semantics extensions of the abstract AF and then checks if any of the extensions is spurious. After one spurious extension is encountered, the algorithm is aborted, because if one extension is spurious the AF is spurious as well.

The BFS approach 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.

DFS On the other hand we have the DFS approach, depicted in Figure 3.5. 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. If an extension cannot be mapped, we found a spurious extension which shows spuriousness of the AF.

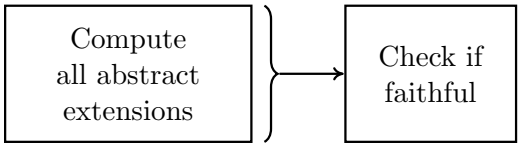


Figure 3.4: BFS visualization

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.

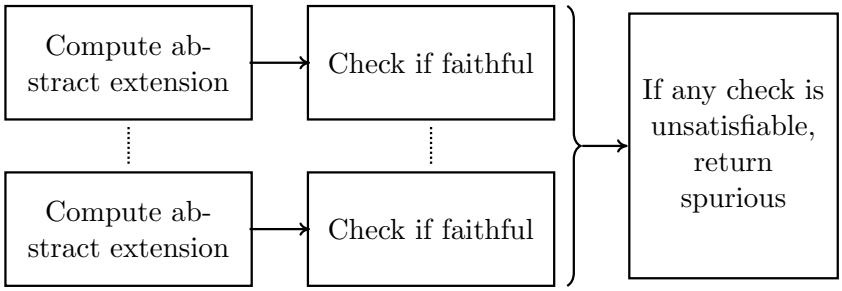


Figure 3.5: DFS visualization

3.2.2 Extension Mapping

To determine if an abstract extension is faithful or spurious, we need to map it to a concrete extension. If this is not possible, we found a spurious extension. We created two procedures for the mapping, i.e., list comparison and SAT-Based check.

List Comparison The first approach of mapping an abstract extension to a concrete extension is the list comparison. Here, all the concrete extensions are computed and are then checked iteratively for the mapping. Since list comparisons are highly optimized and do not need an additional SAT-Solver invocation, the approach can be efficient under certain circumstances.

The issue with the list comparison of the computed extension is, that for AFs with big clusters (i.e. a cluster containing many arguments), the direct mapping of an abstract extension is tedious. This is because of the many possibilities a cluster can transform into, e.g., for a cluster $\hat{h}$ containing the arguments a and b, the extension $\{\hat{h}\}$ could transformed into $\{a\}$, $\{b\}$ and $\{a, b\}$. This works fine for small clusters, but can be become more computationally intensive for larger clusters.

SAT-Based check For optimization reasons, we altered the way on how to check for faithfulness. That is, why we implemented a SAT-based check. Instead of computing

the concrete extensions, we check if the formula of the semantics from the concrete AF is still satisfiable, if we add the abstract extension. The abstract extension is encoded with the corresponding Boolean variables, where every argument in the extension is set to *true* and the arguments not in the extension is set to *false*. The clusters are enrolled by the clustered arguments disjunctively, to cover every possible cluster mapping.

3.3 Concretizing Singletons

When operating on abstract 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 information, 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 compensate 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 3.6, where we concretize the arguments a and b.

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

Figure 3.6: 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 3.7 and the pseudo-code in Algorithm 1. We defined the arguments to be concretized as a list of arguments $list(S)$, with S being the computed concretizer list. Furthermore, we have an AF datastructure $F : AF(A, R)$. Here A is a dictionary of arguments, where each argument has the property *attacks* (i.e. a list of arguments that are being attacked by the argument) and *attacked_by* (i.e. a list of

arguments that attack the argument). The attacks of the AF are stored in the list R, where the entities of an attack $r = (a, b)$ can be extracted with $r.source$, mapping to the attacker a of the attack r and $r.target$, mapping to the argument that is being attacked b in the attack r.

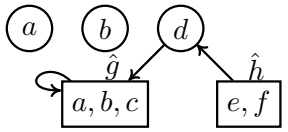


Figure 3.7: Concretized AF $\hat{F}'$ after Step 1

Algorithm 1 Concretizing Singletons Pseudocode Step 1

Require: $\hat{F} : AF(\hat{A}, \hat{R})$ ▷ Abstract Clustered AF
Require: $e : list(S)$ ▷ Concretizer List
1: $\hat{F}' \leftarrow \hat{F}$ ▷ N = Concretized Cluster
2: **for** a_i in e **do**
3: **for** c_i in $\hat{F}.clusters$ **do**
4: **if** a_i in c_i **then**
5: $c_i.remove(a_i)$
6: **end if**
7: **end for**
8: $\hat{F}'.addSingleton(a_i)$
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 3.8 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.

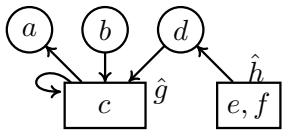


Figure 3.8: Concretized AF $\hat{F}'$ after Step 2

Algorithm 2 Concretizing Singletons Pseudocode Step 2

Require: $F : AF(A, R)$ ▷ Concrete AF
Require: $e : list(S)$ ▷ Concretizer List
Require: $\hat{F}' : AF(\hat{A}', \hat{R}')$ ▷ Concretized Cluster

```
for e_i in e do
    for att_i in F[e_i].attacks do                ▷ Add attacks (argument, cluster)
        if att_i is cluster then
            F̂'.addAttack((e_i, att_i))
        end if
    end for
    for att_by_i in F[e_i].attacked_by do         ▷ Add attacks (cluster, argument)
        if att_by_i is cluster then
            F̂'.addAttack((att_by_i, e_i))
        end if
    end for
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 3.9 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.

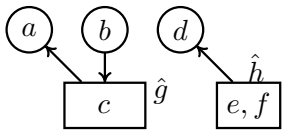


Figure 3.9: Concretized AF $\hat{F}'$ after Step 3

Algorithm 3 Concretizing Singletons Pseudocode Step 3

Require: $F : AF(A, R)$ ▷ Concrete AF
Require: $\hat{F} : AF(\hat{A}, \hat{R})$ ▷ Abstract AF
Require: $\hat{F}' : AF(\hat{A}', \hat{R}')$ ▷ Concretized Cluster

```
for r_i in R̂' do
    if r_i.target is cluster and r_i.source is cluster then                ▷ Part 1
        for a_i in F̂[r_i.source] do
            if not(a_i attacks any of A[r_i.target]) then
                F̂'.removeAttack(r_i)
                break
            end if
        end for
    end if
    if r_i.target is cluster then                                          ▷ Part 2
        if not(r_i.source attacks any of A[r_i.target]) then
            F̂'.removeAttack(r_i)
            continue
        end if
    end if
    if r_i.source is cluster then                                          ▷ Part 3
        if not (r_i.target is attacked by any A[r_i.source]) then
            F̂'.removeAttack(r_i)
            continue
        end if
    end if
end for
```

Step 4: In this step we add the new attacks between the singletons. Due to the fact that we copied all the attacks from $\hat{F}$, we only have to take into consideration the attacks 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 3.10 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.

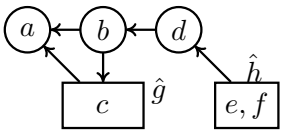


Figure 3.10: Concretized AF $\hat{F}'$ after Step 4

Algorithm 4 Concretizing Singletons Pseudocode Step 4

Require: $F : AF(A, R)$ ▷ Concrete AF
Require: $\hat{F} : AF(\hat{A}, \hat{R})$ ▷ Abstract AF
Require: $\hat{F}' : AF(\hat{A}', \hat{R}')$ ▷ Concretized Cluster
Require: $e : list(S)$ ▷ Concretizer List

```
for e_i in e do
    for a_i in A[e_i.attacks] do
        if a_i is singleton and (e_i, a_i) not in R then
            F̂'.addAttack((e_i, a_i))
        end if
    end for
    for a_i in A[e_i.attacked_by] do
        if a_i is singleton and (a_i, e_i) not in R then
            F̂'.addAttack((a_i, e_i))
        end if
    end for
end for
```

Step 5: The last step is to clean up the argumentation framework $\hat{F}'$ by removing all empty clusters and mutating the clusters with exactly one singleton to the mentioned singleton. The resulting AF $\hat{F}'$ is depicted in Figure 3.11 and the pseudo-code in Algorithm 5.

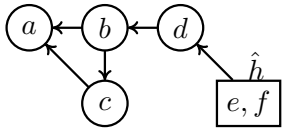


Figure 3.11: Concretized AF $\hat{F}'$ after Step 5

Algorithm 5 Concretizing Singletons Pseudocode Step 5

Require: $F : AF(A, R)$ ▷ Concrete AF
Require: $\hat{F} : AF(\hat{A}, \hat{R})$ ▷ Abstract AF
Require: $\hat{F}' : AF(\hat{A}', \hat{R}')$ ▷ Concretized Cluster
Require: $e : list(S)$ ▷ Concretizer List

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

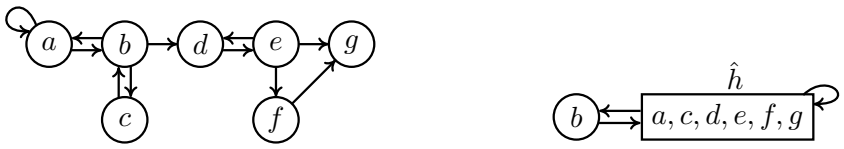
3.4 Computation of Concretizer List

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 abstract 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 3.12(a) and a corresponding abstract AF $\hat{G}$ in Figure 3.12(b).



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

Figure 3.12: Concrete and abstract AFs

If we have a look at the stable extensions of the concrete AF G, i.e. $stb(G) = \big\{\{b, c\}\big\}$ and at the abstractly stable extensions of the abstract AF $\hat{G}$, i.e. $\hat{stb}(G) = \big\{\{b, \hat{h}\}, \{\hat{h}\}, \{b\}\big\}$, 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 1 represents the direct attacker of an 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. 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.

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 3.13 depicted in dotted line style and the attacker, depicted in dashed line style. The pseudo-code of this procedure is listed in Algorithm 6 where N represents the list of neighbours at depth 2 that we are computing. Furthermore, the function $neighbours(arg)$ returns all the direct neighbours of the argument arg.

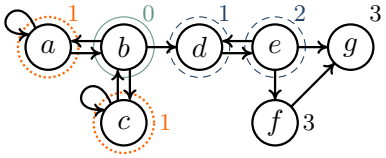


Figure 3.13: Depth of singletons from the perspective of b

Algorithm 6 Computation of Concretizer list Algorithm: Neighbours

Require: $G : AF(A, R)$ ▷ Concrete AF
Require: $s : list(Arguments)$ ▷ Working List
$N \leftarrow \emptyset$ ▷ N = list of neighbours
$n(1) \leftarrow \emptyset$ ▷ List of neighbours with depth 1
for s_i in s **do** ▷ Get neighbours
    for n in $neighbours(s_i)$ **do** ▷ depth 1 attacker
        $n(1).append(n)$
        $N.append(n)$
    end for
    for $n(1)_i$ in $n(1)$ **do**
        for n in $neighbours(n(1)_i)$ **do** ▷ depth 2 attacker
            $N.append(n)$
        end for
    end for
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,

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

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

we add them to each combination set. After adding them, we filter for duplicates to keep the concretizer list size to a minimum.

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

Finally, we sort the list by the set size and return it. In the current example we would return the whole table, because no concretizer arguments were provided by the user. So the concretizer list would be $\big\{\{a\}\ \{c\}\ \{d\}\ \{a, c\}\ \{a, d\}\ \{c, d\}\ \{d, e\}\ \{a, c, d\}\ \{a, d, e\}\ \{c, d, e\}\ \{a, c, d, e\}\big\}$. 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
$C \leftarrow$ combinations of N with $range(1, len(N) - 1)$ ▷ Combination List
for ca_i in ca **do** ▷ Parameter Arguments to be concretized
    for c_i in C **do**
        $c_i.append(ca_i)$
    end for
end for
$C.deduplicate()$
for s_i in C **do** ▷ Remove clusters
    for a_i in s_i **do**
        if $\hat{G}[a_i]$ is cluster **then**
            $s_i.remove(a_i)$
        end if
    end for
end for
return $s.sortBySize()$

3.5 Finding 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 abstract 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 3.4. 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* $R = \big\{(a, a), (a, c), (b, a), (c, b), (c, d), (d, c), (e, b), (b, e), (f, f)\big\}$*, depicted in Figure 3.14(a). When clustering the arguments* $\{a, b, f\}$ *we obtain the abstract AF* $\hat{G} = (\hat{A}, \hat{R})$ *depicted in Figure 3.14(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* $\{a, b, f\}$*.*

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

Figure 3.14: Concrete and abstract AF

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

Algorithm 8 Compute Faithful Clusters

Require: $G : AF(A, R)$ ▷ Concrete AF
Require: $\hat{G} : AF(\hat{A}, \hat{R})$ ▷ Abstract AF
1: $sp \leftarrow computeSpuriousExtensions(G, \hat{G})$
2: $conc \leftarrow computeConcretizerList(sp)$
3: **for** set_i in sp **do**
4: $\quad \hat{G}' \leftarrow concretize(\hat{G}, set_i)$
5: $\quad$ **if** $checkFaithfulness(G, \hat{G}')$ **then**
6: $\quad\quad$ **return** $\hat{G}'$
7: $\quad$ **end if**
8: **end for**

3.6 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 no 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.

$$\eta_{adm}(F) = \overline{cf}(F) \vee \overline{def}(F)$$

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

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

Stable For the stable semantics we apply the same principle as for abstract admissibility. Since we cannot conclude anything from the subsets of the computed (abstract) semantics, we need to add the negation of the stable formula for the concrete AF. With this, we aim to reduce the semantics extensions which need to be computed for showing spuriousness or faithfulness, 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.

$$\eta_{stb}(F) = \overline{cf}(F) \vee \overline{att}(F)$$

The two components (i.e. $\overline{cf}$, $\overline{att}$) represent two different conditions which together define the refinement. Here, $\overline{cf}$ is the negated part of conflict-freeness of the concrete AF and $\overline{att}$ describes that we want to ignore the concrete stable sets which impliy that if and argument being outside the stable set, has an attacker inside the stable set.

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

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

3.7 Relations between Spurious Sets

When developing the tool, we came up with different theories. Some theories were discarded immediately, some were implemented (recall Section 3.6) 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 $R = \big\{(b, a),\ (d, a),\ (a, d),\ (a, c),\ (c, e),\ (c, f),\ (d, f)\big\}$ depicted in Figure 3.15(a). The spurious abstract AF $\hat{G} = (\hat{A}, \hat{R})$ is defined by the arguments $\hat{A} = \{\hat{g}, e\}$, with the cluster $\hat{g}$ containing the arguments $\{a, b, c, d, f\}$ and the attacks being $\hat{R} = \big\{(\hat{g}, \hat{g}), (\hat{g}, e)\big\}$ depicted in Figure 3.15(b).*

(a) Concrete AF (b) Spurious abstract AF

Figure 3.15: Concrete and spurious abstract AF

When computing the stable extensions of G, we obtain $\big\{b, c, d\big\}$, for the abstract AF $\hat{G}$ we get the abstractly stable extensions $\big\{\{\hat{g}\}, \{\hat{g}, e\}\big\}$. Since the abstractly stable extensions $\{\hat{g}, e\}$ is 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 trying to map a subset of $\{\hat{g}, e\}$, e.g. $\{\hat{g}\}$ to the concrete extension $\{b, c, d\}$, we can see that it is faithful. Thus, we found a subset of a spurious extension, which is not spurious.

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. In Section 4.2 we describe the usage of the implemented tool and the settings to change the inner workings of the algorithms.

4.1 Generating 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 a probability of `<p>` to be an actual attack of the generated AF.

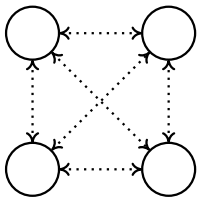
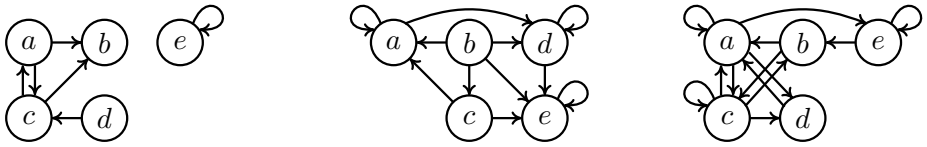


Figure 4.1: Random-based approach with $Amount = 4$ and $p = 1$

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



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

Figure 4.2: Example AF generated with random-based approach

Grid-based Next we are going to discuss the grid-based approach. The arguments for the script are `<arg_amount>`, being the amount of arguments the AF has and `<p>`, which is the probability that an attack between two arguments occurs. Different to the random-based approach, attacks can only happen between the direct neighbours of the grid (i.e. top, bottom, right, left). The grid is an $n \times n$ grid, with n being equal to $\lfloor(\sqrt{\texttt{<arg\_amount>}})\rfloor$. An example grid can be seen in Figure 4.3.

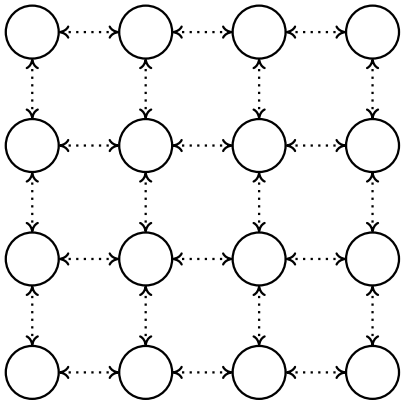
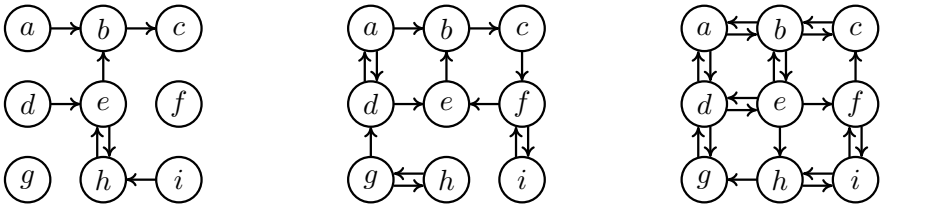


Figure 4.3: Grid-based approach with $Amount = 16$ and $p = 1$

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 `<arg_amount>`, `<level>` and `<p>`. Same as for the grid-based and random-based approach, `<arg_amount>` defines how many arguments the computed AF has. The `<level>` argument restricts the height of the grid to the provided value and `<p>` is again the probability that an attack between to arguments occurs. The difference to the grid-based approach is the dimension of the



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

Figure 4.4: Example AF generated with grid-based approach

grid. While the grid-based approach uses an $n \times n$ grid, in the level-based approach we use a `<level>` $\times n$ grid. In this context, n is equal to $\lceil$`<arg_amount>`/`<level>`$\rceil$. An example grid is depicted in Figure 4.5.

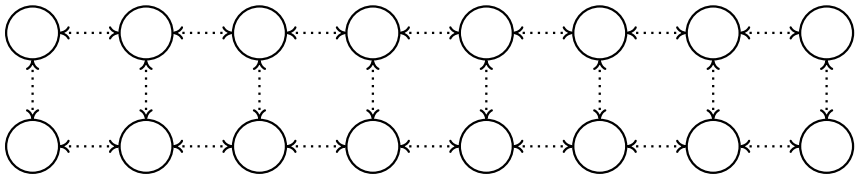


Figure 4.5: Level-based approach with $Level = 2$, $Amount = 16$ and $p = 1$

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 $min($`<level>` $+ 1, 4)$ amount of direct neighbours. This reduces the neighbours even further and thus, decreases the overall computation effort. Example AFs created with the level-based script can be seen in Figure 4.6

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

Figure 4.6: Example AF generated with grid-based approach

clustering The random-based, grid-based and level-based only generate concrete AFs. To be able to generate the abstract AFs based on the concrete ones, we created another independent script. Since clusters make a lot more sense when respecting the locality of the arguments, we cluster arguments which are next to each other (e.g. direct neighbours) by traversing the AF via the attacks and adding the argument if we visit it the first time. 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 Usage and Settings

We implemented the previously mentioned encodings and algorithm into the software called ClustArg. The tool uses Python as a 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, i.e. `SETS`, `CHECK`, `CONCRETIZE` and `FAITHFUL`. Additionally, we need to add the corresponding data via flags, which we can use to change the program's behavior.

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

```
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 in the paragraph below.

Spurious/Faithful check To check if an abstract AF is faithful or spurious, the program is called with the `CHECK` argument and 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 following arguments are again the path to the abstract AF, 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 more control over which arguments are concretized to obtain faithfulness, the program must be called with the `CONCRETIZE` flag.

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

In addition to the abstract and concrete AF paths, 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 complete 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 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 arguments that need to be concrete are listed after the `-p` tag and 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 turned off with the following flag.

```
-noref
```

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

```
-vis
```

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

4.2.1 Input Format

The input format for the AFs has a clear structure and is divided into four parts: i.e. *header*, *attacks*, *orphans* and *clusters*. The optional parts (i.e., *orphans* and *clusters*) are introduced with the `--<name>--` tag. Each tag is defined and the usage of the part is explained in the corresponding paragraph. If an optional part is empty, e.g. we have a concrete AF without clusters, we can simply omit the part by not including the tag. The singletons and clusters need to be indexed consecutively with positive integers, i.e., letters and symbols are not allowed.

Header The header is the first line of the input file and is the unique "p-line". Here we specify the amount of singletons `<n>` present in the AF.

```
p af <n>
```

The amount `<n>` does not include the amount of clusters from the specific AF and has to be a positive integer greater than 0. The abstracted arguments that are clustered are not included in the amount `<n>` as well.

Attacks Next, we define the attacks of the AF. Every attack is defined on a unique line and consists of a source `<s>` and a target `<t>`, which are the numerical indices of the arguments where `<s>` attacks `<t>`.

```
<s1> <t1>
<s2> <t2>
<s3> <t3>
<s4> <t4>
```

Orphans If an argument is not referenced in the attacks, i.e. the argument does not attack any other argument and is not attacked by any other argument, we call it an orphan. Orphans are not required coercively and are defined in the orphans section with the `--attackless--` tag. Every orphan is defined on a unique line and is the numerical indices of the argument.

```
--attackless--
<a1>
<a2>
<a3>
<a4>
```

Clusters The next optional part are the clusters. The tag is the `--cluster--` tag and each cluster is defined by a numerical positive integer and the indices of the arguments that are inside the cluster.

```
--cluster--
<c1> <- <a1> <a2> <a3>
<c2> <- <a4> <a5> <a6>
<c3> <- <a7> <a8> <a9>
```

The cluster `<c>` is followed by an arrow pointing to the cluster `<-`, followed by a list of argument indeces separated by spaces.

Comments Finally, we can use comment lines. A comment is marked as such, with a hashtag `#` at the beginning of the line. Every character followed by the hastag is ignored by the parser.

```
# This is a comment
```

Example 13. *Let us define $\hat{G} = (\hat{A}, \hat{R})$ to be an abstract AF depicted in Figure 4.7, with the arguments $\hat{A} = \{a, b, c, \hat{h}\}$ where the cluster $\hat{h}$ contains the arguments d, e and f. The attacks of the cluster are defined as $\hat{R} = \{(a, b), (\hat{h}, b), (\hat{h}, \hat{h})\}$.*

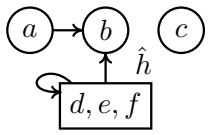


Figure 4.7: Abstract AF $\hat{G}$

We can translate the AF into the following input format. The arguments are mapped to numerical positive numbers, i.e., $a \to 1$, $b \to 2$, $c \to 3$, $d \to 4$, $e \to 5$, $f \to 6$ and $\hat{h} \to 7$.

```
p af 3
# This is a comment
1 2
7 2
7 7
--attackless--
3
--cluster--
7 <- 4 5 6
```

5 Experimental Evaluation

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. Next, we split up the experiment into two programs, the faithful/spurious check in Section 5.3 and the concretizing arguments program in Section 5.4. For every program, we ran test-run instances (i.e., a test-run is a single program invocation, solving a specific task) according to the two approaches, i.e., BFS and DFS and investigated further the efficiency of the applied refinements to the corresponding semantics. Furthermore, we compared all the test-runs of the specific program and visualized the runtimes according to the semantics. For the faithful/spurious check program, we also compared the two extension mapping approaches.

5.1 Generation of Input Instances

We generated the experiment instances with the previously mentioned scripts in Section 4.1. To have a 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 versatility 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 Experimental 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 Checking Faithulness

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 approaches to check for faithfulness, i.e., BFS and DFS. Both were tested and showed different results according to the AFs generation procedure. Note that we had a total amount of 450 test-runs, split up into 9 plots (i.e., each plot has 50 test-runs, as we have 25 test instances per generator approach, each of which was executed twice: one with refinements and once without). The plots are structured in a way, s.t. every step on the y-axis represents a test-run instance (i.e. a single invocation of the program to solve a specific task) and the x-axis represents the runtime of the test-run instance in seconds. 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 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 is less. Thus, BFS and DFS are similar.

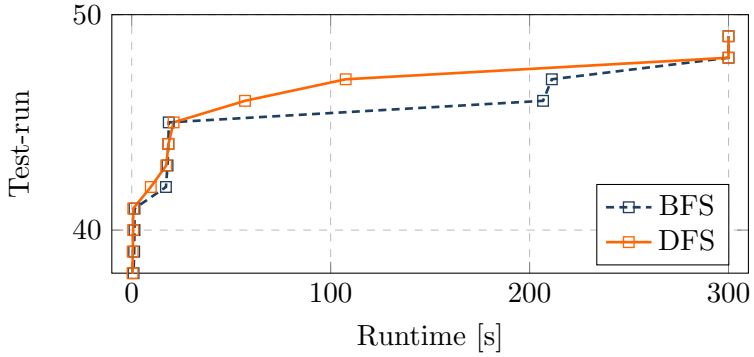


Figure 5.1: Runtime of random-based AFs over conflict-free semantics

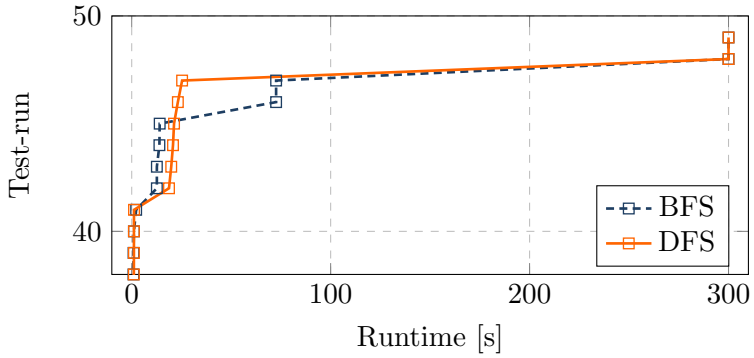


Figure 5.2: Runtime of random-based AFs over admissible semantics

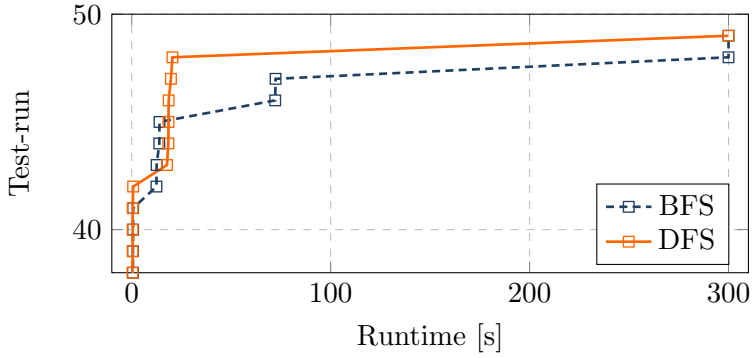


Figure 5.3: Runtime of random-based AFs over stable semantics

BFS and DFS differences increase 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. Due to the grid-based AF generation procedure, we created AFs with many more semantic extensions than on

the random-based approach. Since BFS needs to compute every extension to prove 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 test-run instances were computed randomly, the chance of generating a spurious AF is more likely, the DFS algorithm is favored.

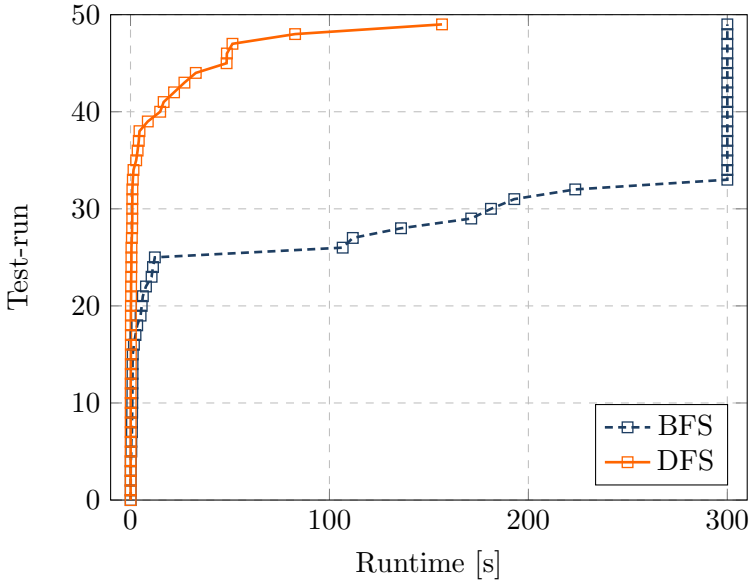


Figure 5.4: Runtime of grid-based AFs over conflict-free semantics

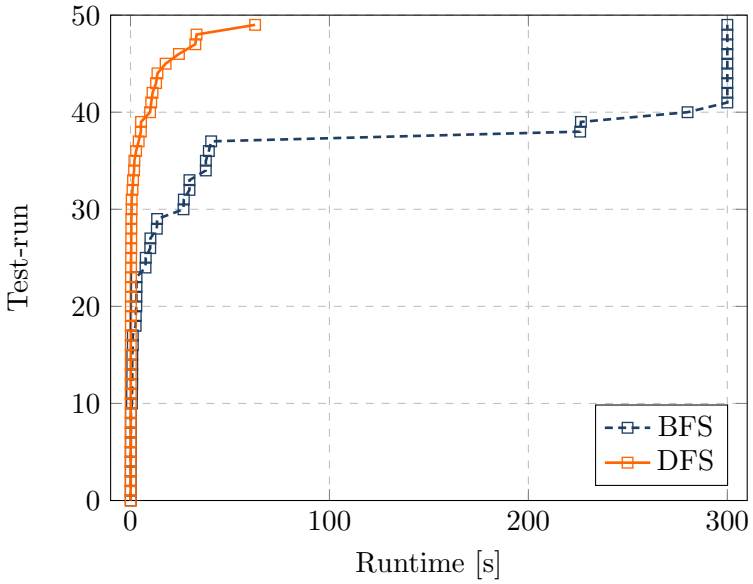


Figure 5.5: Runtime of grid-based AFs over admissible semantics

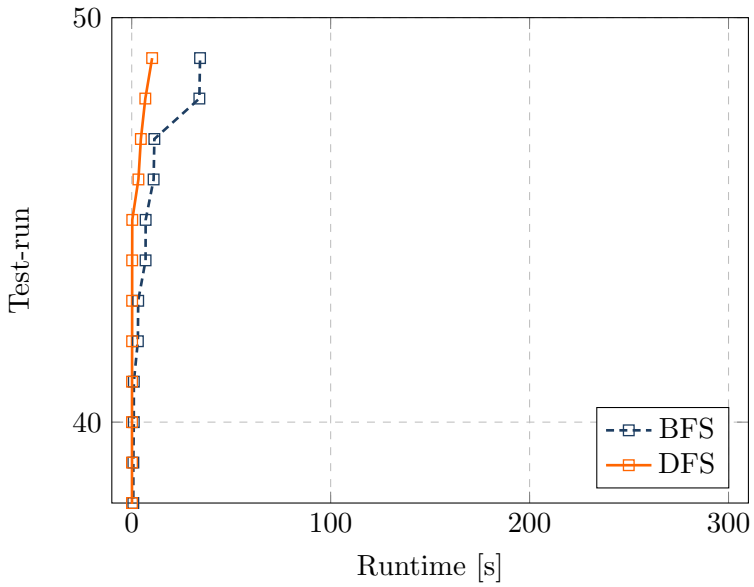


Figure 5.6: Runtime of grid-based AFs over stable semantics

The difference increases even further when the AF generator approach is changed 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.

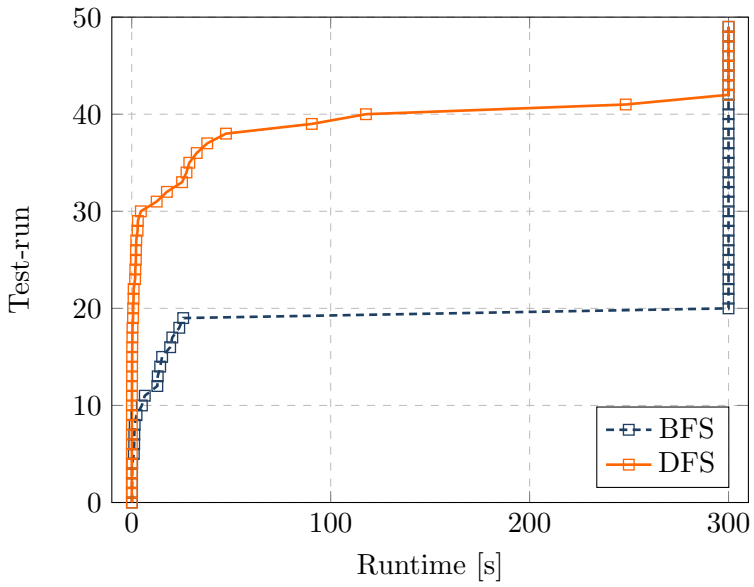


Figure 5.7: Runtime of level-based AFs over conflict-free semantics

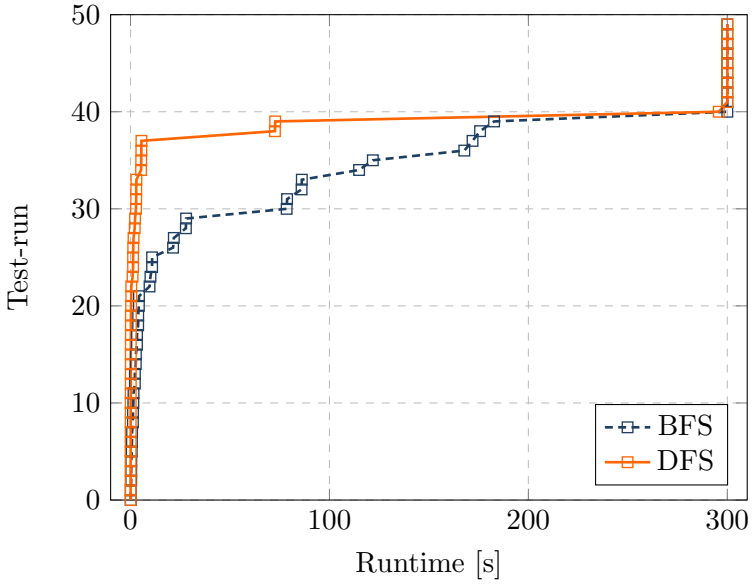


Figure 5.8: Runtime of level-based AFs over admissible semantics

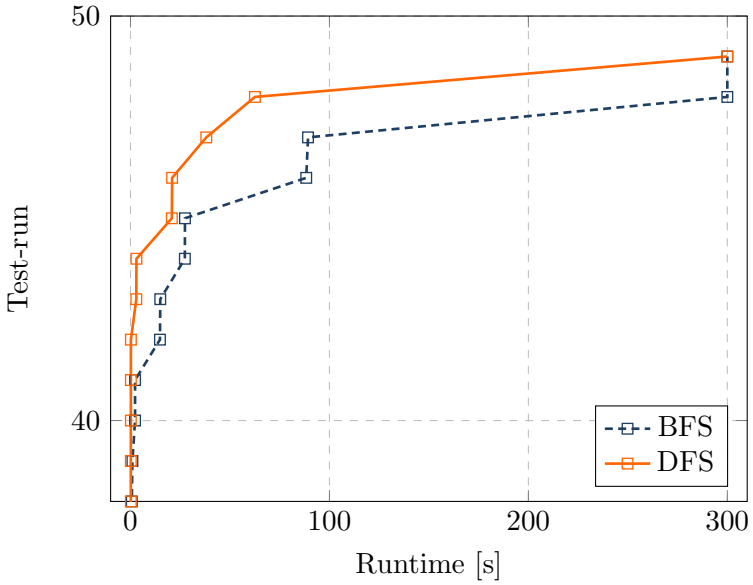


Figure 5.9: Runtime of level-based AFs over stable semantics

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 on the BFS approach, and in the few cases where BFS is more efficient than DFS, it is by a minor factor. Especially with the AFs generated with the level-based procedure, DFS is way more efficient. This is because, on level-based AFs, we have far more extensions

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

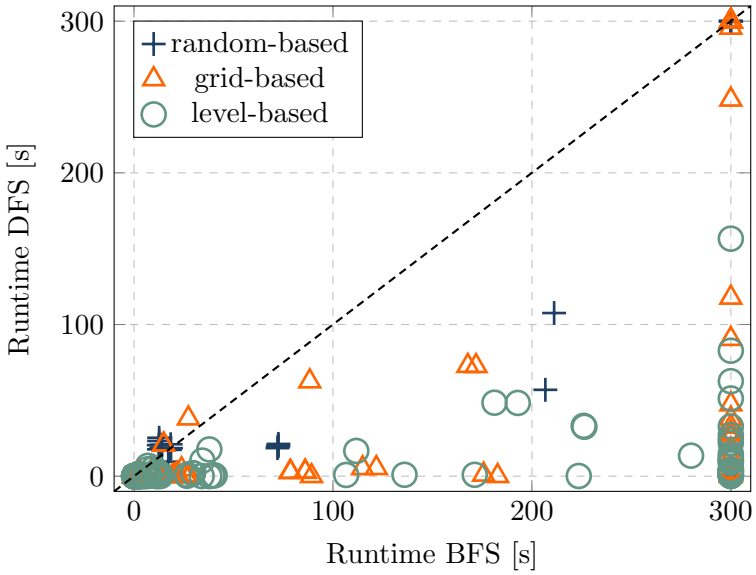


Figure 5.10: Runtime of BFS and DFS grouped by generator approach

5.3.2 Impact of Refinements

We implemented different refinements for each semantics. As data has shown, the refinements made for conflict-free semantics have a significant impact, as shown in Figure 5.11 and Figure 5.12. Note that we had a total amount of 450 test-runs, but we only show the interesting conflict-free results with 50 test-runs per AF generator. The plots are structured in a way, s.t. every step on the y-axis represents a test-run instance (i.e. a single invocation of the program to solve a specific task) and the x-axis represents the runtime of the test-run instance in seconds. The refinement reduces the runtime significantly, especially for AFs with many conflict-free sets. We got some test-runs for the grid-based AFs, where we obtained a better result without the refinement. 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 checks, which leads to an increase in runtime that the DFS algorithm, without 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 only had a minor impact. We speculate that the SAT-Solver discarded the refinement

for optimization reasons or the overhead of increasing the Boolean formula, which led to no significant improvement.

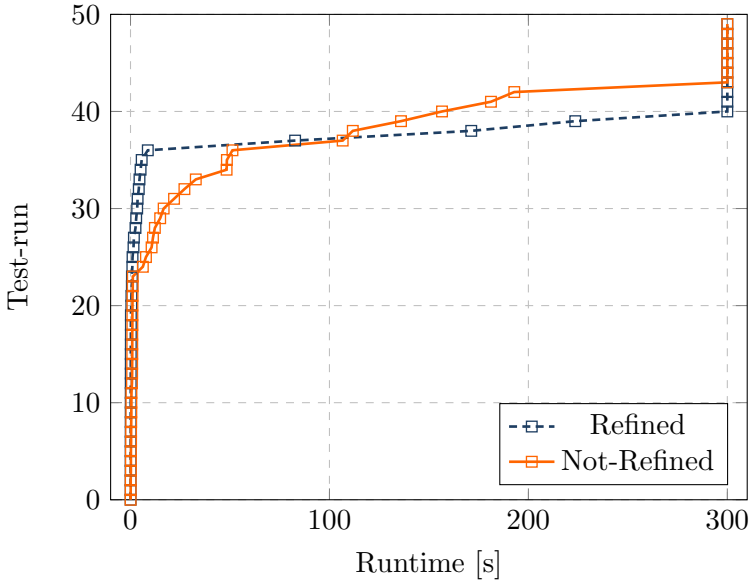


Figure 5.11: Runtime of grid-based AFs over conflict-free semantics

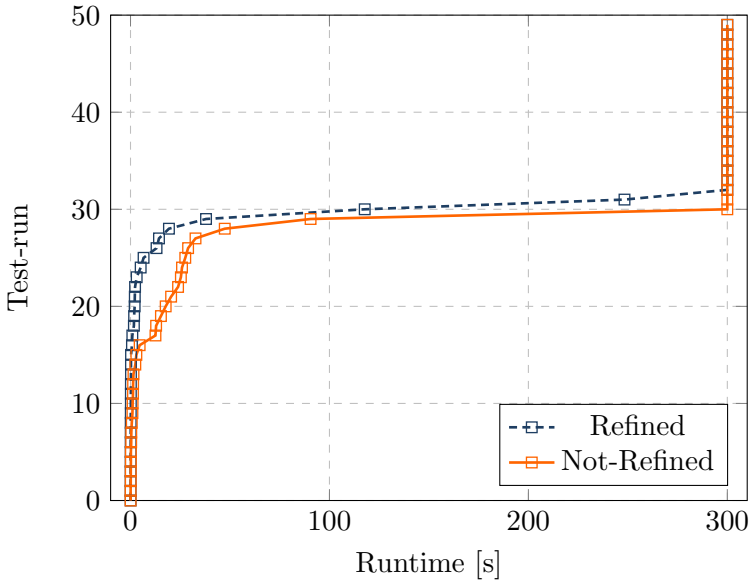


Figure 5.12: Runtime of level-based AFs over conflict-free semantics

5.3.3 Comparison of all Test-runs

In this section, we compare the data over all the test-run instances. In Figure 5.13, we can see the performance of the refinements against no refinements. The scatter plot represents for every datapoint the runtime of the testcase with the refinements as the x-value and the runtime without the refinement as the y-value. There is no significant difference except for some outliers for admissible and stable. We hypothesize that due to the refinement, we pushed the SAT-Solver in a specific 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 fundamental impact on the runtime. On the other hand, the refinement made for conflict-free semantics significantly improved the runtime, besides some outliers.

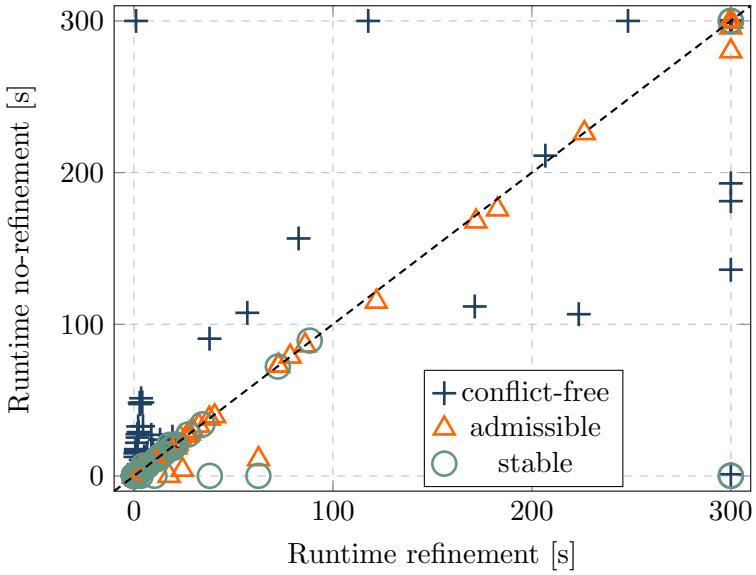


Figure 5.13: Performance comparison of Refinement and No-Refinement by Semantics

Furthermore, we look at how many test-runs did not run into a timeout depending on the semantics and the mean runtime of the passed test cases. We ran a total amount of 900 test-runs, split up into 3 semantics leading to 300 test-runs for each semantics and depicted the runtime of the solved instances. Note, that the y-axis represents the numeration of test-runs starting at 100, since all the test-runs below were solved in seconds. The x-axis shows the runtime of the test-run. In Figure 5.14 we can observe that the most solvable semantics is the stable semantics. Following admissible semantics, the least solved test-runs have 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.

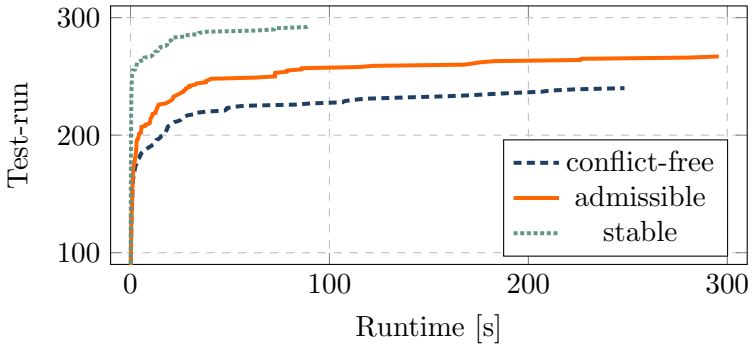


Figure 5.14: Solved test-runs depending on semantics

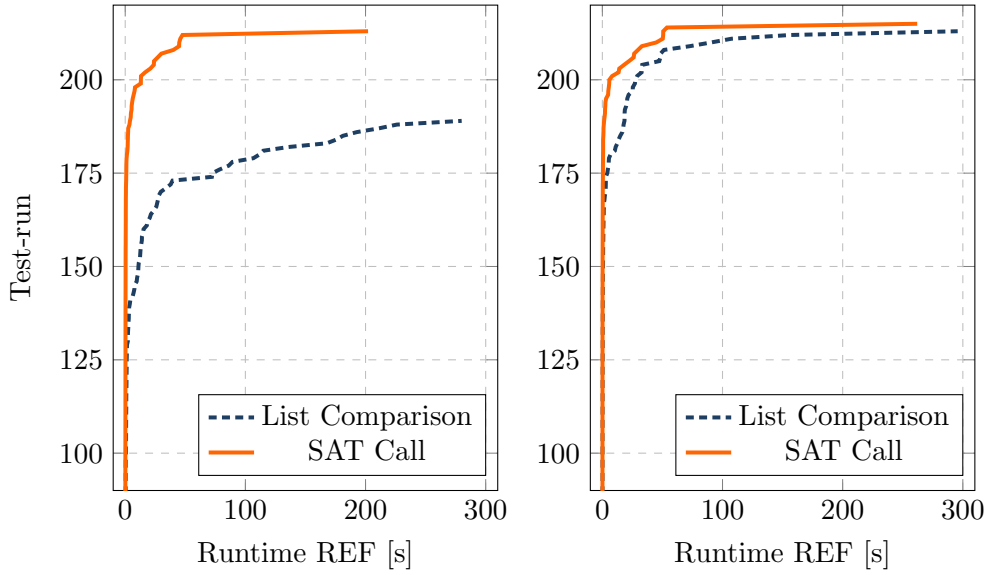
The data was arranged in a tabular format to enable a more comprehensive comparison from a higher-level viewpoint. We computed the mean value of the test-runs depending on the AF number of arguments and semantics. Furthermore, we excluded the runtime from the test runs that ran into a timeout and stated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

5.3.4 Comparison of Faithful Check Approaches

We tested the difference between the two faithful/spurious check approaches and compared them in 5.15. Data shows that the SAT call approach is more efficient than the list comparison. This is because the list comparison needs to deconstruct the clusters. This is especially bad for AFs with clusters that have an enormous amount of argument.



(a) BFS algorithm (b) DFS algorithm

Figure 5.15: Comparison of faithful/spurious algorithms

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 5.16 the conflict-free semantics is shown, in Figure 5.17 we depicted the admissible semantics and in Figure 5.18 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 not only stands no chance against DFS, but in most instances, it does not terminate before the timeout. This is especially critical for the admissible

and stable semantics.

For the runtime on the DFS approach, most of the AFs created with the random-based procedure were trivial to solve, and some ran into a timeout once the complexity of the AFs increased.

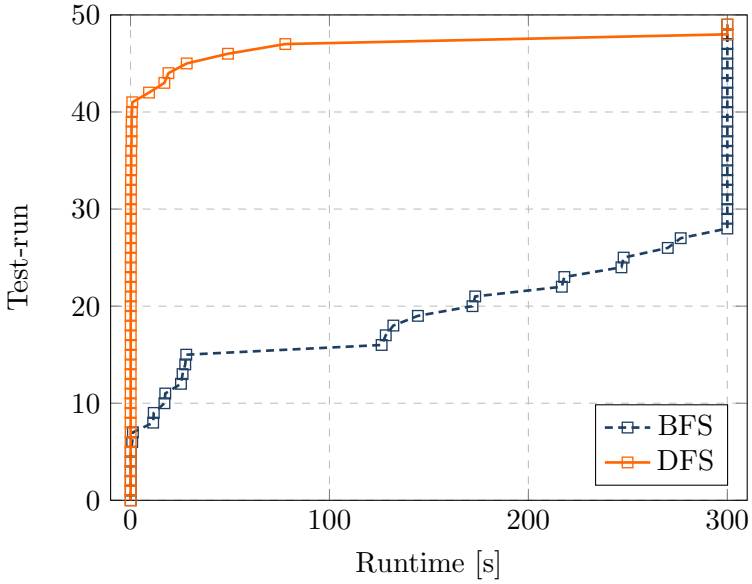


Figure 5.16: Runtime of random-based AFs over conflict-free semantics

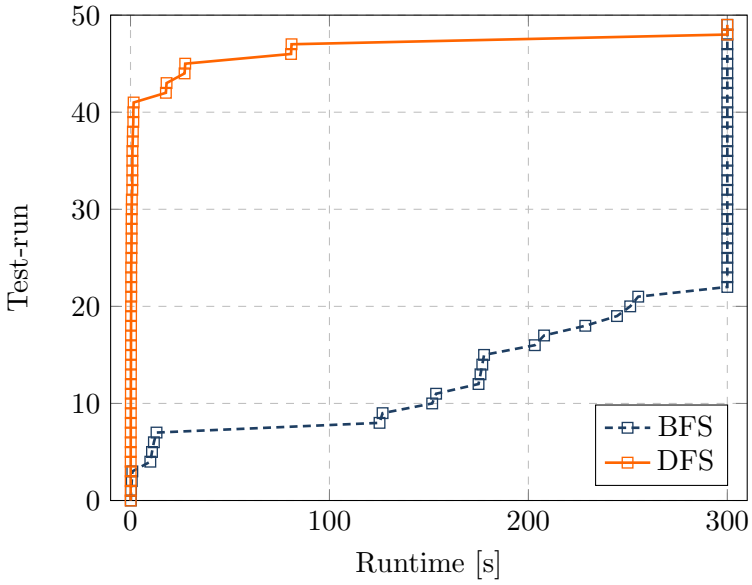


Figure 5.17: Runtime of random-based AFs over admissible semantics

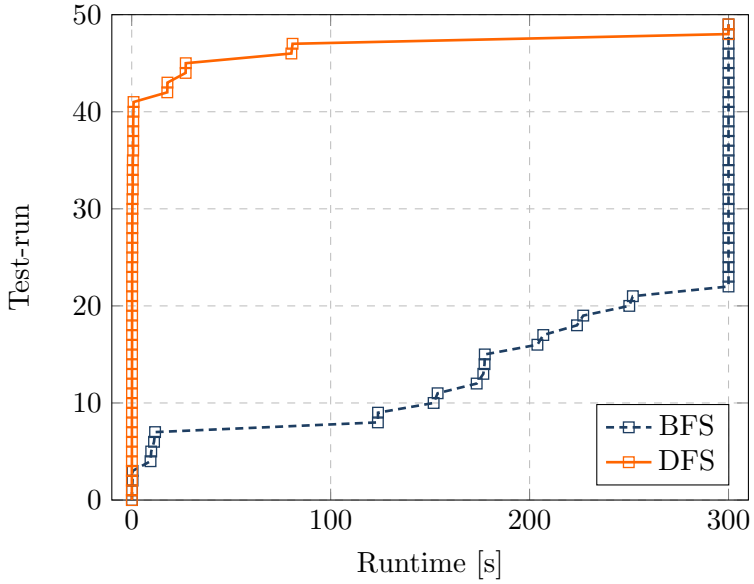


Figure 5.18: Runtime of random-based AFs over stable semantics

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 5.19, Figure 5.20 and Figure 5.21.

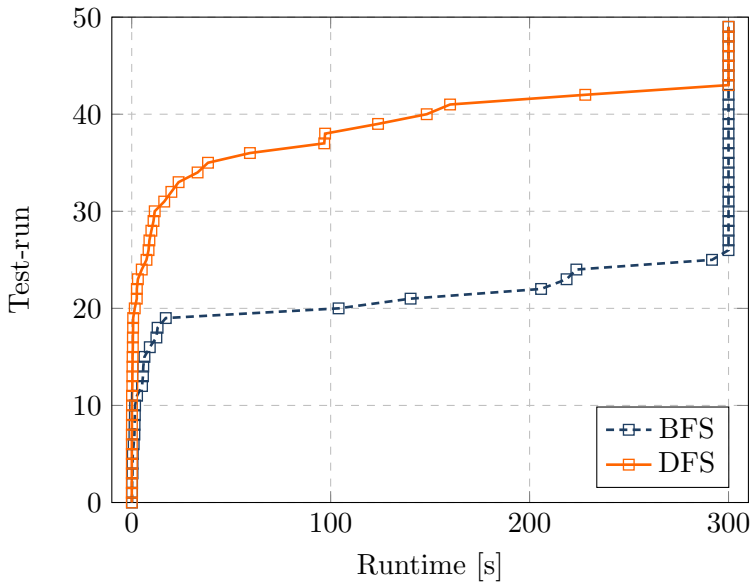


Figure 5.19: Runtime of grid-based AFs over conflict-free semantics

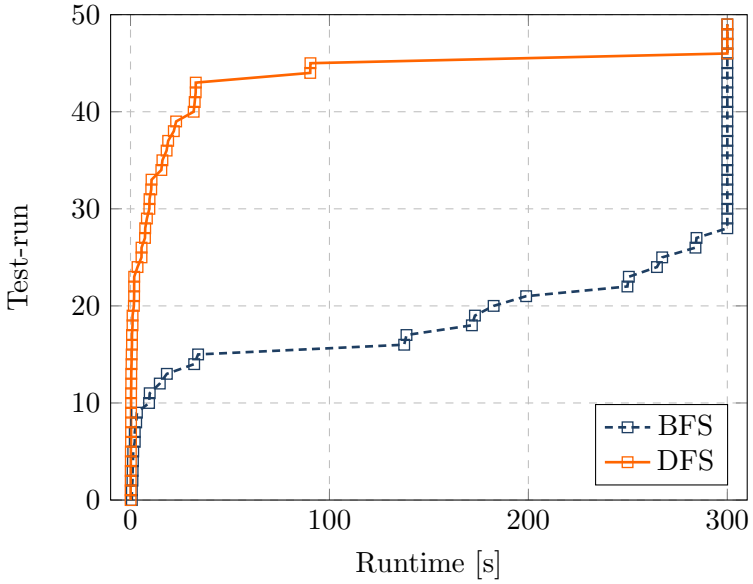


Figure 5.20: Runtime of grid-based AFs over admissible semantics

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

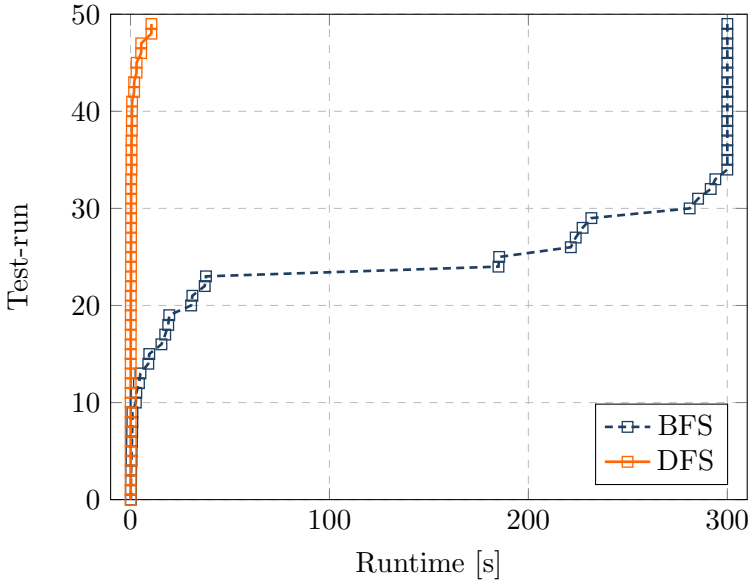


Figure 5.21: Runtime of grid-based AFs over stable semantics

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 5.22, admissible in Figure 5.23 and stable in Figure 5.24. Also, 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 more challenging 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 that remain spurious even after concretizing arguments.

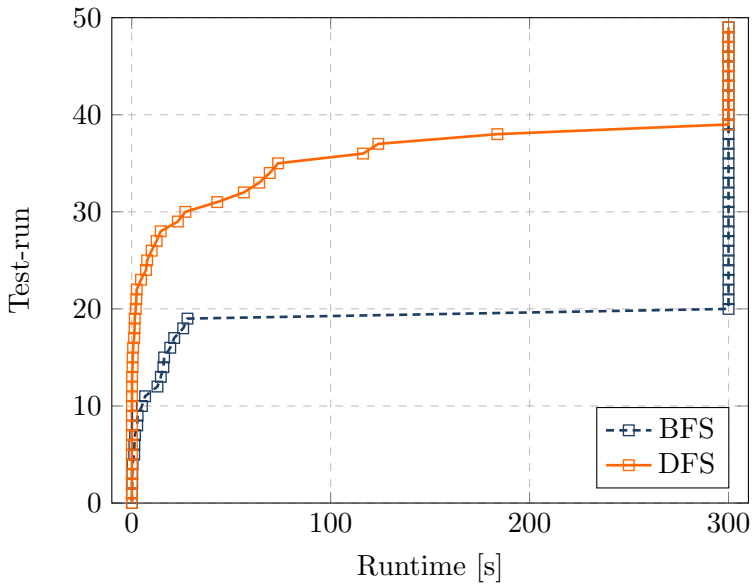


Figure 5.22: Runtime of level-based AFs over conflict-free semantics

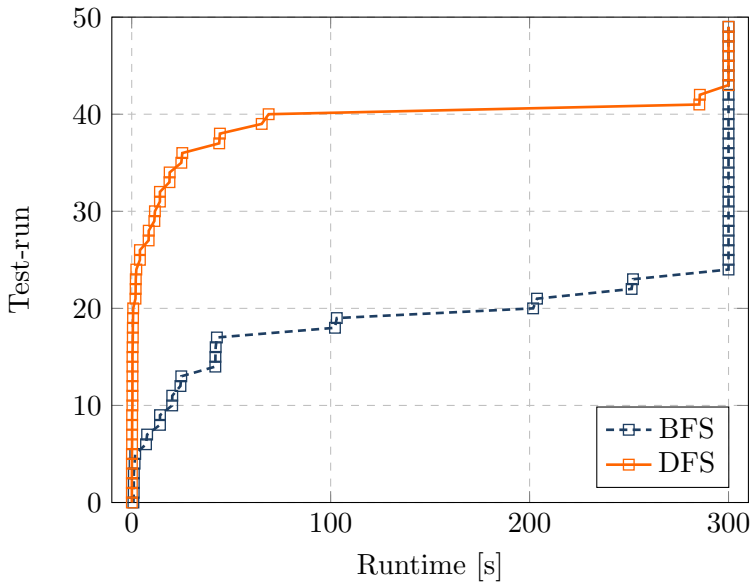


Figure 5.23: Runtime of level-based AFs over admissible semantics

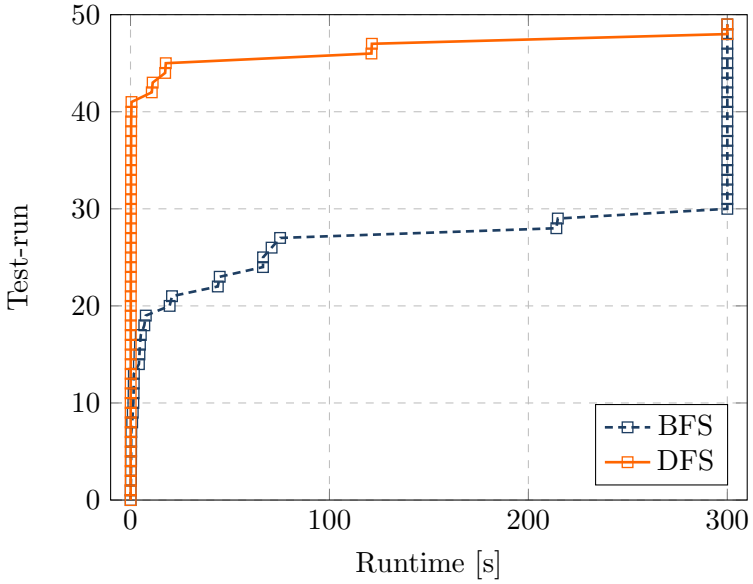


Figure 5.24: Runtime of level-based AFs over stable semantics

Finally, we grouped the data by the AF generator procedure and plugged it into a single plot. In Figure 5.25, 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.

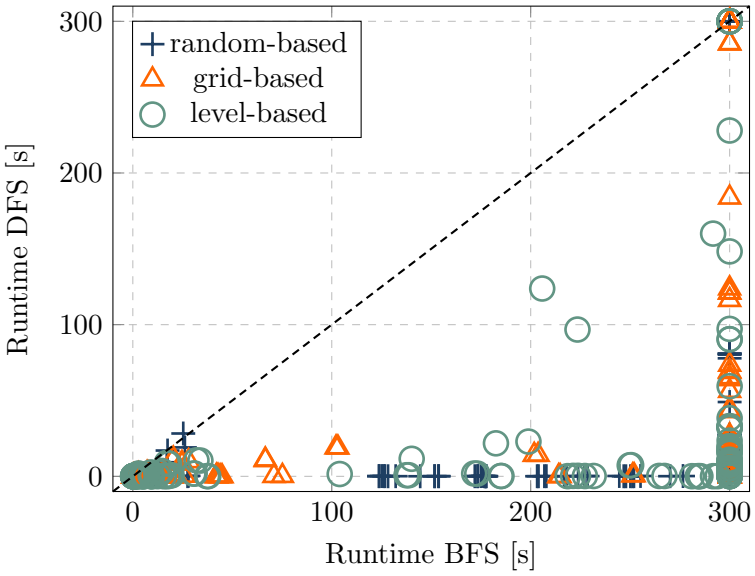


Figure 5.25: Runtime of BFS and DFS grouped by generator approach

5.4.2 Impact of Refinements

The results of 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 significantly impact the runtime. 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 added to the Boolean formula. However, it is more of 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 5.26 for the AFs generated with the grid-based procedure and for the AFs generated with the level-based procedure in Figure 5.27. 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.

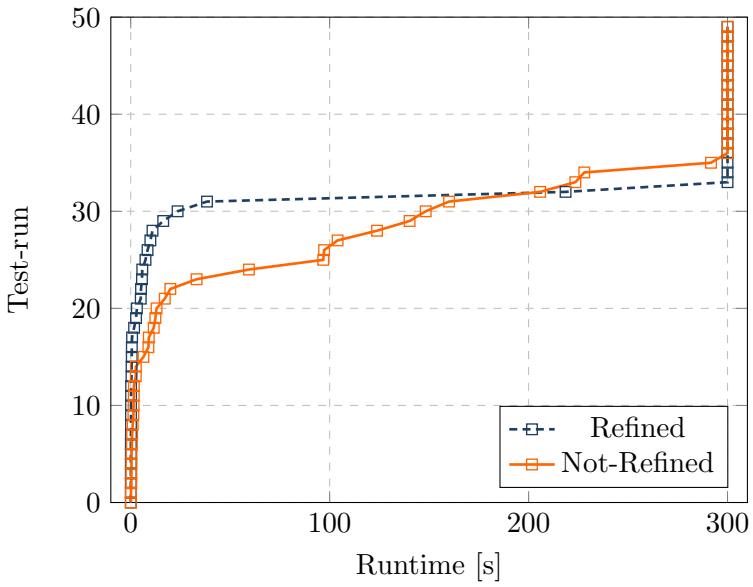


Figure 5.26: Runtime of grid-based AFs over conflict-free semantics

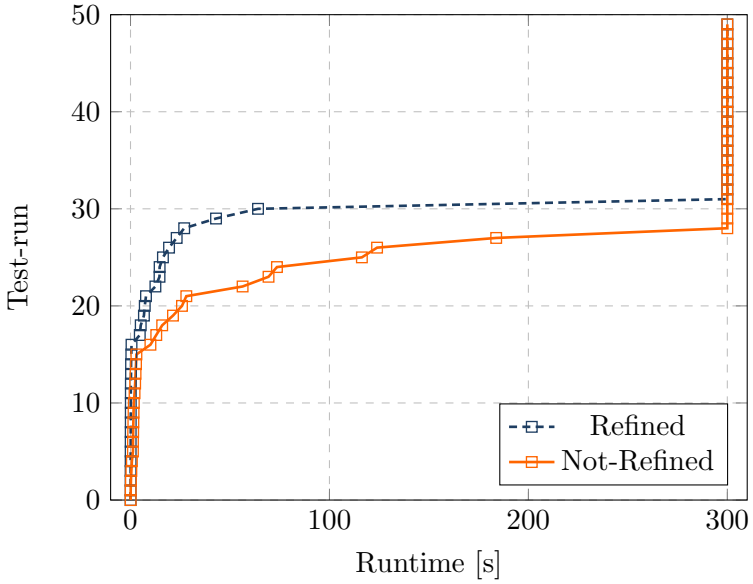


Figure 5.27: Runtime of level-based AFs over conflict-free semantics

5.4.3 Comparison of all Test-runs

In this section, we compare the efficiency of the refinements to the corresponding semantics in Figure 5.28. As the plot shows, the refinements for admissible and stable had no significant impact. Nevertheless, for conflict-free semantics, refinement had a significant impact. Besides five outliers, the refinement decreased the runtime significantly.

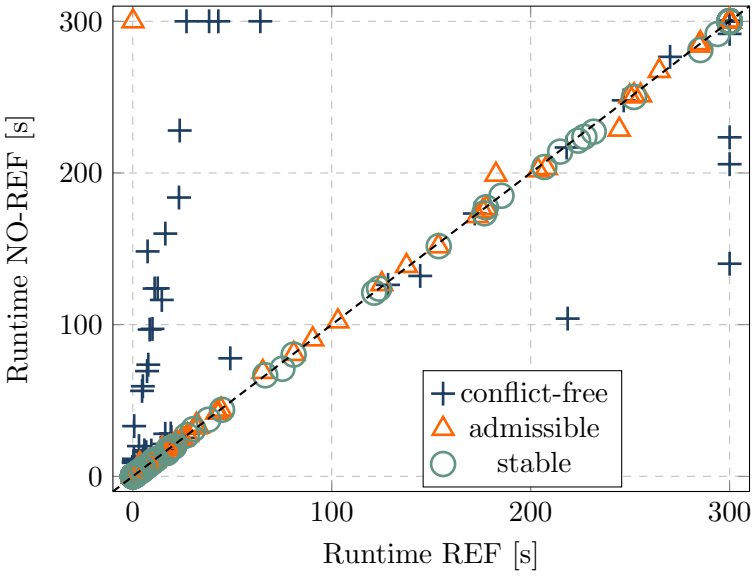


Figure 5.28: Performance comparison of Refinement and No-Refinement by Semantics

In Figure 5.29 we visualize the runtimes of the test-run instances which did not time-out. We split the data into the three implemented semantics and observed that stable semantics is again the semantics, with the least amount of timeouts. But other than for the faithful/spurious check, for the concretizing arguments program, conflict-free and admissible semantics are very similar when compared to the runtime of the solved test-run instances.

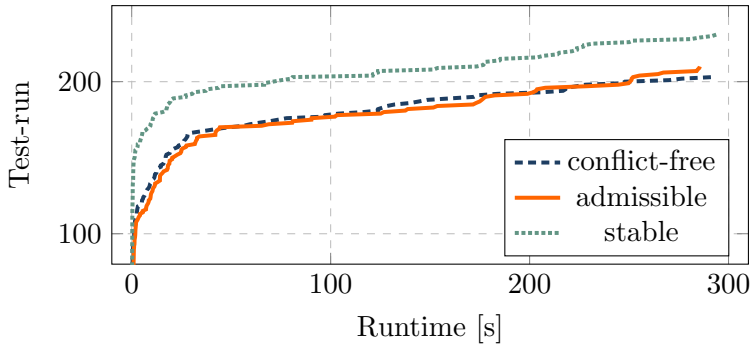


Figure 5.29: Solved test-runs depending on semantics

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 that did not terminate before the timeout of 300s. As the table shows, there is a direct correlation between the increase in 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 [27]. Since then, a tool was created (absarg-clustering) [1] 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) [2] runs a competition, aiming to assess the state of the art in practical systems for reasoning in central argumentation formalisms. The technique of abstraction is a widely used concept in computer science where much research is being done at the moment. It can be applied to model building and problem solving with the use of ASP. Last but not least, a similar SAT-Solver experiment was conducted with concrete argumentation frameworks.

Absarg-clustering The project called *absarg-clustering* is a tool to simplify the Dung-Style argumentation frameworks by partitioning arguments into clusters. The tool was developed in 2022 and is available on GitHub under the open-source license GPL-3.0. It is a similar implementation to our tool but uses Answer Set Programming (ASP) with the clingo solver [16] in combination with Python3. The tool covers solutions to various problems, e.g., computing classical extensions, checking whether an extension is spurious, or finding spurious extensions from an abstract AF.

After some minor test-runs with the faithful/spurious check, we can see that it performs similar to the BFS implementation of our tool with no refinements and the SAT-based check. Since DFS is faster than BFS in the most cases, we speculate that our tool is more efficient than *absarg-clustering* when choosing the best settings. A thorough empirical evaluation could be an interesting direction for future work.

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 part of

[1] https://github.com/rbankosegger/absarg-clustering
[2] https://argumentationcompetition.org/2023/solvers.html

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 worldwide 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` [3] and second place was awarded to the solver from the University of Helsinki with the name μ-`Toksia` [4] [22].

Abstraction in other fields Abstraction is not only used in combination with argumentation frameworks, but does also find the application in model building and problem solving. It is an important technique to reduce the complexity of the program and still being able to find a solution. Abstraction is also explored in the context of non-ground Answer Set Programming (ASP) [26]. An often used example in the concept of ASP is Sudoku, which is a puzzle played on a 9×9 grid. With abstraction, we can encode the rules of the puzzle by considering higher-level constraints like focusing on rows, columns and blocks.

Abstraction is also applied in the context of model-checking [9]. In this paper, abstraction is applied to the state explosion of symbolic model checking. In the presented method, an initial abstract model is generated which may be erroneous. This erroneous abstraction is refined over and over until no counterexamples can be found. This procedure can be compared to our implementation of concretizing arguments until faithfulness is reached.

Encodings of AF semantics The usage of SAT-Solver in combination with argumentation frameworks was already studied previously [5]. In this paper, they were interested in the problem which consists in deciding whether a set of arguments is accpetable under a given semantics. The investigation was done exclusively on concrete AFs. We adapted the formulas to compute also abstract semantics.

[3] https://github.com/crillab/crustabri
[4] https://bitbucket.org/andreasniskanen/mu-toksia

7 Conclusions

This thesis aimed to investigate the efficiency of SAT-Solvers within the context of clustered argumentation frameworks. We applied the SAT-Solver z3 to solve four different problems. Our tool is called ClustArg and is available on GitHub under the open-source MIT license. Furthermore, we designed refinements covering all implemented semantics, i.e., conflict-free, admissible, and stable. These refinements are supposed to speed up the process of solving the task.

Additionally, we developed two different algorithms, i.e., BFS and DFS, which operate differently, and thus, one dominates the other depending on the input AFs. We ran benchmarks to determine the effect of the refinements, which algorithm is more dominant, and under which circumstances. In order to cover a variety of different AFs, we used three different AF generator procedures, i.e., random-based, grid-based, and level-based.

In this concluding chapter, the key findings are summarized, and their implications of practicality are discussed. To wrap up this research, we also stated the efficiency of the refinements and under which circumstances one algorithm dominates the other. Finally, we will discuss the limits and issues of our implementation and how it could be improved in future works.

Refinements Let us begin by discussing the impact of the refinements. As data has shown, the refinements for admissible and stable had a minimal impact. We hypothesize that the SAT-Solver has to invest more computation time for a single extension by adding the refinement to the Boolean formula and almost doubling it. Nevertheless, due to the refinement, we are reducing the total amount of extensions that need to be calculated. We assume these factors cancel out, and the runtime is not significantly improved. However, for conflict-free, the refinements contribute to a big improvement in the runtime for most instances. There are some outliers at spurious AFs, which can be attributed to a spurious extension found by the DFS algorithm. For faithful AFs, the refinement will always contribute to a lower runtime.

BFS vs DFS Next, we will debate the choice of faithful/spurious check algorithm. The BFS approach is the algorithm with more stability, computing independently from the seed of the SAT-Solver. This is because BFS calculates all the semantic extensions before checking for spuriousness. Therefore, the order in which the extensions are calculated does not matter. BFS shows better results than DFS if the input AFs are faithful since there are no context switches. The same holds for AFs with very few semantics extensions. However, DFS dominates BFS in every other aspect. DFS is highly dependent on

the seed of the SAT-Solver and can show spuriousness more efficiently without computing all of the faithful semantic extensions. The DFS algorithm scales well with the size of the AFs by finding solutions even for AFs with 30 arguments.

Nevertheless, for faithful AFs, the runtime is the same as BFS, with an additional overhead of the context switches. We recommend using the BFS algorithm for small AFs (with fewer arguments than 15) and if the probability of faithfulness is high. For bigger AFs (with more than 15 arguments), we recommend using DFS.

Limits and issues We pushed ClustArg to the limits and analyzed the issues. First of all, the tool does not scale well with the size of the AFs. If the number of arguments increases, the computation time of generating semantic extensions also increases, leading to a higher runtime for all the implemented programs. Furthermore, the concretizing arguments program is not guaranteed to find a solution. This is due to the restriction of the neighbors depth (which in our implementation is 2). By increasing the depth, s.t., the depth equals the number of arguments the AF has, we could guarantee to find a solution. However, it would also aggravate the last issue: exponential growth of the combination table when computing the concretized list. The concretized list defines the arguments to be concretized until faithfulness is reached and depends on the number of neighbors a spurious argument has. With each new neighbor, we increase the combination table by a factor of 2. Therefore, by increasing the depth of neighbors, we could generate a combination table taking up so much memory that the computation is infeasible to solve.

Future Work If the usage of SAT-Solvers should be preserved, the tool could be improved by substituting the current SAT-Solver (i.e., *z3*) with a more efficient one. Every year a SAT competition [1] is hosted and the most efficient SAT-Solver is awarded. Due to the modular structure of the implementation, exchanging the SAT-Solver is not too complex and would lead to a significant improvement.

The programs of ClustArg could also be extended in future work. The ability to build a faithful abstract AF based on a concrete AF would be a valuable improvement.

Furthermore, an empirical evaluation could be done, testing the efficiency of ClustArg compared to the related tool *absarg-clustering*.

[1] https://satcompetition.github.io/2024/results.html

Bibliography

[1] Katie Atkinson, Pietro Baroni, Massimiliano Giacomin, Anthony Hunter, Henry Prakken, Chris Reed, Guillermo Simari, Matthias Thimm, and Serena Villata. "Toward Artificial Argumentation." In: *Artificial Intelligence Magazine* 38.3 (2017), pp. 25–36. DOI: 10.1609/aimag.v38i3.2704.

[2] Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. "Maximum Satisfiabiliy." In: *Handbook of Satisfiability - Second Edition*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 929–991. DOI: 10.3233/FAIA201008.

[3] Pietro Baroni, Martin Caminada, and Massimiliano Giacomin. "An introduction to argumentation semantics." In: *Knowledge Engineering Review* 26.4 (2011), pp. 365–410. DOI: 10.1017/S0269888911000166.

[4] Pietro Baroni, Dov Gabbay, Massimiliano Giacomin, and Leendert van der Torre. *Handbook of Formal Argumentation*. Vol. 1. College Publications, 2018.

[5] James Delgrande and Torsten Schaub, eds. *Checking the acceptability of a set of arguments*. Springer Verlag, 2004, pp. 59–64.

[6] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, eds. *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009.

[7] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. "Answer set programming at a glance." In: *Communications of the ACM* 54.12 (2011), pp. 92–103. DOI: 10.1145/2043174.2043195.

[8] Elena Cabrio, Graeme Hirst, Serena Villata, and Adam Wyner. "Natural Language Argumentation: Mining, Processing, and Reasoning over Textual Arguments." In: *Dagstuhl Reports* 6.4 (2016). Ed. by Elena Cabrio, Graeme Hirst, Serena Villata, and Adam Wyner, pp. 80–109. DOI: 10.4230/DagRep.6.4.80.

[9] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample guided abstraction refinement for symbolic model checking." In: *Journal of the Association for Computing Machinery* 50.5 (2003), pp. 752–794. DOI: 10.1145/876638.876643.

[10] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures." In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. Ed. by Jeffrey D. Ullman Michael A. Harrison Ranan B. Banerji. Association for Computing Machinery, 1971, pp. 151–158. DOI: 10.1145/800157.805047.

[11] Phan Minh Dung. "On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games." In: *Artificial Intelligence* 77.2 (1995), pp. 321–357. DOI: 10.1016/0004-3702(94)00041-X.

[12] Wolfgand Dvořák and Paul E. Dunne. "Computational Problems in Formal Argumentation and their Complexity." In: *Handbook of Formal Argumentation*. Ed. by Pietro Baroni, Dov Gabbay, Massimiliano Giacomin, and Leendert van der Torre. College Publications, 2018, pp. 631–688.

[13] Uwe Egly, Sarah Alice Gaggl, and Stefan Woltran. "Answer-set programming encodings for argumentation frameworks." In: *Argument Computation* 1.2 (2010), pp. 147–177. DOI: 10.1080/19462166.2010.486479.

[14] Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. "SAT Competition 2020." In: *Artificial Intelligence* 301 (2021), p. 103572. DOI: 10.1016/J.ARTINT.2021.103572.

[15] Dov Gabbay, Massimiliano Giacomin, Guillermo R. Simari, and Matthias Thimm. *Handbook of Formal Argumentation*. Vol. 2. College Publications, 2021.

[16] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. "Theory Solving Made Easy with Clingo 5." In: *Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016)*. Ed. by Manuel Carro, Andy King, and Marina Neda De Vos Saeedloei. Vol. 52. Open Access Series in Informatics (OASIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016, 2:1–2:15. DOI: 10.4230/OASIcs.ICLP.2016.2.

[17] Martin Gogolla. "Towards Model Validation and Verification with SAT Techniques." In: *Algorithms and Applications for Next Generation SAT Solvers*. Ed. by Bernd Becker, Valeria Bertacoo, Rolf Drechsler, and Masahiro Fujita. Vol. 09461. Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2010, pp. 1–11. DOI: 10.4230/DagSemProc.09461.6.

[18] Guido Governatori, Michael Maher, Grigoris Antoniou, and David Billington. "Argumentation Semantics for Defeasible Logic." In: *Journal of Logic and Computation* 14 (2004), pp. 675–702. DOI: 10.1093/logcom/14.5.675.

[19] Frank van Harmelen, Vladimir Lifschitz, and Bruce W. Porter. *Handbook of Knowledge Representation*. Vol. 3. Foundations of Artificial Intelligence. Elsevier, 2008. URL: https://www.sciencedirect.com/science/bookseries/15746526/3.

[20] Simon Laurent. "Reasoning with Propositional Logic: From SAT Solvers to Knowledge Compilation." In: *A Guided Tour of Artificial Intelligence Research: Vol. II: AI Algorithms*. Ed. by Henri Prade Pierre Marquis Odile Papini. Springer, 2020, pp. 115–152. DOI: 10.1007/978-3-030-06167-8_5.

[21] Leonardo de Moura and Nikolaj Bjørner. "Z3: an efficient SMT solver." In: *Tools and Algorithms for the Construction and Analysis of Systems* 4963 (2008), pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[22] 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.* Ed. by Michael Thielscher Diego Calvanese Esra Erdem. 2020, pp. 800–804. DOI: 10.24963/kr.2020/82.

[23] 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.* Autonomous Agents and Multi-Agent Systems 2002. Bologna, Italy: Association for Computing Machinery, 2002, pp. 394–401. DOI: 10.1145/544741.544835.

[24] Nils Przigoda, Robert Wille, Judith Przigoda, and Rolf Drechsler. *Automated Validation & Verification of UML/OCL Models Using Satisfiability Solvers.* Springer, 2018. DOI: 10.1007/978-3-319-72814-8.

[25] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition).* Pearson, 2020.

[26] Zeynep Saribatur, Thomas Eiter, and Peter Schüller. "Abstraction for non-ground answer set programs." In: *Artificial Intelligence* 300 (2021), p. 103563. DOI: 10.1016/j.artint.2021.103563.

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

[28] Francesca Toni. "A tutorial on assumption-based argumentation." In: *Argument and Computation* 5 (2014), pp. 89–117. DOI: 10.1080/19462166.2013.869878.

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

[30] Georg Weissenbacher and Sharad Malik. "Boolean Satisfiability Solvers: Techniques and Extensions." In: *Software Safety and Security - Tools for Analysis and Verification.* Ed. by Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann. Vol. 33. IOS Press, 2012, pp. 205–253. DOI: 10.3233/978-1-61499-028-4-205.