FernUniversität in Hagen

Preprocessing Methoden für Abstrakte Argumentation

Nils Gosing

Lehrgebiet: Künstliche Intelligenz
Prof. Dr. Thimm

15. November 2022

Inhaltsverzeichnis

1 Einleitung

Die Argumentationstheorie hat angesichts der Forschung an Künstlicher Intelligenz sowie als interdisziplinäre Schnittstelle zu geisteswissenschaftlichen Forschungsgebieten, zunehmend an Bedeutung gewonnen. Insbesondere die Forschungsarbeit von Dung (vgl. [22],[6]) im Bereich der formalen Argumentation trug zu diesem Aufmerksamkeitsschub bei. In seiner Arbeit entwickelte Dung mit dem *Abstract Argumentation Framework* (AF) ein sehr einfaches, aber zugleich ausdrucksstarkes Modell. Ein AF kann als gerichteter Graph dargestellt werden, dabei repräsentieren Knoten Argumente und gerichtete Kanten repräsentieren Angriffe zwischen Argumenten. Um diese Strukturen interpretieren zu können, werden durch Semantiken Mengen an akzeptierten Argumenten definiert.

Die Berechnungsprobleme, die in der formalen Argumentation auftreten, sind für die meisten Semantiken NP-vollständig. Allgemein werden die unterschiedlichen Problemstellungen in der formalen Argumentation durch Solver gelöst und eine Vielzahl an unterschiedlichen Herangehensweisen wurden bereits entwickelt (vgl. [15]). Um immer komplexere Probleminstanzen in kürzerer Zeit zu verarbeiten, wurden im Zuge dessen bereits unterschiedliche Methodiken wie z.B. aus den Bereichen der Parallelisierung (vgl. [18]) oder des Clustering von AFs (vgl. [21], [7]) entwickelt.

Diese Arbeit greift die Idee aus [24], dem Preprocessing eines AFs, auf. Anstelle des direkten Lösens der unterschiedlichen Problemstellungen, wird der AF erst in einem Vorverarbeitungsschritt vereinfacht. Bei diesem Preprocessing-Schritt werden gewisse Angriffsmuster in einem AF gesucht, die Vereinfachungen zulassen ohne die Mengen an akzeptierten Argumenten zu verändern. Im Mittelpunkt dieser Arbeit steht die Untersuchung der Anwendungsreihenfolge der Muster, sowie die Entwicklung einer effizienten Suchheuristik zur Identifizierung der Muster.

Die Kapitel 2 und 3 präsentieren den formalen Hintergrund einschließlich die Vereinfachung von AFs. In Kapitel 4 wird die Anwendungsreihenfolge von Mustern genauer untersucht. Aufbauend darauf wird im nachfolgenden Kapitel unter anderem die Funktionsweise der in dieser Arbeit entwickelten Heuristik näher erläutert. Hieran schließt sich in Kapitel 6 eine experimentelle Evaluation der Methodik sowie der Heuristik an. Das Kapitel 7 setzt sich mit der Existenz weiterer Muster auseinander. Abschließend werden die gewonnenen Ergebnisse in Kapitel 8 zusammengetragen.

2 Formale Argumentation

Ein *Abstract Argumentation Framework*, besteht aus einer Menge von nicht näher spezifizierten Argumenten und einer Angriffsrelation, das den Angriff eines Arguments auf ein anderes Argument festlegt. Auf datenstruktureller

Ebene kann ein solches Framework als gerichteter Graph dargestellt werden.

Definition 2.1 (Abstract Argumentation Framework)**.** Ein *Abstract Argumentation Framework* (AF) ist ein Tupel $\langle Ar, att \rangle$ mit einer endlichen Menge von Argumenten Ar und einer Angriffsrelation $att \subseteq Ar \times Ar$.

Dieses Framework definiert die Struktur bzw. die Syntax zwischen den Argumenten. Es macht keine Angaben darüber, ob ein Argument in einer Menge von akzeptierten Argumenten enthalten ist. Dies wird mithilfe von Semantiken ermittelt, die die Vorschriften für die Akzeptanz der Argumente definieren.

Semantiken

In diesem Abschnitt werden die *vollständigen*, die *präferierten* und die *stabilen* Semantiken vorgestellt. Weitere Semantiken sind in [5] zu finden.

Eine Semantik definiert, ob ein Argument eines AFs zu einer akzeptierten Menge gehört. Die Definition der Semantiken kann mithilfe zweier Ansätze erfolgen:

- Label-basierter Ansatz
- Extension-basierter Ansatz

Bei Ersterem wird jedes Argument auf ein Label abgebildet. Die Labelmenge kann bspw. die Elemente {*in, out, undefined*} enthalten, die in diesem Fall die Bedeutung *akzeptiert*, *abgelehnt* und *undefiniert* haben. Der zweite Ansatz berechnet Mengen von akzeptierten Argumenten. Eine Menge von *akzeptierten* Argumenten heißt Extension und die darin enthaltenen Argumente bekämen im Rahmen des labelbasierten Ansatzes das *in* Label zugewiesen. Folglich ist es möglich, die Ergebnisse beider Ansätze in das Format ihres Gegenparts zu transformieren. Im weiteren Verlauf werden nur Extension-basierte Semantiken, die durch eine Funktion σ dargestellt werden, betrachtet. Für eine Funktion σ gilt mit einem gegebenen AF F und dessen Knotenmenge Ar: $\sigma(F) \subseteq 2^{Ar}$.

Eine Knotenmenge $Args^+$ repräsentiert die Knotenmenge, die von einer Knotenmenge $Args$ angegriffen wird und $Args^-$ repräsentiert die Kontenmenge, die $Args$ angreift. Analog wird diese Notation für einen einzelnen Knoten benutzt.

Definition 2.2 (Angreifer, Verteidiger)**.** Gegeben sei ein *Abstract Argumentation Framework* $AF = \langle Ar, att \rangle$. Sei $Args \subseteq Ar$ und $a \in Ar$

$$\begin{aligned} Args^+ &= \{b \in Ar \mid \exists c \in Args : (c, b) \in att\} \\ Args^- &= \{b \in Ar \mid \exists c \in Args : (b, c) \in att\} \\ a^+ &= \{b \in Ar \mid (a, b) \in att\} \\ a^- &= \{b \in Ar \mid (b, a) \in att\} \end{aligned}$$

Die Angriffsreichweite einer Kontenmenge $Args$ fasst diese Knotenmenge, sowie alle Knoten, die von dieser Menge angegriffen werden, zusammen.

Definition 2.3 (Angriffsreichweite)**.** Gegeben sei ein *Abstract Argumentation Framework* $\langle Ar, att \rangle$ und sei $Args \subseteq Ar$. Für die *Angriffsreichweite* $Args^{\oplus}$ von $Args$ in AF gilt:

$$Args^{\oplus} = Args \cup Args^{+}$$

Die *Charakteristik*-Funktion eines AFs liefert für eine Teilmenge $Args$ die Menge an Argumenten zurück, die von $Args$ verteidigt werden.

Definition 2.4 (Verteidigung)**.** Gegeben sei ein abstraktes Argumentation Framework $\langle Ar, att \rangle$. Eine Menge $Args \subseteq Ar$ *verteidigt* ein Argument $a \in Ar$, falls

$$a^{-} \quad \subseteq Args^{+}$$

gilt.
Für die *Charakteristik-Funktion* F_{AF} von AF gilt

$$F_{AF} \ : \ 2^{Ar} \rightarrow 2^{Ar} \text{ mit } F_{AF}(Args) = \{a \mid Args \textit{ verteidigt } a\}$$

Falls die Argumente der Teilmenge $Args$ sich gegenseitig nicht angreifen, wird diese Teilmenge auch als *konfliktfrei* bezeichnet.

Definition 2.5 (Konfliktfrei)**.** Gegeben sei ein *Abstract Argumentation Framework* $\langle Ar, att \rangle$. Eine Teilmenge $Args \subseteq Ar$ heißt konfliktfrei, falls $\neg \exists a, b \in Args : (a, b) \in att$ gilt.

Eine konfliktfreie Menge, die durch die *Charakteristik-Funktion* berechnet wurde, ist eine *zulässige* Menge.

Definition 2.6 (Zulässige Menge)**.** Gegeben sei ein *Abstract Argumentation Framework* $\langle Ar, att \rangle$. Eine Teilmenge $Args \subseteq Ar$ heißt *zulässig*, falls diese konfliktfrei ist und $Args \subseteq F_{AF}(Args)$ gilt.

Eine *vollständige* Extension ist eine *zulässige* Menge mit der Einschränkung, dass jedes Argument, das durch diese Menge verteidigt wird, ebenfalls in dieser enthalten ist. Die Extension ist somit ein Fixpunkt der *Charakteristik-Funktion*.

Definition 2.7 (Vollständige Semantik)**.** Gegeben sei ein *Abstract Argumentation Framework* $\langle Ar, att \rangle$. Eine Teilmenge $Args \subseteq Ar$ ist eine Extension einer *vollständigen* Semantik, falls $Args$ *konfliktfrei*, sowie ein Fixpunkt der *Charakteristik-Funktion* $Args = F_{AF}(Args)$ ist. Die *vollständige* Semantik wird mit σ_{com} abgekürzt.

Eine Extension der *präferierten* Semantik hat die Eigenschaft, dass diese eine größtmögliche *zulässige* Menge bildet.

Definition 2.8 (Präferierte Semantik)**.** Gegeben sei ein *Abstract Argumentation Framework* $\langle Ar, att \rangle$. Sei E eine Extension der *präferierten* Semantik und die Menge $com(AF)$ enthält alle *vollständigen* Extensionen. Dann gilt:

$$E = \{A \mid A \in com(AF) \text{ und } \nexists B \supset A \text{ mit } B \in com(AF)\}$$

Die *präferierte* Semantik wird mit σ_{pref} abgekürzt.

Die *stabile* Semantik nimmt eine Einschränkung bzgl. der Akzeptanz vor. Ein Argument darf entweder akzeptiert oder zurückgewiesen werden. Diese Beschränkung modelliert ein Schwarz-Weiß-Denken.

Definition 2.9 (Stabile Semantik)**.** Gegeben sei ein *Abstract Argumentation Framework* $\langle Ar, att \rangle$. Eine *stabile* Extension ist eine *vollständige* Extension mit der Einschränkung, dass für die Menge $Ar \setminus (Args \cup Args^{+}) = \emptyset$ gilt. Die *stabile* Semantik wird mit σ_{stb} abgekürzt.

Zusammenfassend haben die genannten Definitionen folgende Abhängigkeiten:

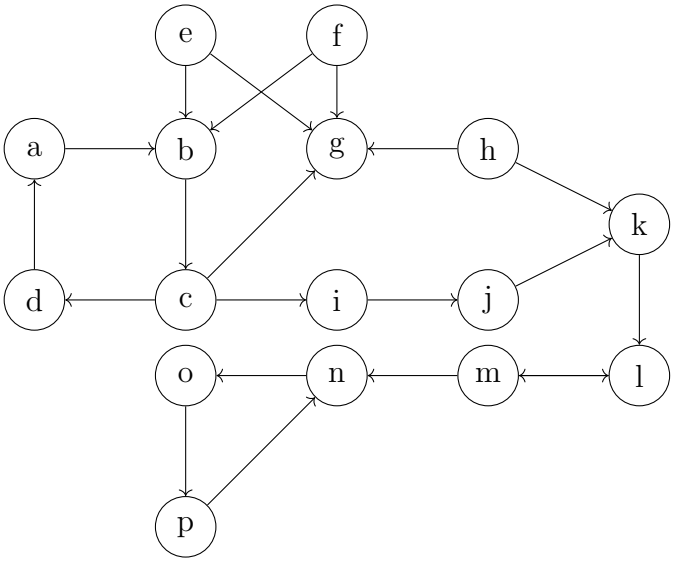
- Jede *zulässige* Extension ist eine *konfliktfreie* Extension
- Jede *vollständige* Extension ist eine *zulässige* Extension
- Jede *präferierte* Extension ist eine *vollständige* Extension
- Jede *stabile* Extension ist eine *präferierte* Extension

Der AF F in Beispiel 1 veranschaulicht die eben genannten Abhängigkeiten. Wie aus der zugehörigen Tabelle ersichtlich, gelten für die Extensionen der Semantiken folgende Teilmengenbeziehungen: $\sigma_{stb}(F) \subseteq \sigma_{pref}(F) \subseteq \sigma_{com}(F)$. Bei einer genaueren Betrachtung der Extensionen lässt sich eine Menge von Argumenten, mit $\{a, c, e, f, h, j\}$, identifizieren, die in jeder Extension enthalten sind. Für solche Argumente wäre es potenziell möglich, diese zusammenzufügen. Gleichzeitig sind manche Argumente in keiner Extension enthalten. Ein Zusammenfügen oder ein Entfernen von Argumenten würde einen AF vereinfachen. Wann und wie Argumente zusammengefügt oder entfernt werden dürfen, führt zu der Thematik der Ersetzungsmuster (Replacement Patterns), die im nächsten Kapitel vorgestellt werden.

3 Vereinfachung eines AFs

In der formalen Argumentation ist es von Interesse, die Akzeptanz eines Arguments zu ermitteln, um Schlussfolgerungen ziehen zu können. Hinsichtlich der Akzeptanz ist zu unterscheiden, ob ein Argument in mindestens einer

Beispiel 1 Ein AF mit den Extensionen der unterschiedlichen Semantiken



Extensionen		
σ_{com}	σ_{pref}	σ_{stb}
$\{a, c, e, f, h, j, m, o\}$ $\{a, c, e, f, h, j, l\}$ $\{a, c, e, f, h, j\}$	$\{a, c, e, f, h, j, m, o\}$ $\{a, c, e, f, h, j, l\}$	$\{a, c, e, f, h, j, m, o\}$

Extension (*credulous accepted*) oder in allen Extensionen (*sceptical accepted*) enthalten ist. Das Entscheidungsproblem, das den leichtgläubigen (*credulous*) Fall behandelt, ist für die *vollständigen*, *präferierten* und *stabilen* Semantiken NP-vollständig. Ein Algorithmus muss überprüfen, ob ein Argument in einer Extension enthalten ist und im Worst-Case-Szenario muss dieser alle Extensionen überprüfen. Für den *sceptical accepted* Fall ist das Entscheidungsproblem für die *vollständige* und *stabile* Semantik ebenfalls NP-vollständig und für die *präferierte* Semantik liegt das Entscheidungsproblem in der polynomiellen Hierarchie sogar eine Stufe höher. Eine detaillierte Untersuchung der Komplexitätsklassen für weitere Berechnungsprobleme in der Formalen Argumentation ist in [23] zu finden. Auf Grund der Komplexität ist es von Interesse solche Berechnungen effizient zu lösen. Um diesem Interesse zu begegnen, ist der hier verfolgte Ansatz die Vereinfachung eines AFs.

Die Idee ist es, mittels Ersetzungsmustern einen AF zu vereinfachen, um nachfolgende Berechnungen zu beschleunigen. Die Ersetzungsmuster, die in diesem Kapitel vorgestellt werden, wurden ursprünglich in der Arbeit [24] von Dvorák et al. präsentiert.

Wie im vorherigen Kapitel beschrieben, kann ein AF durch einen gerichteten

Graphen repräsentiert werden. Durch die Anwendung eines Ersetzungsmusters kommt es zu einer strukturellen Veränderung des Graphens, weshalb es zwingend notwendig ist, dass der transformierte Graph semantisch äquivalent zu seinem Ursprungsgraphen ist.

3.1 Semantische Äquivalenz

Für die semantische Äquivalenz und die Ersetzungsmuster werden die nachfolgend behandelten Mengenoperationen benötigt.

Definition 3.1 (Mengenoperationen)**.** Gegeben seien zwei AFs F = $\langle Ar, att\rangle$ und F' = $\langle Ar', att'\rangle$. Sei S mit $S \subseteq Ar$ eine Menge von Argumenten und $T \subseteq (Ar \times Ar)$ eine Menge von Angriffen.
Für die Vereinigung zweier AFs gilt:

$$F \cup F' = \langle Ar \cup Ar', att \cup att'\rangle$$

Für den Schnitt eines AFs mit einer Argumentenmenge gilt:

$$F \cap S = \langle Ar \cap S, att \cap ((att \cap S) \times (att \cap S))\rangle$$

Für die Differenz eines AFs mit einer Menge von Angriffen gilt:

$$F \setminus T = \langle Ar, att \setminus T\rangle$$

Die *normale* semantische Äquivalenz ist gegeben, wenn zwei AFs für eine gegebene Semantik σ eine identische Extensionsmenge besitzen. Dieses Konzept garantiert keine zuverlässigen Ersetzungen von AFs, die in größeren AFs eingebettet sind. Dabei ist eine zuverlässige Ersetzung gegeben, falls die semantische Äquivalenz erhalten bleibt. Folgendes Beispiel soll die Problematik illustrieren. Angenommen zwei AFs $F = \langle\{x_1, x_2, x_3\}, \{(x_1, x_2), (x_2, x_3), (x_3, x_1)\}\rangle$ und $F' = \langle\{x_1, x_2\}, \{(x_1, x_1), (x_1, x_2)\}\rangle$ sind in dem AF G durch den Knoten x_4 eingebettet (vgl. Abbildung 1), dann ist die Ersetzung F durch F' für die *zulässige* Semantik valide. Wenn hingegen das Argument x_4 nicht x_1, sondern x_2 angreift, ist es nicht erlaubt, F durch F' zu ersetzen. Die Einschränkungen hinsichtlich der erlaubten Angriffe werden in der *normalen* semantischen Äquivalenz nicht festgehalten, sodass eine genauere Beschreibung notwendig ist. Die σ-Äquivalenz aus [9] präzisiert die semantische Äquivalenz zweier AFs. Im Rahmen dessen wurde der Begriff der *Kernargumente* (*core arguments*) und die dazu gehörige *C-beschränkte*-Semantik eingeführt. Eine *C-beschränkte*-Semantik grenzt die Merkmale einer Originalsemantik auf die *Kernargumente* sowie auf die Interaktion der *Kernargumente* mit Argumenten des restlichen Graphen ein. Zusätzlich gilt für die *Kernargumente* die Bedingung, dass die Beziehungen untereinander fixiert sind. Angenommen zwei AFs F, F' enthalten eine Kernargumentenmenge C, die wiederum nicht in einem dritten AF H enthalten ist, dann sind F und F' semantisch äquivalent, wenn die beiden vereinigt mit irgendeinem AF H identische Mengen an Extensionen haben.

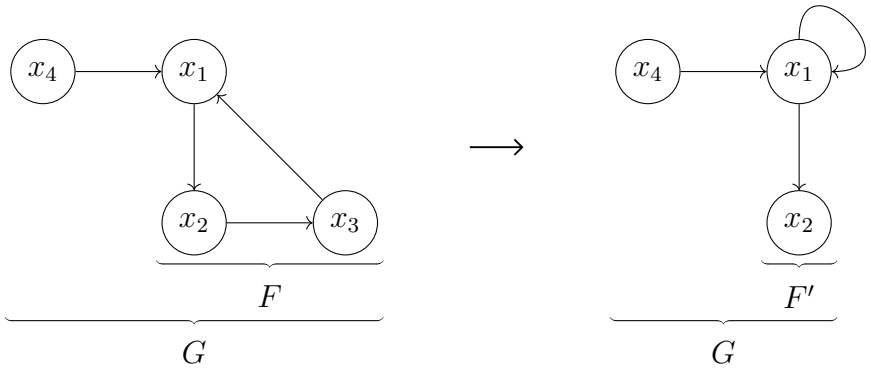


Abbildung 1: F, F' eingebettet in G

Definition 3.2 (Semantische Äquivalenz)**.** Gegeben seien zwei AFs F, F' und eine Semantik σ. F und F' enthalten eine Argumentemenge C, die auch *Kernargumente* genannt wird. Diese Menge C ist nicht in irgendeinem AF H enthalten. Dann sind F und F' semantisch äquivalent, wenn

$$\sigma(F \cup H) = \sigma(F' \cup H)$$

für jeden AF H gilt.

Im Kontext der Ersetzungsmuster wären die beiden AFs F und F' aus der Definition 3.2 das Ursprungsmuster mit dem jeweilig zugehörigen Ersetzungsmuster. Ein Muster kann angewendet werden, falls gegen gewisse Einschränkungen nicht verstoßen wird. Dadurch ist es sogar möglich, verschiedene Muster verkettet auszuführen ohne gegen die semantische Äquivalenz zu verstoßen. Orientiert an der Semantik fallen diese Einschränkungen unterschiedlich aus. Die im folgenden Abschnitt behandelten Muster besitzen alle die Eigenschaft, dass der Graph nach Anwendung eines Musters immer weniger Knoten- bzw. Kanten besitzt, als vor der Ausführung.

3.2 Ersetzungsmuster

Um einen AF zu vereinfachen, ist auf sogenannte Ersetzungsmuster zurückzugreifen, die Untersuchungsgegenstand dieses Abschnittes sind. Zur Veranschaulichung existiert zu jeder Definition eines Ersetzungsmusters eine Abbildung. Bei diesen ist zu beachten, dass durchgezogene Kanten notwendige und gestrichelte Kanten optionale Kanten repräsentieren. Ein nicht beschrifteter Knoten stellt irgendeinen anderen Knoten des Graphen dar (vgl. bspw. Abbildung 2).

Für die Ersetzungsmuster wird das Konzept eines *isolierenden* AFs benötigt.

Definition 3.3 (Isolierung)**.** Gegeben seien zwei AFs $F = \langle Ar, att \rangle$ und $F_I = \langle Ar', att' \rangle$ mit $Ar \subseteq Ar'$. Ein AF F_I *isoliert* F, wenn gilt:

$$F_I \cap Ar = F \text{ und } Ar^{\oplus}_{F_I} \cup Ar^{-}_{F_I} = Ar'$$

Ein AF F_I *isoliert* einen anderen AF F, indem alle Argumente Ar von F auch in F_I enthalten sind. Die Argumentenmenge Ar' bildet sich aus den Argumenten Ar, sowie allen Argumenten, die Ar angreifen oder von Ar angegriffen werden. Ein Argument a, mit $a \in Ar' \setminus Ar$, wird auch allgemein als *isolierendes* Argument bezeichnet und zusätzlich als *äußerer* Angreifer, wenn dieses ein Argument aus Ar angreift. Zudem wird der AF F auch als *Kern-AF* bezeichnet und bildet ein Schema für ein Muster, indem eine Grundstruktur spezifiziert wird.

Wie die Abbildung 1 aus dem vorherigen Kapitel illustriert hat, muss eine Einschränkung hinsichtlich zulässiger Angriffe auf den *Kern-AF* spezifiziert werden, bevor eine Transformation auf einen AF angewendet werden darf. Ansonsten ist die semantische Äquivalenz nicht gewährleistet. Wenn ein AF F_I einen *Kern-AF* F *isoliert* und die geforderten Einschränkungen hinsichtlich der Angriffe eingehalten werden, dann kann eine Transformation vollzogen werden.

Bei der Transformation von einigen der hier definierten Muster werden zwei Knoten zu einem Knoten zusammengefügt. Bei dem Zusammenfügen zweier Argumente a und b entsteht ein neues Argument $m_{a,b}$, das alle Angriffe als auch Attacken von Argument a und Argument b erbt.

Definition 3.4 (Zusammenfügen)**.** Gegeben sei ein AF $F = \langle Ar, att \rangle$. Zwei Argumente $a, b \in Ar$ werden durch $M(F, a, b)$ zu einem Knoten $m_{a,b}$ zusammengefasst (merging):

$$\begin{array}{lcl} M(F,a,b) & = & F' \\ F' & = & \langle Ar', att' \rangle \\ Ar' & = & \{Ar \setminus \{a,b\} \cup m_{a,b}\} \\ att' & = & att \cap (Ar' \times Ar') \cup \{(m_{a,b}, c) \mid (a,c) \in att \text{ oder } (b,c) \in att\} \\ & & \cup \{(c, m_{a,b}) \mid (c,a) \in att \text{ oder } (c,b) \in att\}\} \end{array}$$

Bei den vier Grundmustern aus [24] handelt es sich um das *3-Pfad-* (vgl. Abbildung 2), das *3-Kreis-* (vgl. Abbildung 3), das *3-Kegel-* (vgl. Abbildung 4) und das *2-zu-1*-Muster (vgl. Abbildung 5). Diese Muster decken typische Graphenstrukturen ab, um einen Graphen unter Berücksichtigung der semantischen Äquivalenz in einen kleineren Graphen überführen zu können. Darüber hinaus können die Muster *3-Pfad*, *3-Kegel* und *2-zu-1* zu größeren Mustern, wie zum Beispiel das *3-Pfad*-Muster zu einem *5-Pfad*-Muster, verallgemeinert werden. Alle Muster erfüllen die geforderte semantische Äquivalenz gegenüber der *stabilen* Semantik, da diese am restriktivsten bzgl. eines akzeptierten Arguments unter den drei betrachteten Semantiken ist. Dadurch existieren weniger potenzielle Akzeptanzmöglichkeiten und somit weniger Einschränkungen,

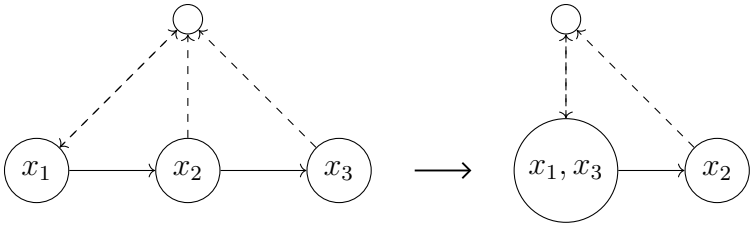


Abbildung 2: 3-Pfad-Muster

die ein Muster berücksichtigen muss. Die Anwendungsmöglichkeiten für die *vollständige* und *präferierte* Semantik unterliegen bei dem *3-Kreis*- und dem *3-Kegel*-Muster speziellen Ausnahmen, die im Folgenden erläutert werden.

Das *3-Pfad*-Muster in Abbildung 2 ist für jede der hier behandelten Semantiken einsetzbar. Wenn ein AF F_{3P} im AF F enthalten ist (vgl. Definition 3.5), bzw. F *isoliert* F_{3P} und die Knoten x_2 und x_3 keine *äußeren* Angreifer besitzen ($\{x_2, x_3\}_F^- = \{x_1, x_2\}$), dann ist Muster P^{3P} anwendbar. Dabei wird F in F' durch die Zusammenfügungsoperation aus Definition 3.4 überführt, indem der Knoten x_3 mit all seinen Kanten mit dem Knoten x_1 zusammenfügt wird.

Definition 3.5 (3-Pfad-Muster)**.** Gegeben sei ein AF $F_{3P} = (\{x_1, x_2, x_3\}, \{(x_1, x_2), (x_2, x_3)\})$. Für das Ersetzungmuster *3-Pfad* gilt:

$$P^{3P}_{x_1,x_2,x_3} = \{(F, F') \mid F \textit{ isoliert } F_{3P}, \{x_2, x_3\}_F^- = \{x_1, x_2\}, F' = M(F \setminus \{(x_2, x_3)\}, x_1, x_3)\}$$

Ähnlich wie beim *3-Pfad*-Muster besteht das *3-Kreis*-Muster aus drei Argumenten, wobei das letzte das erste Argument angreift. Analog zu dem vorherigen Muster gilt hier die Einschränkung, dass nur das Argument x_1 von außen angeriffen werden darf. Für diesen Knoten muss bei allen betrachteten Semantiken eine Kante auf sich selbst hinzugefügt werden. Bei der *stabilen* Semantik wird der lokal betrachtete Subgraph in zwei Komponenten zerlegt, da der Knoten x_3 entfernt wird. Es ist deshalb erlaubt, da die Knoten x_1 und x_3 nie in einer Extension enthalten sein können. Dies gilt auch für die *präferierte* sowie die *vollständige* Semantik. Deswegen ist die Entfernung von x_3 mit der Einschränkung erlaubt, dass x_1 alle ausgehenden Kanten von x_3 erbt. Zusätzlich darf bei den vorgenannten Semantiken die Kante (x_1, x_2) nicht entfernt werden.

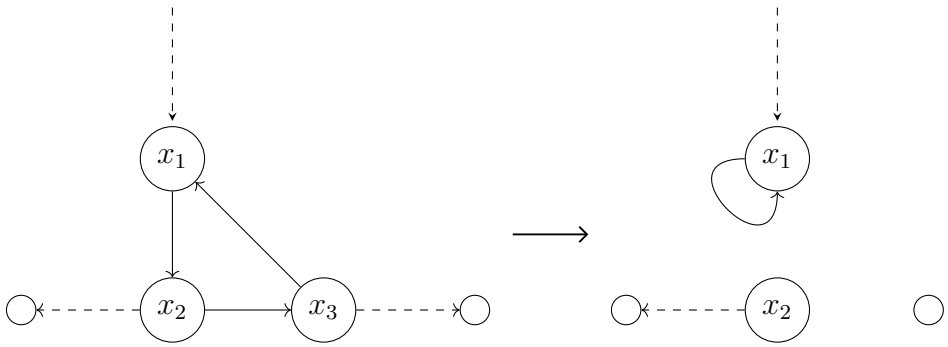


Abbildung 3: *3-Kreis*-Muster, *stabile* Semantik

Definition 3.6 (3-Kreis-Muster)**.** Gegeben sei ein AF $F_{3Kr} = (\{x_1, x_2, x_3\}, \{(x_1, x_2), (x_2, x_3), (x_3, x_1)\})$. Für das Ersetzungmuster *3-Kreis* im Kontext der *stabilen* Semantik gilt:

$$P^{3Kr}_{x_1,x_2,x_3} = \{(F, F') \mid F \setminus \{(x_1, x_1), (x_3, x_3)\} \textit{ isoliert } F_{3Kr}, \{x_2, x_3\}^-_F \subseteq \{x_1, x_2, x_3\}, F' = (F \setminus \{x_3, (x_2, x_3)\}) \cup \{(x_1, x_1)\}$$

Das *3-Kegel*-Muster kann nur für die *stabile* Semantik ohne Einschränkungen verwendet werden. Dabei wird der Knoten x_3 entfernt und es entsteht ein nicht zusammenhängender Graph. Dies ist bei der *stabilen* Semantik erlaubt, da entweder x_1 oder x_2 in einer Extension enthalten sein muss. Bei der *präferierten* Semantik muss zusätzlich der Knoten x_1 oder x_2 sich gegen alle Angreifer verteidigen. Die *vollständige* Semantik ist für dieses Muster nicht anwendbar.

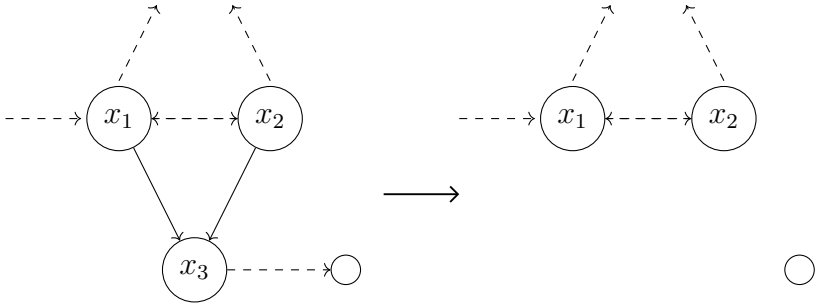


Abbildung 4: *3-Kegel*-Muster - *stabile* Semantik

Definition 3.7 (3-Kegel-Muster)**.** Gegeben sei ein AF $F_{3Ke} = (\{x_1, x_2, x_3\}, \{(x_1, x_3), (x_2, x_3)\})$. Für das Ersetzungmuster *3-Kegel* im Kontext der *stabilen* Semantik gilt:

$$P^{3Kr}_{x_1,x_2,x_3} = \{(F, F') \mid F \setminus \{(x_1, x_2), (x_2, x_1)\} \textit{ isoliert } F_{3Ke}, \{x_2\}^-_F \subseteq \{x_1\}, F' = F \setminus \{x_3\}$$

Das *2-zu-1*-Muster kann für alle hier vorgestellten Semantiken eingesetzt werden. Es ist anwendbar, wenn zwei Knoten die gleichen oder keinen Angreifer besitzen und wenn diese sich nicht angreifen. Bei dem Transformationsschritt werden beide Argumente zu einem Argument zusammengefügt.

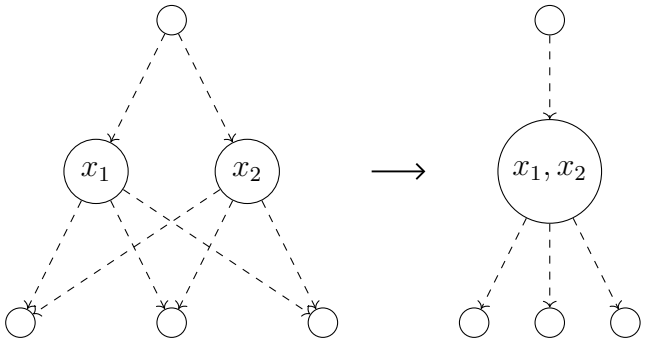


Abbildung 5: *2-zu-1*-Muster

Definition 3.8 (2-zu-1-Muster)**.** Gegeben sei ein AF $F_{2zu1} = (\{x_1, x_2\}, \emptyset)$. Für das Ersetzungmuster *2-zu-1* gilt:

$$\begin{aligned} P^{2zu1}_{x_1,x_2} = \quad & \{(F, F') \mid F \textit{ isoliert } F_{2zu1}, \\ & \{x_1\}^-_F = \{x_2\}^-_F \ , \ F' = M(F, x_1, x_2)\} \end{aligned}$$

Bei der verallgemeinerten Version, dem *3-zu-2*-Muster, greifen sich zwei der drei Argumente gegenseitig an. Das dritte Argument kann in diesen Fall zu beiden Argumenten hinzugefügt werden.

Definition 3.9 (3-zu-2-Muster)**.** Gegeben sei ein AF $F_{3zu2} = (\{x_1, x_2, x_3\}, \emptyset)$ und $AF\ F = \langle Ar, att \rangle$. Für das Ersetzungmuster *3-zu-2* gilt:

$$\begin{aligned} P^{3zu2}_{x_1,x_2,x_3} &= \{(F, F') \mid F \setminus \{(x_1, x_2), (x_2, x_1)\} \textit{ isoliert } F_{3zu2}, \\ &\quad \{x_1\}^-_F = \{x_2\}^-_F = \{x_3\}^-_F\} \\ F' &= \langle Ar', att' \rangle \\ Ar' &= Ar \setminus \{x_1, x_2, x_3\} \cup \{m_{x_1,x_3}, m_{x_2,x_3}\} \\ att' &= att \cap \{Ar' \times Ar'\} \cup \{(m_{x,x_3}, x_3) \mid x \in \{x_1, x_2\} \\ &\quad \wedge ((x_1, x_3) \in att \vee (x_2, x_3) \in att)\} \cup \\ &\quad \{(x_4, m_{x_1,x_3}), (x_4, m_{x_2,x_3}) \mid x_4 \in \{x_3\}^-\}\} \end{aligned}$$

Neben den fünf genannten Mustern wurden in der wissenschaftlichen Arbeit in [24] zwei weitere triviale Muster zwar nicht definiert, jedoch in der Mustersuche des Algorithmus verwendet. Es handelt sich hierbei um ein spezielleres *2-zu-1*-Muster (vgl. Definition 3.10). Als Einschränkung gelten für die Argumente x_1 und x_2, dass diese keine Angreifer besitzen.

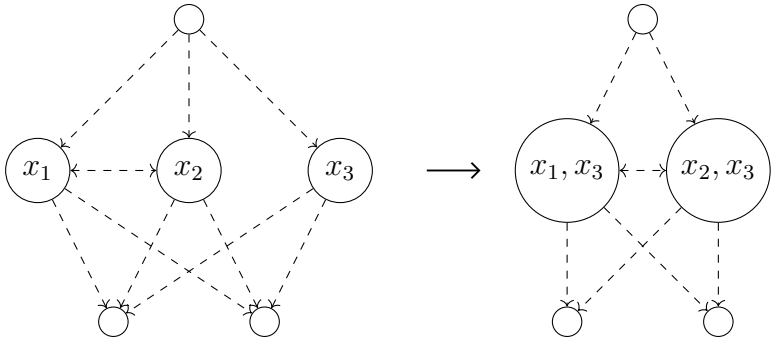


Abbildung 6: *3-zu-2*-Muster

Definition 3.10 (Unattackiertes 2-zu-1-Muster)**.** Gegeben sei ein AF $F_{2zu1} = (\{x_1, x_2\}, \emptyset)$. Für das Ersetzungmuster *unattackiertes 2-zu-1* (unatt. 2-zu-1) gilt:

$$P^{2zu1}_{x_1,x_2} = \{(F, F') \mid F \textit{ isoliert } F_{2zu1}, \{x_1\}^-_F = \{x_2\}^-_F = \emptyset \ , \ F' = M(F, x_1, x_2)\}$$

Bei dem zweiten Muster handelt es sich um das *Grounded*-Muster (vgl. Definition 3.11) in Anlehnung an die gleichnamige Semantik. Bei der *fundierten* (grounded) Semantik wird ein Argument akzeptiert, falls es in einer Menge an Argumenten enthalten ist, die einen kleinsten Fixpunkt von F_{AF} bildet. Ein Argument, das nicht angegriffen wird, bildet ein gewisses Fundament bzgl. der Akzeptanz und muss immer in einer solchen Menge enthalten sein.

Definition 3.11 (Grounded-Muster)**.** Gegeben sei ein AF $F_{1zu1} = (\{x_1, x_2\}, \{(x_1, x_2)\})$. Für das Ersetzungmuster *Grounded* gilt:

$$P^{1zu1}_{x_1,x_2} = \{(F, F') \mid F \textit{ isoliert } F_{1zu1}, \{x_1\}^-_F = \emptyset \ , \ F' = F \setminus x_2\}$$

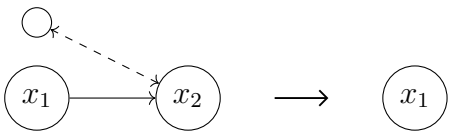
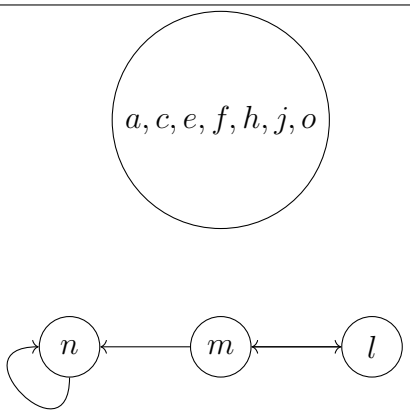


Abbildung 7: *Grounded*-Muster

Mit den vorgestellten Mustern soll nun der AF aus Beispiel 1 transformiert werden. Durch die Transitivität der Äquivalenzrelation (vgl. [24]), können die Muster rekursiv angewendet werden. Das zugehörige Endergebnis findet sich in Beispiel 2. Als erstes wird das *2-zu-1*-Muster für die Argumente e und f an-

Beispiel 2 Transformierter AF erstellt aus dem AF aus Beispiel 1 für die stabile Semantik



gewendet. Anschließend kann g durch das *Grounded*-Muster entfernt werden. Hier bieten sich zwei *Grounded*-Muster mit den nicht attackierten Argumenten *ef* und h an. Analog kann b entfernt werden, da es durch *ef* angegriffen wird. Die Argumente a und c werden durch das *3-Pfad*-Muster *c,d,a* zu *ac* zusammengefügt. Da *ac* nicht mehr durch b angegriffen wird, besitzt es keinen Angreifer und somit ist es erlaubt, d mittels des *Grounded*-Musters zu entfernen. Das *3-Pfad*-Muster *ac,i,j* führt zu *acj*. Im Anschluss kann i durch das *Grounded*-Muster *acj,i* entfernt werden. Das Argument k wird durch das *3-Kegel*-Muster *h,acj,k* entfernt. Die Argumente *ef*, h und *acj* haben keine Angreifer und durch zweimalige Ausführung eines *2-zu-1*-Musters entsteht das Argument *acefhj*. Es kann noch das *3-Kreis*-Muster *n,o,p* angewendet werden. Dabei wird p entfernt und o verliert seinen einzigen Angreifer. Schließlich wird o mit *acefhj* zusammengefügt und es entsteht der AF, wie im Beispiel 2 dargestellt.

Wie bei der vorherigen Anwendung der Transformationsschritte für den AF aus Beispiel 1 ersichtlich, hätten die Muster auch in einer anderen Reihenfolge ausgeführt werden können. Eine verkettete Anwendung von Transformationsschritten für einen gegebenen AF, bis kein Muster mehr anwendbar ist, wird auch *Transformationspfad* genannt. Für den AF aus dem Beispiel 1 wurden alle *Transformationspfade* ermittelt. Dabei wurden initial alle *2-zu-1*-Muster angewendet, dabei wurden über 26.000.000 unterschiedliche *Transformationspfade* ermittelt. Auffällig war, dass alle *Transformationspfade* gleich lang waren und immer in dem gleichen transformierten AF aus Beispiel 2 resultierten. Vor dem Hintergrund dieser Beobachtung wird im nächsten Kapitel die Reihenfolge der Transformationsschritte genauer untersucht.

4 Reihenfolge der Transformationsschritte

In diesem Abschnitt wird die Anwendung der Transformationsschritte im Hinblick auf deren Reihenfolge näher untersucht. Es stehen die beiden folgenden Fragestellungen im Mittelpunkt der Untersuchung:

- Wie wirkt sich eine unterschiedliche Ausführung der Transformationsschritte auf einen AF aus?
- Existieren unterschiedlich lange *Transformationspfade*?

Wie das Beispiel 3 zeigt, ist eine unterschiedliche Anwendungsreihenfolge für den gegebenen Graphen möglich. Wird zuerst das *3-Kegel*-Muster angewendet (linker Pfad), kann im Anschluss das *3-Pfad*-Muster nicht mehr ausgeführt werden, wie dies im rechten Pfad geschehen ist. Das Endresultat des Graphen ist aber trotz der Eliminierung des *3-Pfad*-Musters im linken Pfad identisch im Vergleich zu der Ausführungsreihenfolge des rechten Pfades. Geschuldet ist dies unter anderem dem *Grounded*-Muster.

Die *3-Kegel*- und *3-Pfad*-Muster haben sich im Beispiel 3 im Argument c überschnitten. Die Überschneidung von Mustern ist zentraler Untersuchungsgegenstand in diesem Kapitel. Dazu muss der Begriff Überschneidung präzisiert werden (vgl. Definition 4.1). Zwei Muster überschneiden sich *schwach*, falls die *Kernargumente* des einen Musters sich nur mit *isolierenden* Argumenten des anderen Musters überschneiden. Wenn hingegen die Schnittmenge zwischen den beiden Kernargumentenmengen nicht die leere Menge ergibt, so handelt es sich um eine *starke* Überschneidung.

Definition 4.1 (Schwache und starke Überschneidung)**.** Gegeben seien zwei Muster $P_{C_1} = (F_1, F_1^{'})$ und $P_{C_2} = (F_2, F_2^{'})$ mit den *Kernargumenten* C_1, C_2 und *isolierenden* AFs F_1, F_2. P_{C_1} und P_{C_2} überschneiden sich *schwach*, gdw.:

$$(F_1 \setminus C_1) \cap C_2 \neq \emptyset$$
$$\text{oder}$$
$$(F_2 \setminus C_2) \cap C_1 \neq \emptyset$$

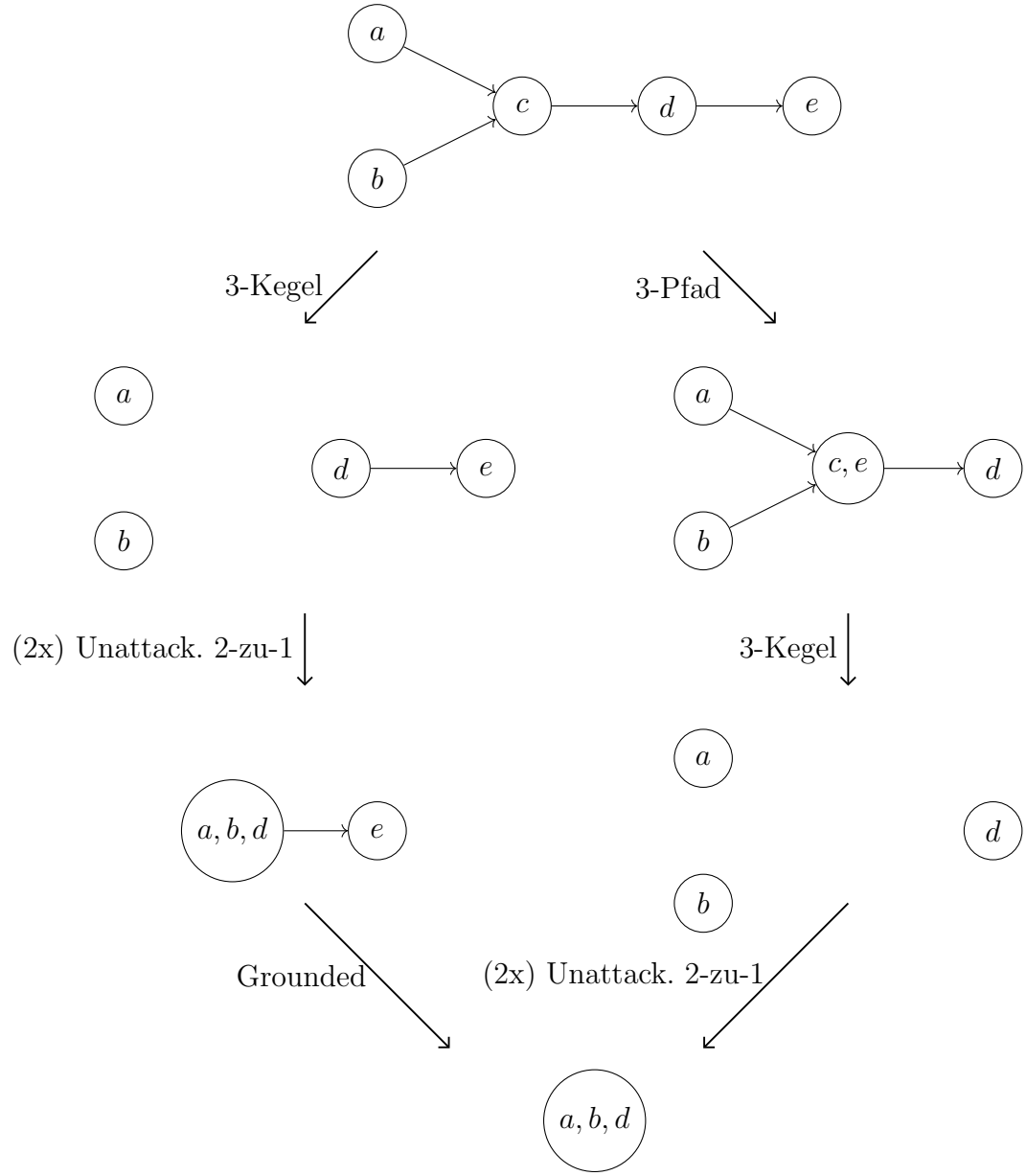
P_{C_1} und P_{C_2} überschneiden sich *stark*, gdw.:

$$C_1 \cap C_2 \neq \emptyset$$

In dem AF aus Beispiel 1 überschneidet sich bspw. das *3-Kegel*-Muster efb mit dem *3-Pfad*-Muster *cda* schwach, da das Argument b Teil des *isolierenden* AFs von dem *3-Pfad*-Muster ist, aber nicht in der zugehörigen Kernargumentenmenge liegt. Hingegen überschneiden sich das *3-Kegel*-Muster efb und das *3-Pfad*-Muster *bcd* stark.

Vorab wird die *schwache* Überschneidung einmal genauer betrachtet. Angenommen es liegt eine *schwache* Überschneidung zwischen zwei Mustern P_1

Beispiel 3 Musteranwendung in unterschiedlicher Reihenfolge



und P_2 mit den Kernargumentenmengen C_1 bzw. C_2 durch ein Argument x mit $x \in C_1$ und $x \notin C_2$ vor. Dann ist das Argument x ein Angreifer oder Angegriffener eines Arguments aus C_2. Somit sind die Angriffe für den *Kern-AF*, der in P_2 liegt, optional. Da die Angriffe optional sind, haben die Transformationsschritte der beiden Muster keinen direkten Einfluss aufeinander. Deshalb muss die *schwache* Überschneidung nicht weiter betrachtet werden. Aus diesem Grund wird im weiteren Verlauf nur die *schwere* Überschneidung untersucht und im restlichen Teil des Kapitels bezieht sich der Begriff Überschneidung auf die *schwere* Überschneidung.

Bevor es zu den Beweisen übergeht, werden einige Formalien, die bei den Beweisführungen genutzt werden, vorangestellt. Bei einigen Fallunterscheidungen exisitiert eine größere Anzahl an Überschneidungen, deshalb werden diese weiter nach Szenarien differenziert. Wenn zwei Argumente a, b zu einem Argument zusammengefügt wurden, wird dies mittels runden Klammern dargestellt: (ab). Die Namenszuordnung der Argumente für die Fallunterscheidungen in diesem Kapitel richten sich nach den Abbildungen 2, 3, 4, 5 und 7 und sind in Tabelle 1 zu finden.

Muster	Argument A	Argument B	Argument C
3-Pfad	$x_1 = a$	$x_2 = b$	$x_3 = c$
3-Kreis	$x_1 = d$	$x_2 = e$	$x_3 = f$
3-Kegel	$x_1 = g$	$x_2 = h$	$x_3 = i$
2-zu-1	$x_1 = j$	$x_2 = k$	
Grounded	$x_1 = q$	$x_2 = r$	

Tabelle 1: Argumentenzuordnung in den Beweisen

Stabile Semantik

Zuerst wird die *stabile* Semantik untersucht. Entsprechend verwenden die betrachteten Muster die Validierungs- und Transformationsoperationen für die *stabile* Semantik. Das ein AF am Ende eines Transformationspfades nicht immer der kleinstmögliche transformierte AF ist, wird in Theorem 1 gezeigt.

Theorem 1 (Verkleinerter AF)**.** Ein AF, der durch die Muster aus Kapitel 2 unter der *stabilen* Semantik in beliebiger Reihenfolge transformiert wurde bis kein weiteres Muster mehr anwendbar ist, kann eine unterschiedliche Anzahl von Argumenten besitzen.

Beweis. Angenommen ein *3-Kreis*-Muster *def* und ein *2-zu-1*-Muster *jk* überschneiden sich stark mit $e = j$. Daraus folgt, dass $\{d\} = \{k\}^-$ gilt. Wird das *2-zu-1*-Muster angewendet, ist die Anwendung des *3-Kreis*-Musters im zweiten Schritt weiterhin möglich. Umgekehrt ist dies nicht der Fall, da bei dem Transformationsschritt des *3-Kreis*-Musters die Kante (d, e) entfernt wird (vgl. Abbildung 8).

Es folgt $\{j\}^- = \emptyset$ und $\{k\}^- = \{d\}$. Die Ausführung des *2-zu-1*-Musters vor dem *3-Kreis*-Muster führt in diesem Fall zu einem kleineren AF. □

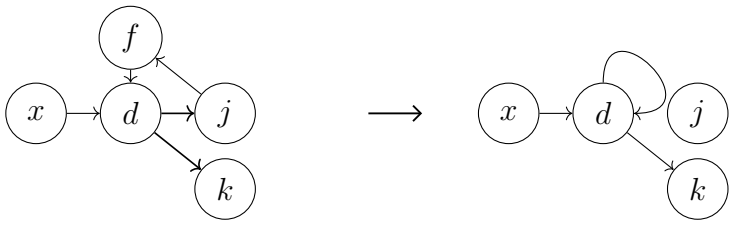


Abbildung 8: Das *2-zu-1*-Muster jk ist nach dem Transformationsschritt des *3-Kreis*-Musters djf nicht mehr anwendbar.

Wie bereits im vorherigen Kapitel beobachtet, besitzt der AF aus Beispiel 1 einen eindeutig verkleinerten AF. Mit folgendem Theorem wird gezeigt, dass es immer einen eindeutigen Graphen gibt, der unter den betrachteten Mustern, mit Ausnahme des *3-Kreis*-Musters, nicht mehr verkleinert werden kann.

Theorem 2 (Eindeutig verkleinerter AF)**.** Gegeben sei ein AF F und dieser wird durch die Muster aus Kapitel 3.2, mit Ausnahme des *3-Kreis*-Musters, zu F' in einer beliebigen Reihenfolge im Kontext der *stabilen* Semantik transformiert bis kein Muster mehr anwendbar ist. Dann ist F' ein eindeutig verkleinerter AF und zudem der kleinstmögliche transformierte AF.

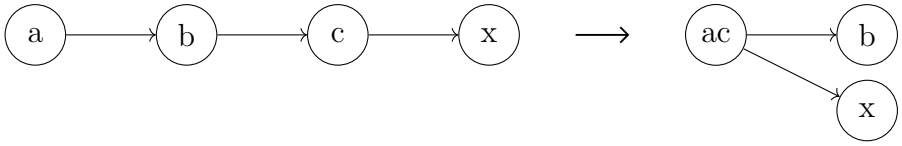
Beweis. Es werden die unterschiedlichen Überschneidungen für die Muster *3-Pfad*, *2-zu-1*, *Grounded* und *3-Kegel* in zehn Fallunterscheidungen (vgl. Tabelle 2) untersucht.

Überschneidung		
Zusammenfügungsop. Zusammenfügungsop.	Eliminierungsop. Zusammenfügungsop.	Eliminierungsop. Eliminierungsop.
F_1 : *3-Pfad* × *3-Pfad*	F_4 : *Grd.* × *3-Pfad*	F_8 : *Grd.* × *Grd.*
F_2 : *3-Pfad* × *2-zu-1*	F_5 : *Grd.* × *2-zu-1*	F_9 : *Grd.* × *3-Kegel*
F_3 : *2-zu-1* × *2-zu-1*	F_6 : *3-Kegel* × *3-Pfad*	F_{10} : *3-Kegel* × *3-Kegel*
	F_7 : *3-Kegel* × *2-zu-1*	

Tabelle 2: Fallunterscheidungen für die unterschiedlichen Muster

Zuerst werden die Fälle überprüft, bei denen sich zwei Zusammenfügungsoperationen überschneiden. Die eben genannte Konstellation kann geschehen, wenn sich zwei *3-Pfad*-Muster oder ein *3-Pfad*- und ein *2-zu-1*-Muster oder zwei *2-zu-1*-Muster überschneiden.

F_1 : Es überschneiden sich zwei *3-Pfad*-Muster, dann bilden die Argumente eine Angriffskette. Falls das *3-Pfad*-Muster, das das Ende der Kette bildet,



(a) Zwei *3-Pfad*-Muster überschneiden sich

(b) Ein *3-Pfad*-Muster und ein *2-zu-1*-Muster überschneiden sich

Abbildung 9: Exemplarische Überschneidung von *3-Pfad*- und *2-zu-1*-Mustern

ausführt wird, dann würde die Kette einfach zusammenschrumpfen. Das vordere *3-Pfad*-Muster wäre weiterhin ausführbar. Wenn hingegen das vordere *3-Pfad*-Muster ausgeführt wird, dann werden die restlichen Argumente durch die Zusammenfügungsoperation umgehangen. Durch das Umhängen entsteht ein *2-zu-1*-Muster. Das neu entstandene *2-zu-1*-Muster ersetzt das *3-Pfad*-Muster, das nicht mehr anwendbar ist. Exemplarisch ist dies in Abbildung 9 (a) illustriert. Dieses Szenario führt zum zweiten Fall, in dem sich ein *2-zu-1*-Muster und ein *3-Pfad*-Muster überschneiden.

F_2 : Ein *2-zu-1*- und ein *3-Pfad*-Muster können sich nur in einem Argument überschneiden. Dazu werden die beiden folgenden Szenarien betrachtet:

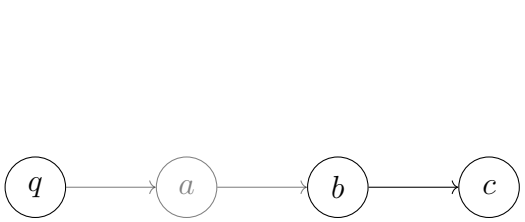
- $S_1 : a = j \vee b = j$
- $S_2 : c = j$

Wird zuerst das *2-zu-1*-Muster ausgeführt, bleibt in beiden Szenarien die Anwendung für ein *3-Pfad* mit $(ak)bc$,$a(bk)c$ oder $ab(ck)$ (je nach Überschneidung) bestehen. Gleiches gilt für das *2-zu-1*-Muster, falls das *3-Pfad*-Muster in S_1 zuerst angewendet wird. Wenn hingegen das *3-Pfad*-Muster in S_2 zuerst ausgeführt wird, dann verschwindet das *2-zu-1*-Muster. Dies ist nicht gravierend, da im Vorfeld ein weiteres *3-Pfad*-Muster bereits existierte, das zum gleichen Ergebnis führt (vgl. Abbildung 9 (b)).

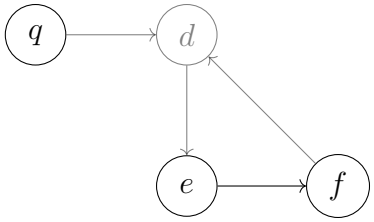
F_3 : Angenommen zwei *2-zu-1*-Muster j_1k und j_2k überschneiden sich im Argument k. Dann werden die drei Argumente von der gleichen Menge an Argumenten angegriffen. Falls das *2-zu-1* Muster j_1k zuerst ausgeführt wird, dann kann im Anschluss das *2-zu-1*-Muster $(j_1k)j_2$ ausgeführt werden, da die Menge der Angreifer sich nicht ändert. Eine umgekehrte Reihenfolge wäre ebenfalls möglich und führt zu dem gleichen Ergebnis.

In den Fällen (4) bis (7) überschneidet sich eine Zusammenfügungsoperation mit einer Eliminierungsoperation.

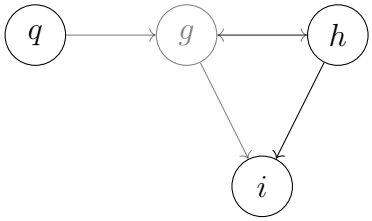
F_4 : Das *Grounded*-Muster qr und *3-Pfad*-Muster abc können sich in den folgenden drei Szenarien überschneiden:



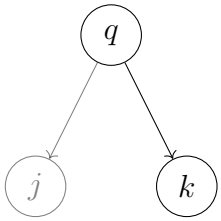
(a) Ehemaliges 3-Pfad-Muster



(b) Ehemaliges 3-Kreis-Muster



(c) Ehemaliges 3-Kegel-Muster



(d) Ehemaliges 2-zu-1-Muster

Abbildung 10: Auswirkung der Zerlegung von Muster

- $S_1 : q = a \wedge r = x$, mit x liegt nicht im Kern-AF vom *3-Pfad*-Muster
- $S_2 : q = a \wedge r = b$
- $S_3 : r = a$

Im ersten Szenario eliminiert das *Grounded*-Muster ein Argument, das außerhalb des Kern-AFs vom *3-Pfad*-Muster liegt. Dementsprechend ist das *3-Pfad*-Muster nicht direkt betroffen und somit können beide Muster in beliebiger Reihenfolge ausgeführt werden. Im zweiten Szenario gilt $q = a$ und $r = b$. Durch Ausführung des *Grounded*-Musters wird das *3-Pfad*-Muster durch ein *2-zu-1*-Muster ersetzt, da die Argumente a und c keinen Angreifer besitzen. Die umgekehrte Ausführungsreihenfolge durch das *3-Pfad*-Muster und dem anschließenden *Grounded*-Muster führt zum gleichen Ergebnis. Im letzten Szenario gilt $r = a$. Angenommen das *3-Pfad*-Muster wird zuerst ausgeführt. Dann entsteht das Argument (ac), das anschließend durch das *Grounded*-Muster $q(ac)$ eliminiert wird. Wenn hingegen zuerst das *Grounded*-Muster angewendet wird, entsteht mit bc ein neues *Grounded*-Muster (vgl. Abb. 10a).

F_5 : Ein *Grounded*-Muster qr und ein *2-zu-1*-Muster jk können sich in einem Argument überschneiden. Dementsprechend können die folgenden beiden Szenarien existieren:

- $S_1 : q = j$
- $S_2 : r = j$

Wenn $q = j$ gilt, dann kann zuerst das *Grounded*-Muster ausgeführt werden, im Anschluss daran das *2-zu-1*-Muster. Die umgekehrte Reihenfolge ist ebenfalls

möglich. Wenn $r = j$ gilt, dann spielt die Reihenfolge ebenfalls keine Rolle. Entweder die Argumente j und k werden durch zwei Transformationen des *Grounded*-Musters entfernt, oder durch eine Zusammenfügungsoperation (*2-zu-1*-Muster) in Verbindung mit einer anschließenden Eliminierungsoperation (*Grounded*-Muster) (vgl. Abbildung 10d). Es macht keinen Unterschied, ob j und k mehrere gemeinsame Angreifer besitzen, da das *Grounded*-Muster j und k entfernen kann.

F_6 : Gegeben sind ein *3-Kegel*-Muster mit *ghi* und ein *3-Pfad*-Muster *abc*. Es sind vier Überschneidungszenarien möglich.

- $S_1 : g = a \wedge h = b$
- $S_2 : g = b \wedge h = c$
- $S_3 : g \in \{a, b, c\}$
- $S_4 : i = a$

Bei den Szenarien (1)-(3) liegen ein bis zwei angreifende Argumente des *3-Kegel*-Musters auf dem *3-Pfad*-Muster. Wenn ein Argument durch den Transformationsschritt des *3-Pfad*-Musters verändert wird, dann ist das *3-Kegel*-Muster weiterhin ausführbar, da die Angriffe umgehängt werden. Der Eliminierungsschritt des *3-Kegel*-Musters betrifft in den Szenarien (1)-(3) das *3-Pfad*-Muster nicht, sondern nur in Szenario (4). In S_4 bleibt aufgrund des Transformationsschrittes des *3-Kegel*-Musters ein *Grounded*-Muster übrig, das den Knoten f eliminiert. Bei der Ausführung des *3-Pfad*-Musters werden a und c zusammengefügt und anschließend durch das *3-Kegel*-Muster entfernt. Das Ergebnis der beiden Ausführungsreihenfolgen ist identisch.

F_7 : Gegeben sind ein *3-Kegel*-Muster mit *ghi* und ein *2-zu-1*-Muster *jk*. Die Muster können sich in den folgenden drei Szenarien überschneiden:

- $S_1 : g = j \wedge h = k$
- $S_2 : g = j \vee h = j$
- $S_3 : i = j$

In S_1 gilt die Einschränkung durch das *2-zu-1*-Muster, dass die Argumente g und h sich nicht angreifen. Durch die Anwendung des *2-zu-1*-Musters entsteht ein *Grounded*-Muster $(gh)i$, das das Argument i im zweiten Schritt entfernt. In umgekehrter Reihenfolge kann das *3-Kegel*-Muster das Argument i zuerst entfernen und das *2-zu-1*-Muster die Argumente g und h anschließend zusammenfügen. In Szenario S_2 überschneidet das *2-zu-1*-Muster nur ein angreifendes Argument des *3-Kegel*-Musters. Dementsprechend entfernt die Eliminierungsoperation des *3-Kegel*-Musters kein Argument, das im *Kern-AF* des *2-zu-1*-Musters liegt, somit ist das Ergebnis beider Ausführungsreihenfolgen identisch. Im letzten Szenario S_3 gilt $i = j$ und somit folgt $\{g, h\} \subseteq \{k\}^-$.

Dementsprechend können i und j zuerst zusammengefügt und dann eliminiert werden oder beide werden in separaten Eliminierungsoperationen entfernt.

Nun werden die Überschneidungen betrachtet, die durch zwei Eliminierungsoperationen auftreten. Deshalb werden nun das *Grounded-* und *3-Kegel*-Muster näher untersucht.

F_8 : Zwei *Grounded*-Muster q_1r_1 bzw. q_2r_2 können sich in einem Argument überschneiden. Die folgenden zwei Szenarien sind möglich:

- $S_1 : q_1 = q_2$
- $S_2 : r_1 = r_2$

In S_1 überschneiden sich die Muster in dem unattackierten Argument und bilden zusammen ein *2-zu-1*-Muster, wie dies exemplarisch in Abbildung 10d dargestellt ist. Dementsprechend verhindert die Ausführung des einen Musters nicht die Anwendbarkeit des Anderen, somit können die Muster in einer willkürlichen Reihenfolge angewendet werden. In S_2 kann das eine Muster nicht nach dem anderen Muster angewendet werden, weil beide das gleiche Argument eliminieren möchten. Folglich kann irgendeins der beiden Muster ausgeführt werden.

F_9 : An dieser Stelle wird nun der Fall untersucht, bei dem sich die *3-Kegel*-Muster P_1 bzw. P_2 mit den Kern-AFs $g_1h_1i_1$ bzw. $g_2h_2i_2$ überschneiden; dies ist in maximal zwei Argumenten möglich. Es existieren drei Szenarien in denen eine Überschneidung möglich ist:

- $S_1 : i_1 = i_2$
- $S_2 : (g_1 = g_2 \vee h_1 = h_2) \wedge i_1 \neq i_2$
- $S_3 : g_1 = i_2$

In S_1 führen beide Muster zum gleichen Resultat. Dementsprechend ist die Reihenfolge irrelevant. Dies ist in S_2 ebenso, da die beiden Muster unterschiedliche Argumente mit $i_1 \neq i_2$ eliminieren möchten. Im dritten Szenario würde durch die Ausführung von P_1 vor P_2 das Muster P_2 zerlegt. Dieses ist ebenfalls nicht negativ einzuordnen, da aus P_2 ein *Grounded*-Muster entsteht, das i_2 entfernt.

F_{10} : Im letzten Fall überschneidet sich ein *Grounded*-Muster qr und ein *3-Kegel*-Muster ghi. Es existieren insgesamt drei unterschiedliche Überschneidungsszenarien:

- $S_1 : i = r \wedge (q = g \vee q = h \vee q = x)$
- $S_2 : (q = g \wedge r = h) \vee (q = h \wedge r = g)$
- $S_3 : q = x \wedge r = g$

In Szenario S_1 ist das Ergebnis beider Muster identisch, da dasselbe Argument entfernt wird. Dabei kann q mit $q = g \vee q = h$ im Kern-AF des *3-Kegel*-Musters enthalten sein oder außerhalb ($q = x$) und einfach das gleiche Argument angreifen. Im Szenario S_2 ist das *3-Kegel*-Muster nicht mehr anwendbar, falls das *Grounded*-Muster zuerst ausgeführt wird. Das *3-Kegel*-Muster muss allerdings ein *Grounded*-Muster *gi* bzw. *hi* enthalten, dass ebenfalls i entfernt. Im letzten Szenario gilt $g = r$. Wird auch hier das *Grounded*-Muster ausgeführt, entsteht aus dem *3-Kegel*-Muster ein *Grounded*-Muster (vgl. Abbildung 10c). Das Ergebnis der unterschiedlichen Ausführungsreihenfolgen ist dementsprechend identisch.

Zusammenfassend lässt sich festhalten, dass, wenn ein Muster durch die Ausführung eines vorherigen Musters nicht mehr anwendbar ist, dann wird dieses durch ein drittes Muster ersetzt. Dies jedoch unter der Bedingung, dass nicht beide Muster das gleiche Argument bei einer Eliminierungsoperation betreffen. Der entstandene AF ist der kleinstmögliche transformierte AF, ansonsten müsste noch ein Muster anwendbar sein. □

Für die theoretische Untersuchung des kürzesten Transformationspfades wird immer ein Muster pro Transformationsschritt angewendet. In der Praxis könnte es sinnvoll sein gewisse Transformationsschritte zusammenzufassen. Beispielsweise könnten bei einem *Grounded*-Muster alle angegriffenen Argumente in einem Zug eliminiert werden.

Lemma 1 (Gleiche Pfadlängen)**.** Gegeben sei ein AF F und dieser wird durch die Muster aus Kapitel 3.2, mit Ausnahme des *3-Kreis* Musters, zu F' in einer beliebigen Reihenfolge transformiert bis kein Muster anwendbar ist. Jeder mögliche Transformationspfad ist gleich lang.

Beweis. In jedem Transformationsschritt werden entweder zwei Knoten zusammengefügt (*3-Pfad-* und *2-zu-1*-Muster) oder es wird ein Knoten eliminiert (*3-Kegel-* und *Grounded*-Muster). Das Ergebnis eines Transformationsschrittes ist beim Zusammenfügen sowie bei der Eliminierung dasselbe: es wird immer nur *ein* Argument aus dem AF entfernt. Wie aus Theorem 2 bekannt, wird ein Muster durch ein Anderes ersetzt, falls es durch einen Transformationsschritt eines anderen Musters nicht mehr anwendbar ist und falls beide nicht das gleiche Argument entfernen. Folglich muss die Länge eines jeden Pfades gleich lang sein. □

Wie das Beispiel im Beweis des Theorems 1 gezeigt hat, ist die Reihenfolge unter Betrachtung aller Muster aus Kapitel 3.2 relevant. Es wird nun die Überschneidung des *3-Kreis*-Musters mit den restlichen Mustern (*3-Kreis-*, *3-Kegel-* und *Grounded*-Muster) überprüft.

Lemma 2 (Starke Überschneidung mit dem *3-Kreis*-Muster)**.** Überschneidet sich ein *3-Kreis*-Muster mit einem anderen *3-Kreis*-Muster, mit einem *3-Kegel-* oder *Grounded*-Muster stark, dann ist die Reihenfolge unter der stabilen Semantik ohne Einfluss auf das Ergebnis.

Beweis. Analog zum Beweis des Theorems 2 wird eine Fallunterscheidung durchgeführt (vgl. Tabelle 3).

Überschneidung	
F_1 : *3-Kreis* $\times$ *3-Kreis*	F_3 : *3-Kreis* $\times$ *Grd.*
F_2 : *3-Kreis* $\times$ *3-Pfad*	F_4 : *3-Kreis* $\times$ *3-Kegel*

Tabelle 3: Fallunterscheidung für das *3-Kreis*-Muster

F_1: Zwei *3-Kreis*-Muster mit den *Kern-AFs* $d_1e_1f_1$ bzw. $d_2e_2f_2$ können sich in den folgenden zwei Szenarien überschneiden:

- $S_1 : d_1 = d_2$ und
- $S_2 : d_1 = d_2 \wedge e_1 = e_2$

Bei S_1 kann eine willkürliche Reihenfolge angewendet werden, da die Ausführung des einen Musters nicht die Anwendbarkeit des Anderen beeinträchtigt. Bei Szenario S_2 wird durch die Anwendung eines *3-Kreis*-Musters die Kante (d_1,e_1) entfernt. Das andere *3-Kreis*-Muster ist dann nicht mehr anwendbar. Durch die Entfernung der Kante (d_1, e_1) entsteht allerdings ein *Grounded*-Muster, das zum gleichen Ergebnis führt.

F_2: Es wird nun die Überschneidung des *3-Kreis*- mit dem *3-Pfad*-Muster betrachtet. Beide Muster können sich in folgenden zwei Szenarien überschneiden:

- $S_1 : a = d$
- $S_2 : a = d \wedge b = e$

In S_1 können die beiden Muster in einer beliebigen Reihenfolge ausgeführt werden, da die Ausführung des einen Musters nicht die Anwendbarkeit des anderen Musters beeinträchtigt. In Szenario S_2 kann durch die Ausführung

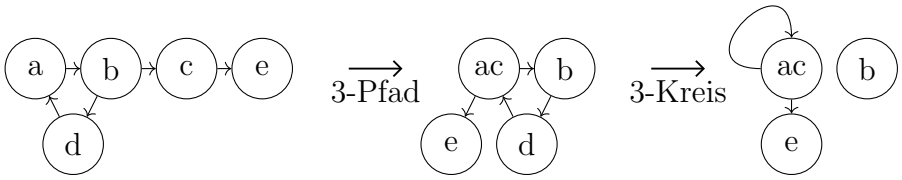


Abbildung 11: *3-Pfad*-Muster überschneidet *3-Kreis*-Muster

des *3-Kreis*-Musters das *3-Pfad*-Muster anschließend nicht angewendet werden, weil die Kante (a, b) entfernt wird. Das *3-Pfad*-Muster wird durch das

Grounded-Muster *bc* ersetzt. Wenn die umgekehrte Ausführungsreihenfolge angewendet wird, d.h., es wird zuerst das *3-Kreis*-Muster ausgeführt, dann ist im Anschluss das *3-Pfad*-Muster weiterhin anwendbar. Es sei hier aber angemerkt, dass falls das *3-Pfad*-Muster ebenfalls eine längere Angriffskette bildet, dann entsteht durch das Umhängen ein *2-zu-1*-Muster (vgl. Abb. 11). Dieses *2-zu-1*-Muster würde sich mit dem *3-Kreis*-Muster überschneiden wie es im Beweis des Theorem 1 bereits erläutert wurde.

F_3: Für ein *Grounded*-Muster *qr* und ein *3-Kreis*-Muster *def* kann eine Überschneidung nur mit $r = d$ exisitieren. Wird zuerst das *Grounded*-Muster ausgeführt, bleibt ef vom *3-Kreis*-Muster übrig, das wiederum ein *Grounded*-Muster darstellt (vgl. Abbildung 10b). Eine umgekehrte Ausführung führt zu dem gleichen Ergebnis.

F_4: Ein *3-Kegel*-Muster *ghi* und ein *3-Kreis*-Muster *def* können sich in den folgenden beiden Szenarien überschneiden:

- $S_1 : i = d$
- $S_2 : g = d$

Für S_1 ist die Reihenfolge analog zu F_3, bei dem sich das *Grounded*- und *3-Kreis*-Muster überschneiden, irrelevant. Dies gilt entsprechend in Szenario S_2, da das *3-Kreis*-Muster nach Ausführung des *3-Kegel*-Musters ausführbar ist und vice versa. □

Vollständige und präferierte Semantik

Nachdem die Auswirkungen der Anwendungsreihenfolge unter der *stabilen* Semantik analysiert wurden, werden nun die *präferierten* und *vollständigen* Semantiken betrachtet. Zur Wiederholung: der Transformationsschritt für das *3-Kreis*-Muster unterscheidet sich für beide genannten Semantiken im Vergleich zu der *stabilen* Semantik. Das *3-Kegel*-Muster ist für die *präferierte* Semantik mit Einschränkungen anwendbar, aber nicht für die *vollständige* Semantik.

Theorem 3 (Reihenfolge unter der vollständigen Semantik)**.** Gegeben sei ein AF F und dieser wird durch die Muster aus Kapitel 3.2, mit Ausnahme des *3-Kegel* Musters, zu F' in einer beliebigen Reihenfolge im Kontext der *vollständigen* Semantik transformiert bis kein Muster mehr anwendbar ist. Dann ist F' ein eindeutig verkleinerter AF und zudem der kleinstmögliche transformierte AF.

Analog zu den vorherigen Beweisführungen wird auch hier eine Falluntersuchung durchgeführt. Da nur das *3-Kreis*-Muster einen anderen Transformationsschritt gegenüber der *stabilen* Semantik besitzt, wird nur die Überschneidung mit einem weiteren *3-Kreis*-Muster und den anderen Mustern überprüft. Für die restlichen Fälle wird auf die Falluntersuchung aus Theorem 2 verwiesen. Folglich werden nun folgende Fälle betrachtet:

Überschneidung	
F_1 : *3-Kreis* $\times$ *3-Pfad*	F_3 : *3-Kreis* $\times$ *2-zu-1*
F_2 : *3-Kreis* $\times$ *3-Kreis*	F_4 : *3-Kreis* $\times$ *Grd.*

Tabelle 4: Fallunterscheidung für das *3-Kreis*-Muster

Beweis. F_1: Es überschneidet sich ein *3-Kreis*-Muster *def* mit einen *3-Pfad*-Muster *abc*. Beide Muster können sich mit einem oder zwei Argumenten überschneiden. Es sind folgende Szenarien möglich:

- $S_1 : \{a\} \subseteq \{d, e\}$
- $S_2 : a = f$
- $S_2 : (a = d \wedge b = e) \vee (a = e \wedge b = f) \vee (a = f \wedge b = d)$

In Szenario S_1 kann eine willkürliche Reihenfolge zwischen den beiden Mustern ausgewählt werden. Die Argumente d und e bleiben nach Ausführung des *3-Kreis*-Musters bestehen, analog gilt dieses auch für das *3-Pfad*-Muster. In Szenario S_2 gilt $a = f$. Falls das *3-Kreis*-Muster angewendet wird, dann wird a entfernt, somit ist das *3-Pfad*-Muster *abc* nicht mehr anwendbar. Allerdings erbt das Argument d, im Vergleich zum Transformationsschritt der *stabilen* Semantik, alle Angriffe von Argument a und es bildet sich ein neues *3-Pfad*-Muster *dbc*. Die umgekehrte Ausführung hat in S_2 keine Auswirkung auf das *3-Kreis*-Muster. Angenommen S_3 liegt vor und es wird erst das *3-Kreis*-Muster ausgeführt. Bei den Überschneidungen $(a = e \wedge b = f) \vee (a = f \wedge b = d)$ entsteht ein *2-zu-1*-Muster, das das *3-Pfad*-Muster ersetzt. Bei der Überschneidung $a = d \wedge b = e$ ist das *3-Pfad*-Muster nach Ausführung des *3-Kreis*-Musters weiterhin anwendbar. Auch hier ist die umgekehrte Ausführung ebenfalls möglich.

F_2: In F_2 überschneiden sich zwei *3-Kreis*-Muster $d_1e_1f_1$ und $d_2e_2f_2$. Es existieren zwei Überschneidungsszenarien mit:

- $S_1 : d_1 = d_2$
- $S_2 : d_1 = d_2 \wedge e_1 = e_2$

Liegt S_1 vor, so kann eine willkürliche Reihenfolge ausgewählt werden, da die Attacke auf sich selbst, die Anwendung des anderen Musters nicht verhindert. Dies trifft in S_2 ebenfalls zu, die Kante (d_1, e_1) wird nicht wie bei der stabilen Semantik entfernt.

F_3: Ein *2-zu-1*-Muster *jk* kann sich mit einen *3-Kreis*-Muster *def* immer nur in einem Argument überschneiden. Eine Überschneidung kann in den folgenden drei Szenarien auftreten:

- $S_1 : d = j$

- $S_2 : f = j$
- $S_3 : e = j$

Wenn S_1 mit $d = j$ vorliegt, dann folgt $\{d\}^- = \{k\}^- = \{f\}$. Durch Ausführung des *3-Kreis*-Musters wird das Argument k umgehangen und wird nun von d angegriffen. Dabei bilden die Argumente k und e ein neues *2-zu-1*-Muster. Wenn S_2 mit $f = j$ vorliegt, dann existiert zusätzlich ein *3-Pfad*-Muster *dek*. Das *3-Pfad*-Muster ersetzt das *2-zu-1*-Muster, falls Argument f durch das *3-Kreis*-Muster entfernt wird. Im letzten Szenario gilt $e = j$, dieses Szenario hat bei der *stabilen* Semantik zu unterschiedlich großen AFs geführt. Dies ist hier allerdings nicht möglich, da im Transformationsschritt die Kante (d, e) erhalten bleibt. Durch Ausführung des *3-Kreis*-Musters bleibt das *2-zu-1*-Muster unberührt. Angenommen das *2-zu-1*-Muster wird vor dem *3-Kreis*-Muster angewendet, dann hat dies keinen Einfluss auf die Anwendbarkeit des *3-Kreis*-Musters.

F_4: Dieser Fall ist identisch zu F_2 aus Lemma 2. Durch die Ausführung eines *Grounded*-Muster entsteht ein weiteres *Grounded*-Muster. Die umgekehrte Ausführung führt zum gleichen Ergebnis. □

Für die *präferierte* Semantik werden die vorherigen Ergebnisse verwendet. Es muss allerdings noch die Überschneidung mit dem *3-Kegel*-Muster überprüft werden.

Theorem 4 (Reihenfolge unter der präferierten Semantik)**.** Gegeben sei ein AF F, dieser wird durch die Muster aus Kapitel 3.2 zu F' in einer beliebigen Reihenfolge im Kontext der *präferierten* Semantik transformiert bis kein Muster mehr anwendbar ist. Dann ist F' ein eindeutig verkleinerter AF und zudem der kleinstmögliche transformierte AF.

Beweis. Ein *3-Kegel*-Muster *ghi* verletzt die *präferierte* Semantik nicht, falls g oder i sich gegen alle Angreifer verteidigen. Somit ist das Muster restriktiver als ein *3-Kegel*-Muster unter der *stabilen* Semantik. Daraus folgt, dass nicht mehr Überschneidungsszenarien existieren als für ein *3-Kegel*-Muster unter der *stabilen* Semantik. □

Zusammenfassend lässt sich festhalten, dass die Reihenfolge der Musteranwendung sich grundsätzlich nicht auf ein Endergebnis auswirkt. Nur bei der *stabilen* Semantik kann genau ein Überschneidungsszenario zu einem unterschiedlichen Ergebnis führen. Aufbauend auf diesem Ergebnis wird nun im folgenden Kapitel die Entwicklung einer Heuristik beschrieben.

5 Entwicklung einer Heuristik

Für die Entwicklung einer Heuristik werden zunächst die verwendeten Datensätze genauer analysiert. Anschließend wird die Funktionsweise des ursprünglichen Programmes aus [24] genauer erklärt.

5.1 Datensätze

Für die Untersuchung der Preprocessing-Methode werden die Instanzen, die für die *International Competition on Computational Models of Argumentation*, kurz *ICCMA*, erstellt wurden, herangezogen. Die *ICCMA* findet alle zwei Jahre statt und teilnehmende Solver müssen unterschiedliche Aufgaben (tasks) für gegebene Instanzen lösen. Die Datensätze aus den Jahren 2017 (vgl. [28]) und 2019 (vgl. [12]) bilden die Basis für die Datenanalyse dieser Arbeit. Die Instanzen der *ICCMA* 2021 (vgl. [35]) werden nicht betrachtet, da die AFs in der Regel zu groß sind und die Daten durch Bruteforce Algorithmen ermittelt wurden.

Die AFs der Datensätze 2017 und 2019 lassen sich in 13 verschiedene Graphenmodelle unterteilen: *Admbuster* (vgl. [13]), *SemBuster* (vgl. [14]), *Planning2AF* (vgl. [16]), *Traffic* (vgl. [20]), *ABA2AF* (vgl. [36]), *AFGen* (vgl. [41]) und *Datalog* (vgl. [45]), dessen Instanzen durch den *Dagger*-Generator (vgl. [46]) erstellt wurden. Die Instanzen der Modelle *Erdös-Renyi* (vgl. [26]), *Barabasi-Albert* (vgl. [3]), *Watts-Strogatz* (vgl. [44]) werden durch *AFBench-Gen2* (vgl. [19]) generiert und die *SccGenerator*, *GroundedGenerator*, *Stable-Generator* Instanzen durch das Benchmark Framework *Probo* (vgl. [17]).

Vorab ist festzuhalten, dass die beiden Datensätze in Ordner gruppiert sind. Teilweise lassen sich Überschneidungen der Datensätze finden, da einige AFs in beiden Datensätzen vorkommen. Für die nun folgende Datenanalyse wurden die Duplikate herausgefiltert; insgesamt enthalten beide Datensätze nach dem Herausfiltern der Duplikate zusammen 563 Instanzen. Einige Modellnamen wurden in den folgenden Graphen abgekürzt; die Tabelle 6 im Anhang enthält die Namenszuordnung. Bei der Initialsuche wurde für jeden Knoten eines AFs die Anzahl der ausführbaren Muster für die *stabile* Semantik ermittelt. Für ein *2-zu-1*-Muster wurden die zugehörigen Knotenpaare (a, b) bzw. (b, a) immer nur einfach gezählt. Analog wurde die gleiche Zählweise für ein *3-Kegel*- oder *3-zu-2*-Muster mit (a, b, c) und (b, a, c) angewendet.

Methodisch wird wie folgt vorgegangen: Zu Beginn werden die Graphenmodelle dahingehend untersucht, ob sie Ersetzungsmuster enthalten und somit von der Preprocessing-Methode überhaupt profitieren können. Die Abbildung 12 stellt die Anzahl der Instanzen differenziert nach ihrer Modellzugehörigkeit dar, die mindestens ein oder kein Muster enthalten. Hierbei fällt auf, dass die fünf Modelle *AFGen*, *Watt-Strogatz*, *Erdös-Renyi*, *SccGenerator* und *Sembuster* herausfallen, da die meisten bis alle Instanzen dieser Modelle gar keine Muster enthalten. Bei den *Traffic* und *Datalog* AFs können ebenfalls einige Instanzen nicht vom Preprocessing profitieren.

Der Aufbau eines *Sembuster* AFs ist sehr schematisch. Die Knoten sind in drei Klassen A, B und C unterteilt. Sie enthalten jeweils gleich viele und geordnete Knoten mit $i < j$. Ein Knoten A_i greift immer sich selbst an und wird ferner von B_i und B_j angegriffen. Ein Knoten aus B_j greift seinen Vorgänger B_i an und besitzt zusätzlich eine bidirektionale Angriffrelation zu C_j. Durch

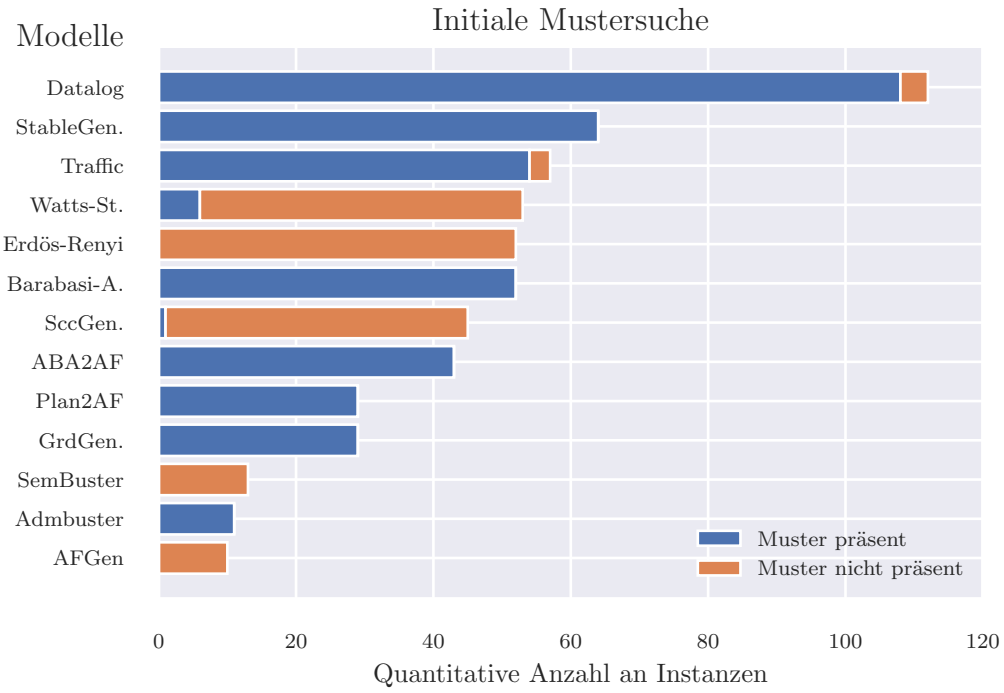


Abbildung 12: Blau: mindestens ein Muster - Orange: kein Muster

diese gleichförmige Struktur wird dementsprechend kein Muster aus Kapitel 3.2 gefunden.

Der *SccGenerator* konstruiert einen Graphen mit vielen stark verbundenen Komponenten (strongly connected Components, kurz *SCC*). Die Komponenten sind untereinander schwach verbunden. Es konnten nur einige wenige *3-Kegel*-Muster identifziert werden.

Ein *Erdös-Renyi* Graph, kurz *ER*-Graph, wird durch eine Wahrscheinlichkeitsberechnung generiert, die keine konkreten Strukturen produziert. Bei der Generierung werden alle Knoten erstellt, anschließend wird jedes Knotenpaar mit einer Wahrscheinlichkeit p verbunden. Das Ergebnis ist, dass die Knotengrade relativ gleichverteilt sind. Das *Watts-Strogatz* (*WS*) Modell orientiert sich an dem *ER* Modell. Im Unterschied dazu besitzt jedoch ein Graph, der durch das *WS* Modell generiert wird, einen größeren Cluster-Koeffizienten, als ein Graph zugehörig zum *ER* Modell. Der *AFGen* Generator implementiert das probabilistische Modell, das in [29] vorgestellt wurde. Das Modell ist ebenfalls an dem *ER* Modell angelehnt, aber je nach Parameterauswahl wird zwischen einem Knotenpaar ein gegenseitiger Angriff erstellt, um die Existenz einer Extension besser zu steuern.

Dass ein probabilistisches Modell Muster enthalten kann, zeigt das *Barabasi-Albert*-Modell (*BA*). Im Vergleich zum *ER* bzw. *WS* Modell lassen sich in allen Instanzen des *Barabasi-Albert* Modells Muster finden. Bei diesem Modell wird erst eine kleinere Anzahl an Argumenten generiert. Anschließend werden

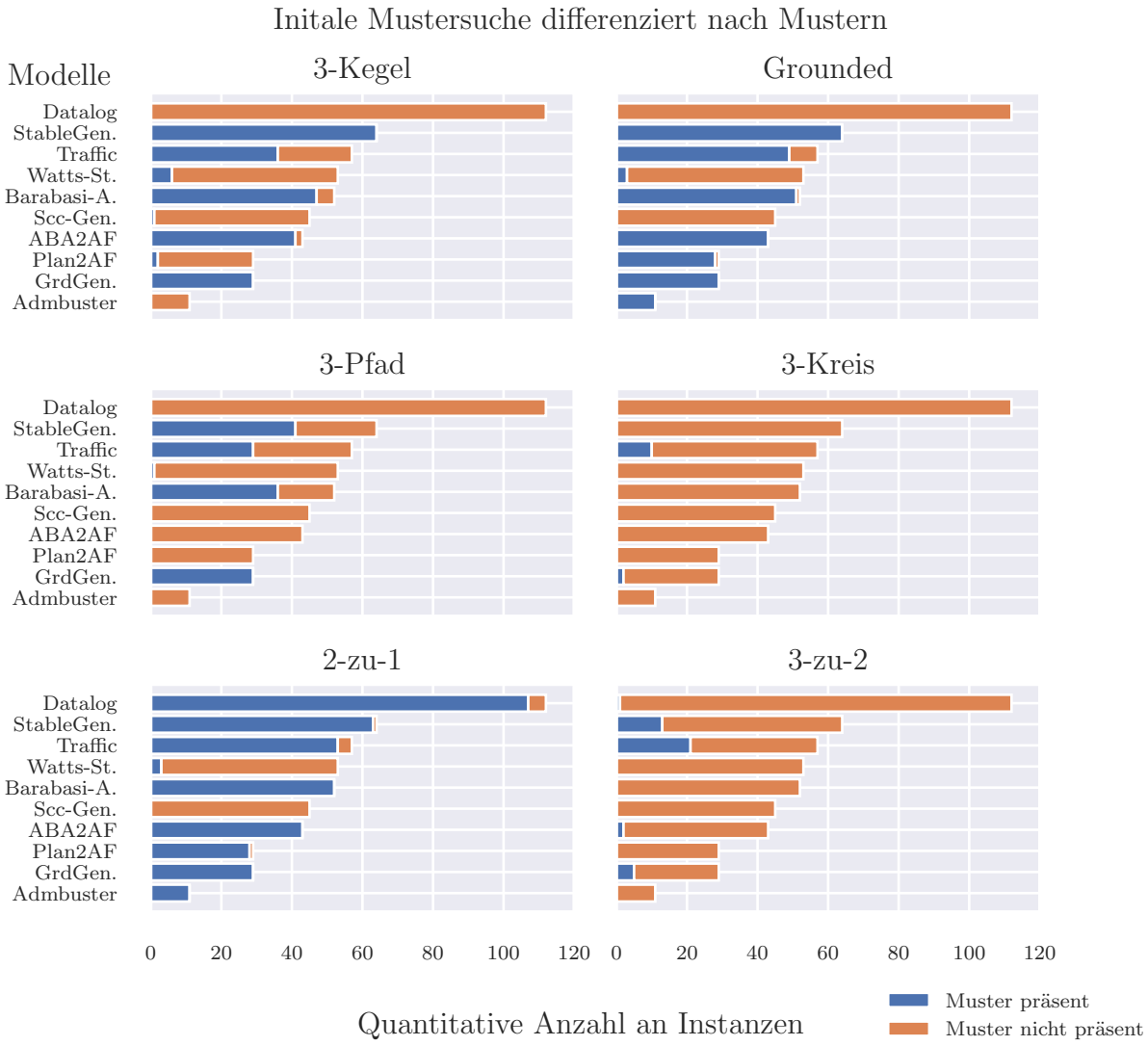


Abbildung 13: Blau: mindestens ein Muster - Orange: kein Muster

sukessive neue Knoten hinzugefügt und eine Kante wird eher mit Knoten, die höhere Knotengrade besitzen, gezogen.

Vor dem Hintergrund dieser Beobachtungen, soll nun im zweiten Schritt das Vorhandensein der einzelnen Muster genauer untersucht werden. Hierbei werden die Modelle *AFGen*, *Erdös-Renyi* und *Sembuster* allerdings nicht weiter betrachtet, da gar keine Muster zu finden sind.

Differenziert nach den Mustern taucht das *Grounded*-Muster in den meisten Fällen in mehr Instanzen auf als das *3-Kegel*-Muster (vgl. Abbildung 13). Insbesondere haben AFs der Modelle *Plan2AF* und *Admbuster* vereinzelt bzw. gar keine *3-Kegel*-Muster in der initialen Mustermenge. Die *Datalog* AFs haben weder *3-Kegel*- noch *Grounded*-Muster. Das *3-Kreis*-Muster lässt sich nur in einzelnen Instanzen der *Traffic* und *GroundedGen.* Modelle finden. Das *3-Pfad*-

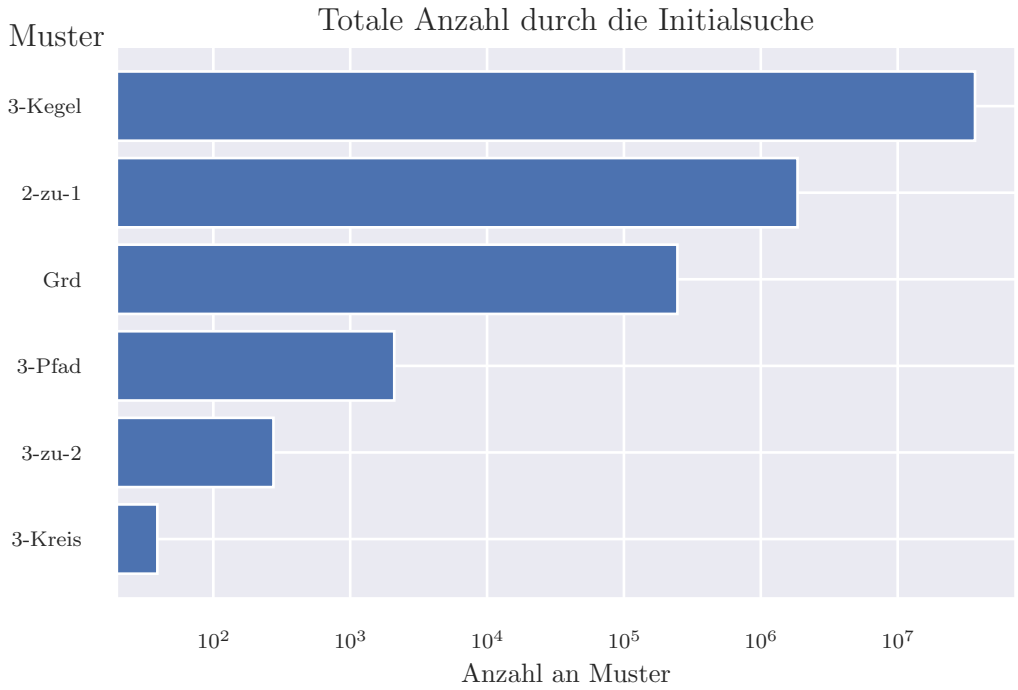


Abbildung 14: Totale Anzahl an Mustern über den gesamten Datensatz

Muster ist in den Modellen *Traffic*, *StableGen.*, *Barabasi-Albert* und *GroundedGen.* vorhanden. Bei dem letztgenannten Modell besitzen alle Instanzen mindestens ein *3-Pfad*-Muster; bei den anderen genannten Modellen existieren hingegen einige Instanzen ohne ein *3-Pfad*-Muster. Das *2-zu-1*-Muster findet sich in fast allen Instanzen, außer den bereits oben erläuterten Modellen *Watts-Strogatz* und *SccGenerator*. Das speziellere *3-zu-2*-Muster ist nur in den fünf Modellen *Datalog*,*Traffic*, *StableGen.*, *ABA2AF* und *GrdGen.* auffindbar, dann allerdings auch nur in wenigen Instanzen.

Nach der Betrachtung des generellen Vorhandenseins von Mustern, wird im dritten Schritt ein Blick auf deren quantitative Verbreitung geworfen.

Die Grundgesamtheit an Mustern ist sehr unterschiedlich verteilt. Die Muster *3-Kegel*, *2-zu-1* und *Grounded* sind bezüglich ihrer Anzahl stärker vertreten, als die *3-Pfad*-, *3-zu-2*- und *3-Kreis*-Muster. Diese Divergenz lässt sich unter anderem dadurch erklären, dass durch das *Grounded*-Muster viele unattakierte Argumente vorhanden sind. Dementsprechend existieren für das *3-Kegel*- und *2-zu-1*-Muster als auch für das *Grounded*-Muster selbst eine Vielzahl an möglichen Musterkombinationen.

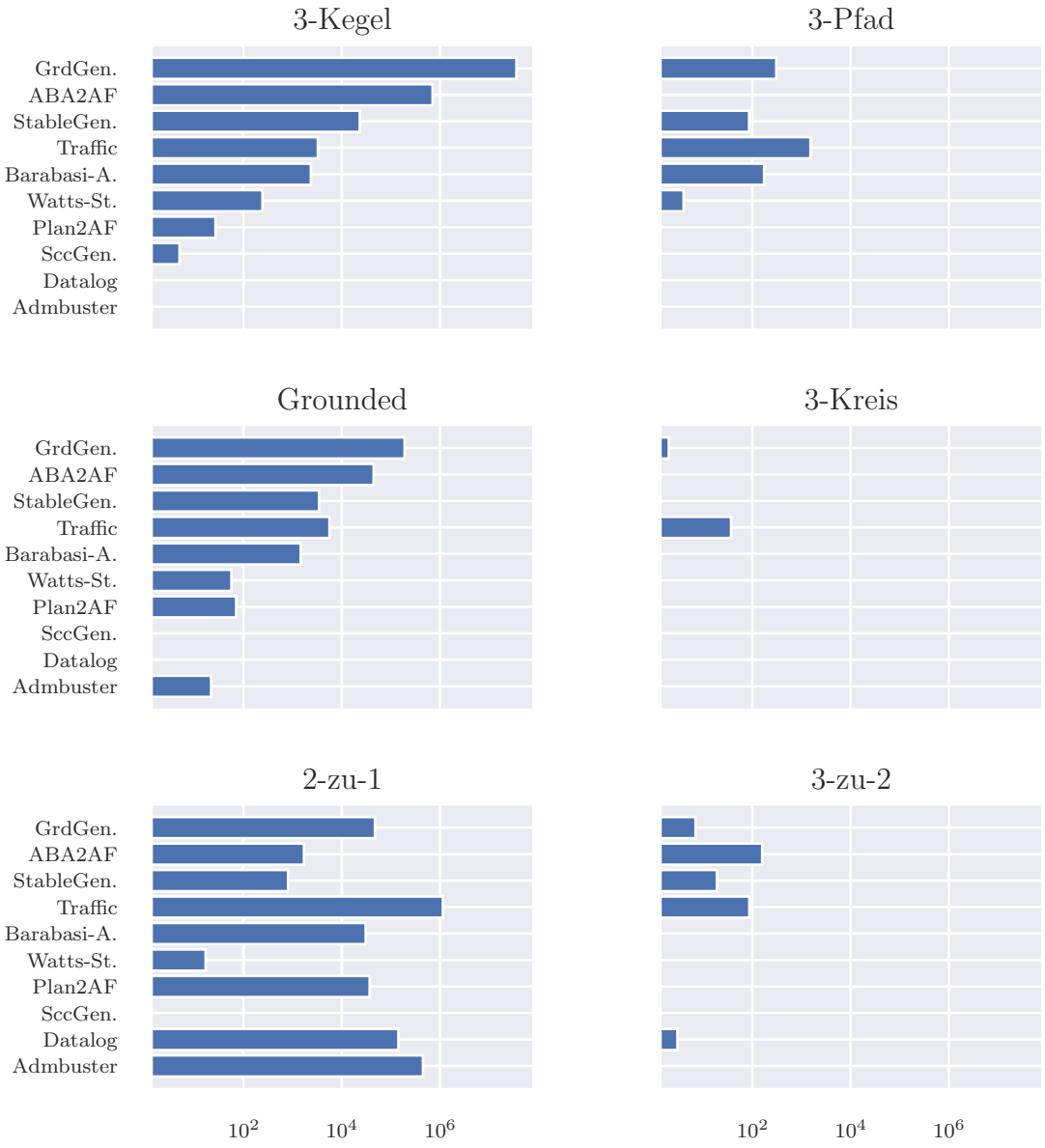


Abbildung 15: Totale Anzahl an Mustern über den gesamten Datensatz differenziert nach den unterschiedlichen Modellen

Bei einer genaueren Betrachtung der Anzahl der Muster mit Fokus auf die Modelle fällt auf, dass das *GroundedGenerator* Modell gefolgt von dem *ABA2AF* die meisten *3-Kegel*- sowie *Grounded*-Muster enthält (vgl. Abb. 15). Das *Admbuster* enthält kein *3-Kegel*-Muster, aber die zweithöchste Anzahl an *2-zu-1*-Muster. Die AFs des *Traffic* Modells sind im Bezug auf die Mustertypen sehr homogen verteilt. Insbesondere die Muster *3-zu-2*, *3-Kreis* und *3-Pfad* finden sich in den Instanzen. Von den *3-zu-2*-, *3-Kreis*- und *3-Pfad*-Mustern ist lediglich das *3-Kreis*-Muster nicht in den Instanzen eines *StableGenerator* Modells vorhanden. Für das *Watts-Strogatz* Modell kommen im absoluten

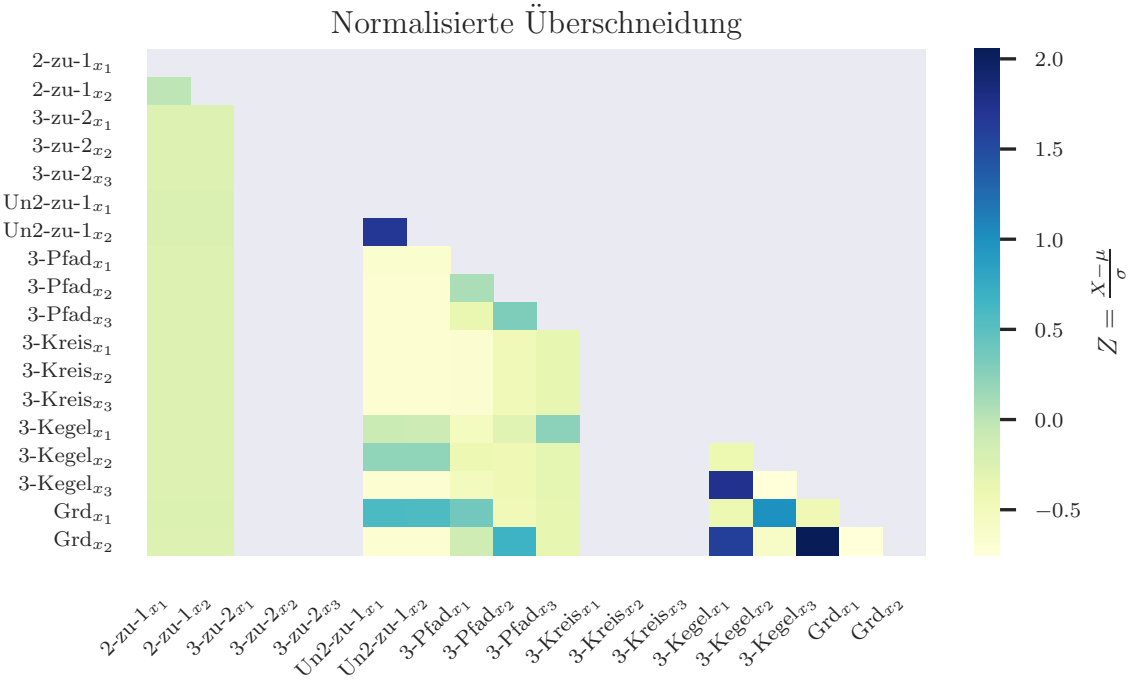


Abbildung 16: Spalten-orientierte Z-Transformation der überlappenden Musterknoten

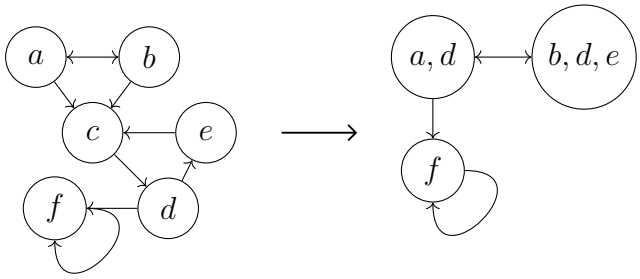
Vergleich weniger Muster vor als bspw. im Modell *Barabasi-Albert*.

Viertens, nun wird die Überschneidung der Muster betrachtet (vgl. Abb. 16). Dabei wurden die Daten durch eine spalten-orientierte Z-Transformation (Z-Standardisierung), mit $Z = \frac{X-\mu}{\sigma}$, normalisiert. Es werden die quantitativen Überschneidungen in einen z-Score überführt. Der z-Score stellt dar, inwieweit ein Wert vom Mittelwert relativiert durch die Standardabweichung entfernt ist. Ein Wert größer als 1 stellt eine relativ hohe Überschneidung dar.

Die grau markierten Zellen der *3-zu-2* und *3-Kreis* Argumente resultieren aus einer fehlenden Überschneidung. Ein Grd_{x_1} überschneidet sich häufiger mit einem 3-$Kegel_{x_2}$ und ein Grd_{x_2} mit einen 3-$Kegel_{x_1}$ Argument. Das heißt, durch die Ausführung eines *Grounded*-Musters fallen viele *3-Kegel*-Muster weg, aber wie in Kapitel 4 festgestellt, bleibt der Eliminierungsschritt des *3-Kegel*-Musters durch ein *Grounded*-Muster bestehen. Neben der stärkeren Korrelation zwischen den beiden Mustern, ist die Korrelation zwischen *Grounded*- und *3-Pfad*-Muster schwächer ausgeprägt. Dabei entfernt das *Grounded*-Muster das mittlere Argument aus einen *3-Pfad*-Muster. Mit sich selbst überschneidet sich ein *3-Pfad*-Muster ebenfalls und bildet in einem solchen Szenario eine Angriffskette. Das *2-zu-1*-Muster überschneidet sich selten bis gar nicht mit den anderen Mustern.

Vor der Beschreibung der Preprocessing Algorithmen ist zuletzt eine nähere Betrachtung der *2-zu-1*-Muster notwendig. Für die Validierung eines *2-zu-1*-

Beispiel 4 Fehler beim AFClingo - d & e werden zusammengefügt



Musters muss die komplette Angreifermenge mit der Angreifermenge eines anderen Knoten überprüft werden. Solche Überprüfungen sind rechenintensiv und sollten möglichst vermieden werden. Unter den *2-zu-1*-Mustern, die bei der Initialsuche identifiziert wurden, werden 99.9% der Muster durch ein oder zwei gemeinsame Argumente angegriffen (vgl. Tabelle 7 im Anhang). Die Muster, die gar keinen Angreifer besitzen, wurden dabei ausgeschlossen. Bei einer analogen Betrachtungsweise für das *3-zu-2*-Muster besitzen nur knapp 30% der Muster maximal zwei gemeinsame Angreifer.

5.2 AFClingo

Bei dem Preprocessing-Algorithmus, der im Rahmen der Arbeit [24] entwickelt wurde, handelt es sich nach Aussage seines Entwicklers um einen reinen Prototypen. Ziel des Programmes ist die Anwendbarkeit der Preprocessing-Methodik zu demonstrieren, indem aufgezeigt wird, dass ein vereinfachter AF die Berechnungszeit von Solvern verkürzt. Das Programm ist in Python implementiert und ruft den externen Answer Set Programming (ASP) Solver *Clingo* (Version 5.3, vgl. [30]) auf. Dementsprechend trägt der bislang unbenannte Algorithmus in dieser Arbeit die Bezeichnung *AFClingo*.

Der Pseudocode ist im Algorithmus 1 skizziert. Das Programm führt die Muster iterativ aus. Dazu ruft es *Clingo* parametrisiert mit den Pfaden des AFs sowie des ASP encodierten Musters auf. Für einen Mustertyp werden immer alle Muster von Clingo identifiziert. Davon wird das erste Muster angewendet, der Rest wird verworfen. Die Mustertypen sind in einer Liste (Patternlist) wie folgt angeordnet: *3-Kegel-*, *3-Pfad-*, *3-Kreis-*, *unattackierte 2-zu-1-*, *4-Kegel-*, *2-zu-1-*, *4-Pfad-*, *3-zu-2-*, *Grounded*-Muster. Die Mustersuche erfolgt immer nach der in der *Patternlist* definierten Ordnung. Angenommen ein *2-zu-1*-Muster wird angewendet, dann sucht der Algorithmus in der nächsten Iteration wieder zuerst nach *3-Kegel*-Mustern. Die Schleife bricht ab, wenn keine Muster mehr gefunden werden. Ein Problem, das bei großen Instanzen oder komplexen Instanzen passieren kann, ist, dass *Clingo* in ein Timeout läuft und es folglich kein Muster zurückliefert. Ein weiterer Kritikpunkt ist, dass die

vorliegende Version einen Programmfehler enthält, sodass teilweise keine originalgetreuen Transformationen gemäß den Musterdefinitionen durchgeführt werden. Der AF in Beispiel 4 illustriert dies exemplarisch. Die Knoten d und e werden zusammengefügt, obwohl die Knoten sich attackieren.

Algorithm 1 AFClingo

Require: AF-Instanz *inst*
  $PatternList \leftarrow$ [3-Kegel, 3-Pfad, 3-Kreis, Unattack. 2-zu-1, 4-Kegel, 2-zu-1, 4-Pfad, 3-zu-2, Grounded]
  $ActiveMuster \leftarrow PatternList$
  while *ActivePattern has Elements* **do**
    $pattern = ActivePattern[0]$
    $AllPossiblePatterns \leftarrow Clingo(pattern, inst)$
    if *AllPossiblePatterns is Empty or Timeout* **then**
      Delete pattern from ActivePattern
      End current iteration
    else
      $Transform(AllPossiblePattern[0])$
    end if
    $ActivePattern = PatternList$
  end while
  return AF-Instanz

5.3 AFPreprocessing

In diesem Abschnitt wird das Programm *AFPreprocessing* (*AFPre*) vorgestellt, das im Zuge dieser Arbeit programmiert wurde. Das Programm unterstützt das Vereinfachen von AFs für die *vollständige*, *präferierte* sowie *stabile* Semantik. Es wurden zwei Suchstrategien implementiert, die sich hinsichtlich der Restriktion der Identifzierung von Mustern unterscheiden und im weiteren Verlauf genauer vorgestellt werden. Im Anhang befindet sich ein UML-Diagrammm, das die grobe Architektur visualisiert (vgl. 22). *AFPreprocessing* baut auf der *Tweety* Bibliothek (vgl. [42]) auf. Die Bibliothek ist in Java implementiert und stellt Klassen und Methoden für die Abstrakte Argumentation nach Dung bereit. Für den eigenen Ansatz wurde auf die Kernklasse *DungTheory* sowie die zugehörige *Argument*-Klasse zum größten Teil verzichtet, da die zu Grunde liegende Datenstruktur einige Nachteile für das Preprocessing besitzt, die an dieser Stelle kurz erläutert werden sollen.

Die *DungTheory* verwaltet alle eingehenden und ausgehenden Angriffe eines Arguments in zwei *HashMaps*. Eine Map Datenstruktur verwaltet Daten allgemein in einem Schlüssel-Werte-Paar. Bei einer *HashMap* ergibt sich der Schlüssel aus dem Hashwert (hash code) eines Objekts. Intern verwaltet die

HashMap eine Hashtabelle und die Hashwerte verweisen auf den jeweiligen Index der Tabelle. Dabei kann es zu einer Kollision kommen, wenn unterschiedliche Hashwerte auf den gleichen Index verweisen. Deshalb steckt hinter jedem Index ein *Bucket* (Behälter). Ein solcher *Bucket* kann wiederum mehrere Objekte enthalten. Für die Verwaltung der Objekte verwendet ein *Bucket* selber einen Binärbaum als Datenstruktur. Um nun die Nachbarn eines Arguments zu betrachten, muss folglich der Hashwert eines Arguments berechnet und anschließend die Nachbarn eines Arguments in einem *Bucket* gefunden werden. In der Theorie hat eine *HashMap* eine Komplexität von O(1), allerdings hat es sich gezeigt, dass die wiederholenden Berechnungen der Hashfunktion bei der Mustersuche einen Nachteil darstellen. Aus diesem Grund wurde eine *ExtendedArgument*-Klasse erstellt. Diese enthält alle attackierenden sowie attackierten Argumente als separate Variablen. Damit entfällt die Berechnung der Hashfunktion, um an die Nachbarn zu gelangen. Zusätzlich wird die Anzahl der Angreifer sowie der Attackierten in gesonderten Variablen gespeichert, da diese Werte ein wichtiges Entscheidungskriterium in der Mustersuche darstellen.

Vor dem Hintergrund des Zusammenfügens von Argumenten wurden ebenfalls Änderungen an der *equals*- sowie *hashcode*-Methode in der *ExtendedArgument*-Klasse im Vergleich zu der *Argument*-Klasse vorgenommen. Das Programm *AFClingo* fügt Argumente zusammen, indem die Strings der Argumentennamen mit einem Unterstrich verbunden werden. Zunächst wurde ein analoger Ansatz bei der Implementierung verfolgt. Bei kleineren Mengen an Argumenten ist dies nicht gravierend, allerdings nimmt bei größeren zusammengefügten Argumenten die Performance rapide ab. Dies liegt daran, dass nach jeder Zusammenfügungsoperation der Name eines neuen Arguments sortiert werden muss. Das Sortieren ist notwendig, damit die *equals*- bzw. *hashcode*-Methode unter einer willkürlichen Ausführungsreihenfolge korrekt funktionieren. Bei längeren Namen ist die Sortierung kostenintensiv hinsichtlich der Performance. Aus diesem Grund verwaltet eine *ExtendedArgument*-Klasse alle enthaltenden *Argument*-Objekte in einem Set. Diese Menge wird nicht für die *equals*- bzw. *hashcode*-Methode verwendet, sondern eine UUID (Universally Unique Identifier), die bei der Initialisierung gesetzt wird. Dadurch wird das Zusammenfügen von Argumenten beschleunigt. Nachdem die Datenstruktur bekannt ist, soll nun die Funktionsweise des Algorithmus erläutert werden. Das Preprocessing eines AFs teilt sich in drei Stufen auf:

- Muster-Kandidaten identifizieren
- Muster-Kandidaten validieren
- Muster transformieren

Die getrennte Betrachtung der Identifizierung und Validierung von Kandidaten ermöglicht die Implementierung von unterschiedlichen Suchstrategien. Beide Schritte bilden zusammen einen Berechnungschritt, der parallel verarbeitet

Algorithm 2 AFPreprocessor

```
Require: AF-Instanz inst
  G ← Merge all unattacked Arguments
  AffectedArgs ← Apply Grounded G
  Patterns ← SearchPattern(AllArguments).
  AffectedArgs ← Apply(Patterns) ∪ AffectedArgs
  while True do
      Patterns ← SearchPattern(AffectedArgs)
      if Patterns.size == 0 then
          break
      end if
      AffectedArgs ← Apply(Patterns)
  end while
  return AF-Instanz
  function SearchPattern(arguments)
      for arg in arguments do                          ▷ Loop parallelization
          3Path-Cand. ← threePathCandidate(arg)
          3Path-Pattern ← validate3Path(3Path-Cand.)
          3Cone-Cand. ← threeConeCandidate(arg)
          3Cone-Pattern ← validatethreeCone(3Cone-Cand.)
          2to1-Cand. ← two2oneCandidate(arg)
          2to1-Pattern ← validateTwo2one(2to1-Cand.)
          3Circle-Pattern ← validate3Circle(3Path-Cand.)
          Grd.-Cand. ← grdCandidate(arg)
          Grd.-Pattern ← validateGrounded(Grd.-Cand.)
      end for
      return patterns
  end function
  function threePathCandidate(arg)
      if arg.getParentSize == 1 then
          for each arg.child.getParentSize == 1 do
              BuildCandidate()
          end for
          return candidates
      end if
  end function
  function threeConeCandidate(arg)
      if arg.getParentSize == 1 then
          return BuildCandidates()
      end if
  end function
  function two2OneCandidate(arg)
      if arg.getParentSize <= 2 then
          return BuildCandidates()
      end if
  end function
```

wird. Konkret ermittelt ein Thread für ein Argument die Kandidaten und validiert diese anschließend. Im darauf folgenden Schritt werden die Muster sequentiell angewendet. Dazu werden in einem Transformationsschritt alle Muster angewendet, sofern deren Argumente nicht Teil eines bereits angewandten Musters waren. Der Suchraum wird klein gehalten, indem im nächsten Suchschritt nur Argumente von angewandten Mustern des vorherigen Suchschrittes und ihren direkten Nachbarn betrachtet werden.

Um die Menge der Kandidaten klein zu halten, sucht der Algorithmus nach speziellen Knoteneigenschaften abhängig von den Mustern. Für das *3-Pfad*- sowie *3-Kreis*-Muster wird das mittlere Argument x_2 gesucht. Bei beiden Mustern darf x_2 sowie x_3 nur einen Angreifer besitzen. Die Suche nach dem *3-Kegel*-Muster geht ähnlich vor. Es wird ein x_2 gesucht und als Restriktion gilt, dass es von x_1 angegriffen werden muss (vgl. Abb. 17). Anderenfalls bildet das x_2 ebenfalls ein *Grounded*-Muster, das zum gleichen Ergebnis führt.

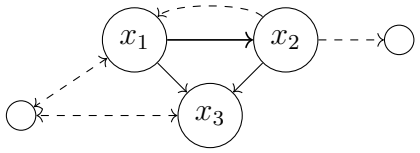


Abbildung 17: Restriktives 3-Kegel

Für ein *2-zu-1*-Muster ist die Identifzierung eines Kandidaten nicht so trivial, da eine unterschiedliche Anzahl an Angriffen auf die Argumente existieren kann. Theoretisch müssten aufwendige Mengenvergleiche durchgeführt werden, um alle *2-zu-1*-Muster zu finden. Praktisch ist dies aber nicht sinnvoll, da sich die Laufzeit bei dieser Vorgehensweise deutlich verlängert. Wie in Kapitel 5.1 bereits ausgeführt, werden 99.9% der Argumente von *2-zu-1*-Mustern, die bei der Initialsuche identifiziert wurden, von maximal zwei Angreifern attackiert. Unter diesem Blickwinkel unterscheiden sich die eingangs erwähnten Suchstrategien. Für die Strategie $AFPre_{fast}$ gilt als Einschränkung, dass nur *2-zu-1*-Muster mit maximal zwei Angreifern identifziert werden. Das *3-zu-2*-Muster wird nicht gesucht, da es zu selten vorkommt und die Identifizierung zu kostenintensiv hinsichtlich der Performance ist. Für die Strategie $AFPre_{complete}$ gelten die eben genannten Einschränkungen nicht.

Wie bereits angesprochen, wendet ein Transformer die Muster sequentiell an. Ein *2-zu-1*-Muster wird immer vor einem *3-Kreis*-Muster ausgeführt. Dies garantiert aber nicht, dass ein AF immer zu dem kleinstmöglichen AF transformiert wird. Infolge der Einschränkung des *2-zu-1*-Musters, kann es passieren, dass ein *2-zu-1*-Muster erst in einer späteren Iteration identifziert wird. Angenommen, die Argumente des *2-zu-1*-Musters besitzen zehn gemeinsame Angreifer, die sukkzessive entfernt werden. Dann hat die Heuristik eventuell in der Zwischenzeit das *3-Kreis*-Muster bereits ausgeführt. In Anbetracht des quantitativ seltenen Vorkommens eines *3-Kreis*-Musters wird dieser Nachteil aber nicht als so gravierend eingeschätzt.

Unter den stabilen Semantiken wird zusätzlich überprüft, ob das Argument x_1 eines *3-Kreis*-Musters nur von x_3 angegriffen wird. Falls diese Bedingung zu-

trifft, wird eine Ausnahmebedingung (Exception) ausgelöst und die Heuristik gibt einen AF mit nur einem selbstattackierenden Argument zurück. Anderenfalls markiert die Heuristik das Argument und die Bedingung wird jedes Mal überprüft, wenn ein Angreifer des Arguments x_1 entfernt wird.

6 Experiment

Nachdem die Preprocessing Algorithmen vorgestellt wurden, steht nun der praktische Nutzen der Methodik im Mittelpunkt. Bei der nachfolgenden Untersuchung sollen die Fragen geklärt werden:

- Wie stark fällt die Reduzierung der einzelnen Modelle aus?
- Wie verhalten sich die Laufzeiten der Preprocessing-Methoden?
- Kann eine kürzere Laufzeit für die Solver erreicht werden?

6.1 Verkleinerung der AFs

Zunächst werden die Ergebnisse der Verkleinerung von AFs genauer betrachtet. Die Tabelle 5 stellt die durchschnittliche Anzahl an Argumenten vor und nach der Verkleinerung für die *stabilen* Semantiken dar. Die Verkleinerung wurde durch *AFPreprocessing* ermittelt. Die Dimension der Zeit ist dabei unerheblich; aus diesem Grund wurde kein Timeout-Parameter gesetzt. Wie bereits in Kapitel 5.1 ausgeführt, eignet sich die Methode nicht für die Modelle *Erdös-Renyi*, *Sembuster*, *SccGenerator* und kaum für *Watt-Strogatz*. Die Instanzen des *StableGenerators* verkleinern sich im Durchschnitt um über 20%. Die *Traffic* Instanzen verkleinern sich im Schnitt um über 80%. An dieser Stelle sei nochmal angesprochen, dass ein kleiner Teil der *Traffic* AFs sich auf Grund der Abwesenheit von Mustern nicht verkleinert (vgl. Abb. 12). Die deutliche Reduzierung resultiert daraus, das einige Instanzen ein unattackiertes *3-Kreis*-Muster enthalten. Folglich besitzen diese Instanzen unter den *stabilen* Semantiken keine Extension. Die Heuristik *AFPreprocessing* verkleinert diese AFs zu einen AF mit einem selbstattakierenden Argument. *Planning2AF* und *Barabasi-Albert* Instanzen profitieren ebenfalls signifikant durch die Methodik. Die stärkste Reduzierung erfolgt bei den AFs der Modelle *ABA2AF*, *GroundedGenerator* und *Admbuster* mit über 98%. Zudem fällt das letztgenannte Modell dadurch auf, dass jeder verkleinerte AF nur ein Argument enthält.

Bei der Ermittlung wurden *2-zu-1*-Muster mit mehr als zwei Angreifern und alle *3-zu-2*-Muster aus der Untersuchung ausgeschlossen. Würden diese ebenfalls angewendet, reduzierten sich die AFs des *Datalog* Modells um durchschnittlich weitere 77,5%. Für die Instanzen des *ABA2AF* Modells reduzierte sich dieser Wert um 0,5%. Bei den restlichen Modellen konnten keine signifikanten Reduzierungen beobachtet werden.

Model	$\varnothing Argumente_{org}$	$\varnothing Argumente_{pre}$	Reduzierung (%)
ABA2AF	448,3488	8,6279	98,0756
Admbuster	316750,0	1,0	99,9996
AFGen	189,6	189,6	0,0
Barabasi-Albert	126,7692	33,0	73,9684
Datalog	76,0625	69,0892	9,1677
Erdös-Renyi	366,5	366,5	0,0
GroundedGenerator	6593,7931	1,0689	99,9837
Planning2AF	366,3103	76,6896	79,0642
SccGenerator	3981,8888	3981,7777	0,0027
SemBuster	2889,2307	2889,2307	0,0
StableGenerator	449,5625	355,1875	20,9926
Traffic	1220,3508	280,9824	76,9752
Watts-Strogatz	379,2452	369,4716	2,5771

Tabelle 5: Durchschnittliche Anzahl der Argumente vor (org) und nach (pre) dem Verkleinern unter der *stabilen* Semantik

Der Tabelle 8 im Anhang können die Reduzierungen für die *vollständigen* und *präferierten* Semantiken entnommen werden. Unter der *stabilen* Semantik fällt die durchschnittliche Reduzierung für *Traffic* Modelle höher aus als im Vergleich zu den beiden letztgenannten Semantiken. Geschuldet ist dies den im Vorfeld genannten unattackierten *3-Kreis*-Mustern. Obwohl das *3-Kegel*-Muster quantitativ am häufigsten in der Initialsuche gefunden wurde und für die *vollständigen* Semantiken nicht anwendbar ist, hat dies kaum Auswirkungen auf die durchschnittliche Reduzierung der AFs.

Für den *ICCMA 2021* Datensatz wurden ebenfalls die AFs durch das Programm *AFPreprocessing* für die *stabile* Semantik vereinfacht. Die AFs sind eine Kombination aus *Erdös-Renyi* und *Barabasi-Albert* Instanzen. Es wurden 3% der AFs vereinfacht und davon reduzierte sich die Argumentenanzahl um durchschnittlich 0,1%.

6.2 Laufzeitanalyse

Die Experimente in diesem Kapitel wurden unter folgender Umgebung ausgeführt: *Intel Haswell* mit 8 Kernen mit jeweils 3.4Ghz, 64GB RAM, Linux-5.4.0-131-generic *Ubuntu 20.04*. Jedes Experiment wurde drei Mal ausgeführt, auf der Basis der Einzelergebnisse wurde ein arithmetischer Mittelwert gebildet.

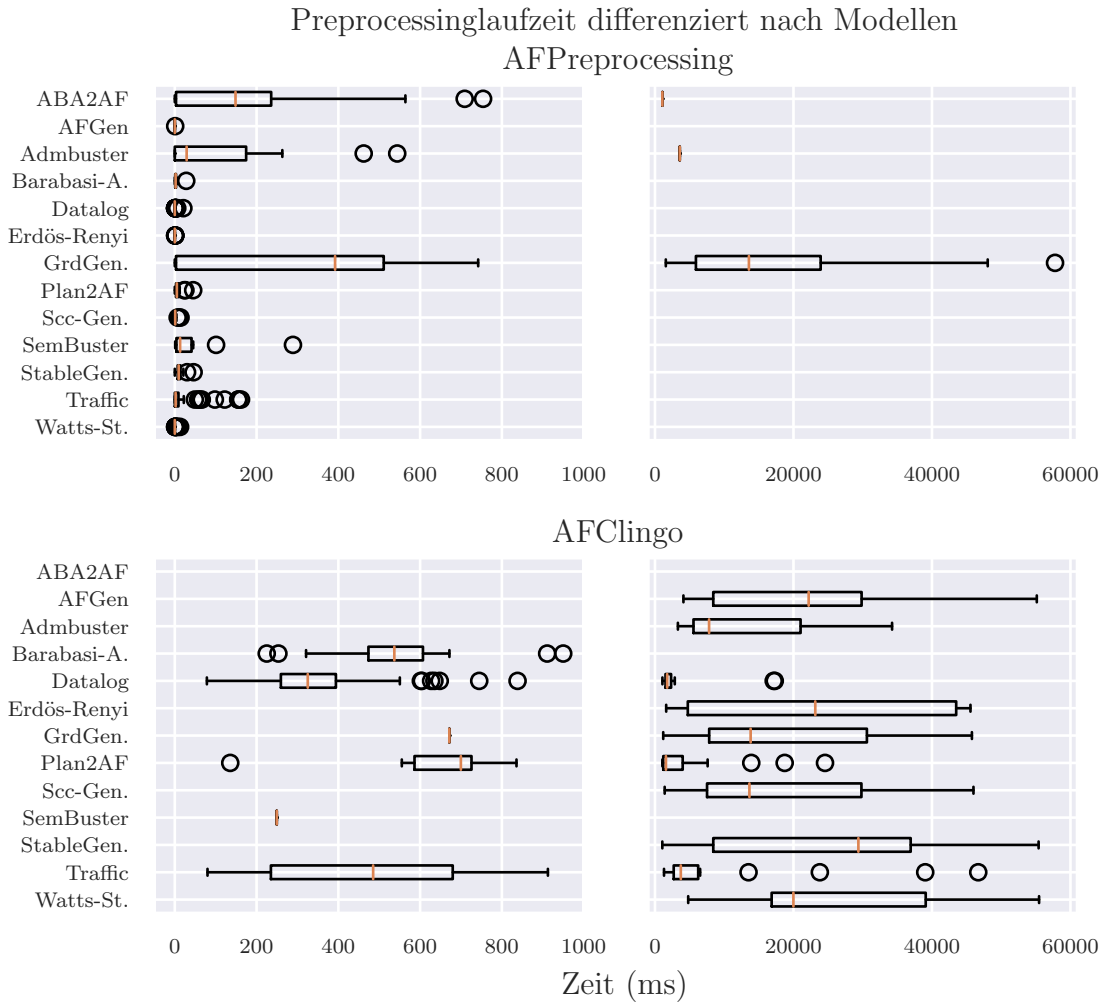


Abbildung 18: Lauzeitanalyse der Implementierungen für die *stabile* Semantik

6.2.1 Preprocessing Methoden

Es werden nachfolgend die Laufzeiten der beiden Preprocessing Methoden gegenüber gestellt. Für das Experiment wurde ein Timeout von 60 Sekunden gewählt. Die Abbildung 18 stellt in der oberen Hälfte die Laufzeit der Heuristik *AFPreprocessing* (vgl. Algorithmus 2) und in der unteren Hälfte die Laufzeit des Prototyps *AFClingo* (vgl. Algorithmus 1) dar. Die Graphen in der linken Hälfte enthalten alle Datenpunkte zwischen 0 und 1 Sekunde und die rechte Hälfte alle über 1 bis 60 Sekunde(n). Das Modell *ABA2AF* führte bei *AFClingo* zu einem Programmabsturz, folglich konnten für dieses Modell keine Daten erhoben werden.

AFPreprocessing verarbeitet über 90% der Instanzen in einem Zeitraum von unterhalb einer Sekunde. *AFClingo* verarbeitet in dieser Zeitspanne im Ver-

gleich dazu nur knapp 20%. Mit über 50% verkleinert *AFClingo* den Großteil der AFs zwischen 1 und 60 Sekunden. Folglich läuft der Algorithmus für ca. 30% der Instanzen in einen Timeout. Im Gegensatz dazu laufen in Anwendung von *AFPreprocessing* nur knapp 5% der Fälle in einen Timeout. Die Tabelle 9 im Anhang schlüsselt die Modelle auf, die zu einem Timeout führen. Vor allem die Modelle, die keine Muster enthalten, führen bei *AFClingo* zu einem Timeout. Die Heuristik verarbeitet solche AFs innerhalb von maximal 500ms. Die Modelle *Admbuster* und *GroundedGenerator* führen bei beiden Implementierungen teilweise zu Timeouts. Diese Modelle zeichnen sich dahingehend aus, dass sie mit bis zu 2.000.000 Argumenten im Verhältnis zu den übrigen Modellen wesentlich größer sind und zusätzlich in der Regel auf ein Argument verkleinert werden. Die Transformationsschritte für einen solchen AF nehmen eine signifikante Zeit in Anspruch. Einige Stichprobenanalysen mit einem Java Laufzeit Profiler haben diese Annahme für die Heuristik Implementierung bestätigt.

Es lässt sich insofern darlegen, dass die Heuristik *AFPreprocessing* mit ihrer parallelisierten und optimierten Suche eine kürzere Laufzeit benötigt, als der Prototyp *AFClingo*. Bedingt durch das längere Laufzeitverhalten wird *AFClingo* bei der weiteren Untersuchung nicht betrachtet.

6.2.2 DC-ST

Für die D(ecide)C(redelous)-ST Aufgabe muss ein Solver für die *stabile* Semantik entscheiden, ob ein gegebenes Argument in einer Extension enthalten ist. Das Entscheidungsproblem ist NP-vollständig (vgl. [23]). Die Argumente wurden durch *probo2* (vgl. [33]) zufallsgeneriert. Auf Grund der strukturellen Unterschiede zwischen den ursprünglichen und den verkleinerten Instanzen, wurden die zufallsgenerierten Argumente der verkleinerten Instanzen für beide Datensätze verwendet. Die Instanzen, die zu einem Timeout bei *AFPreprocessing* führten, sind nicht enthalten.

Die Laufzeitanalysen wurden für die Solver *A-Folio DPDB* (vgl. [27]), *FUDGE* (vgl. [43]), *PYGALF* (vgl. [1]), *μ-toksia* (vgl. [37], Version 2021) sowie *Heureka* (vgl. [31]) durchgeführt. Die Solver *Argtools* (vgl. [38]) und *Aspartix v21* (vgl. [25]) konnten auf Grund von technischen Schwierigkeiten nicht untersucht werden. Die Solver *A-Folio DPDB*, *FUDGE*, *PYGALF* und *μ-toksia* verwenden SAT-Solver und gehören dementsprechend zu der Klasse der reduktionsbasierten Solver. *Heureka* verfolgt einen labelbasierten Ansatz und gehört zu der Klasse der direkten Solver. *PYGALF* greift auf den *Glucose* SAT-Solver (vgl. [2]) zurück, *FUDGE* verwendet den *CaDiCaL* Solver (vgl. [10]) und *μ-toksia* den *CryptoMiniSAT* Solver (vgl. [40]). *A-Folio DPDB* verwendet für die jeweilige Aufgabe unterschiedliche Strategien. Für die hier betrachtete Aufgabe ruft das Programm *μ-toksia* auf. Die reduktionsbasierten Solver haben in der Vergangenheit bei der *ICCMA* die besten Resultate geliefert und sind deshalb hier stärker vertreten.

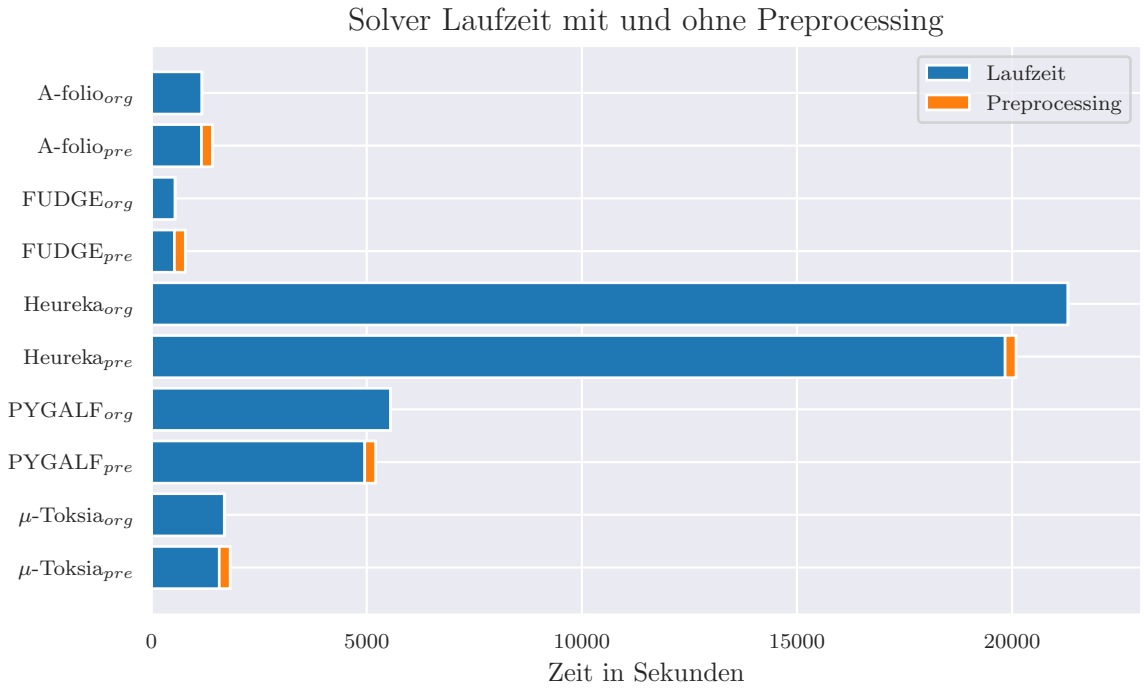


Abbildung 19: Laufzeiten der Solver mit (pre) und ohne (org) Preprecessing

Die Abbildung 19 visualisiert die Gesamtlaufzeit aller Solver und die Abbildung 24 im Anhang stellt dies im Detail dar.

Die Solver *A-folio DPDB*, *FUDGE* und *µ-Toksia* erreichen durch das Verkleinern der Instanzen keine kürzere Gesamtlaufzeit. Zwar verkürzen sich die Laufzeiten geringfügig, aber der Mehraufwand für das Verkleinern ist in Summe dennoch höher. Beim *Pygalf* Solver zeichnet sich ein umgekehrtes Bild. Die Gesamtlaufzeit konnte durch das Verkleinern reduziert werden. Ausschlaggebend sind die Instanzen des *GroundedGenerator* und *ABA2AF* Modells. Zudem lief der Solver als einziger reduktionsbasierter Solver für einige Instanzen des Modells *SemBuster* in einem Timeout, sodass das Preprocessing keine Verbesserung herbeiführen konnte.

Als Vertreter für die direkten Solver besitzt *Heureka* mit Abstand die längste Gesamtlaufzeit. Durch das Preprocessing wurde die Laufzeit um knapp 6% reduziert. Ausschlaggebend für die verkürzte Laufzeit, war primär die Verringerung der Timeouts. Der Solver lief für *Traffic* Instanzen nicht mehr in einen Timeout und für AFs des *StableGenerators* konnte dies um 10% reduziert werden.

Alles in allem, ist der Einfluss auf reduktionsbasierte Solver im Allgemeinen nicht ausreichend, um eine kürzere Laufzeit zu erreichen. Die Identifizierung von unattackierten *3-Kreis*-Mustern führte für die reduktionsbasierten Solvern ebenfalls zu keiner signifikanteren Beschleunigung. Vor allem für die Modelle

GroundedGenerator sowie *Admbuster* ist der Overhead durch das Verkleinern zu hoch. Durch das Verkleinern wird praktisch eine Lösung berechnet, da in der Regel das Endergebnis nur einen Knoten enthält. Allerdings führt das Transformieren dies nicht effizient durch. Für solche Fälle muss entweder der Transformationsprozess optimiert oder eine geeignete Abbruchstrategie entwickelt werden.

6.2.3 EE-CO

Es wird nachfolgend die EE-CO-Aufgabe untersucht. Bei dieser muss ein Solver für einen gegebenen AF unter der *vollständigen* Semantik alle Extensionen berechnen. Das Enumerationsproblem ist ein stark unlösbares Problem (vgl. [34]). Bei der DC-ST-Aufgabe war der Overhead des Verkleinerns für die State-of-the-Art-Solver zu hoch. Deshalb soll hier untersucht werden, ob selbiges für eine Aufgabenstellung gilt bei der immer alle Extensionen berechnet werden müssen. Die Aufgabe wurde letztmalig bei der *ICCMA 2019* gestellt, die Solver aus dem vorherigen Kapitel unterstützen diese Aufgabenstellung allerdings nicht. Aus diesem Grund werden die Laufzeiten der Solver *μ-toksia* (Version 2019) und *THEIA* (vgl. [32]) untersucht. *THEIA* gehört zu der Klasse der direkten Solver und ist an *Heureka* angelehnt. *THEIA* unterstützt unterschiedliche Heuristiken, in dieser Untersuchung wird die *sum*-Heuristik verwendet. Es wurde ein Timeout von 600 Sekunden gewählt. Instanzen, die beim Preprocessing zu einem Timeout oder zu einem Programmabsturz eines Solver führten, wurden bei der Betrachtung entfernt. In der Abbildung 20 wird die Gesamtlaufzeit der beiden Solver präsentiert sowie eine Gegenüberstellung der beiden Solver hinsichtlich ihrer Laufzeiten mit und ohne vereinfachten AFs. In Abbildung 23 (siehe Anhang) findet sich zusätzlich eine Gegenüberstellung der Laufzeiten mit und ohne Preprocessing für die einzelnen Solver.

Die Gesamtlaufzeit von *μ-toksia* ist mit und ohne Preprocessing insgesamt kürzer im Vergleich zu *THEIA*. Durch das Vereinfachen kann für den Solver *μ-toksia* keine kürzere Laufzeit erzielt werden. Der Solver *THEIA* hingegen profitiert durch das Preprocessing, die Laufzeit verringert sich um 3%. Jedoch haben manche Instanzen, die vorher eine Laufzeit um die 0,01 Sekunden hatten, durch den Overhead des Preprocessing nun eine längere Laufzeit. Für das *ABA2AF* Modell hat sich die Laufzeit angeglichen. Dabei hat *μ-toksia* nach dem Preprocessing eine kürzere Laufzeit und *THEIA* eine längere Laufzeit (vgl. Abb. 23). μ-toksia berechnet die Extensionen für die Modelle *Scc-Generator*, *Watts-Strogatz* und *StableGenerator* in der Regel schneller als *THEIA*. Dagegen ist *THEIA* schneller bei den Modellen *Traffic*, *Plan2AF* und *Barabasi-Albert*. Überdies verkürzt sich die Laufzeit für diese Modelle unter *THEIA* durch das Preprocessing leicht. Wie schon bei der *DC-ST* Aufgabe ist das Preprocessing von *GroundedGenerator* Modellen auch hier von Nachteil. Für die gewählte Zeitbeschränkung von 600 Sekunden liefen circa 20% der Berechnungen für *μ-toksia* und 23% für *THEIA* in einen Timeout. Durch das

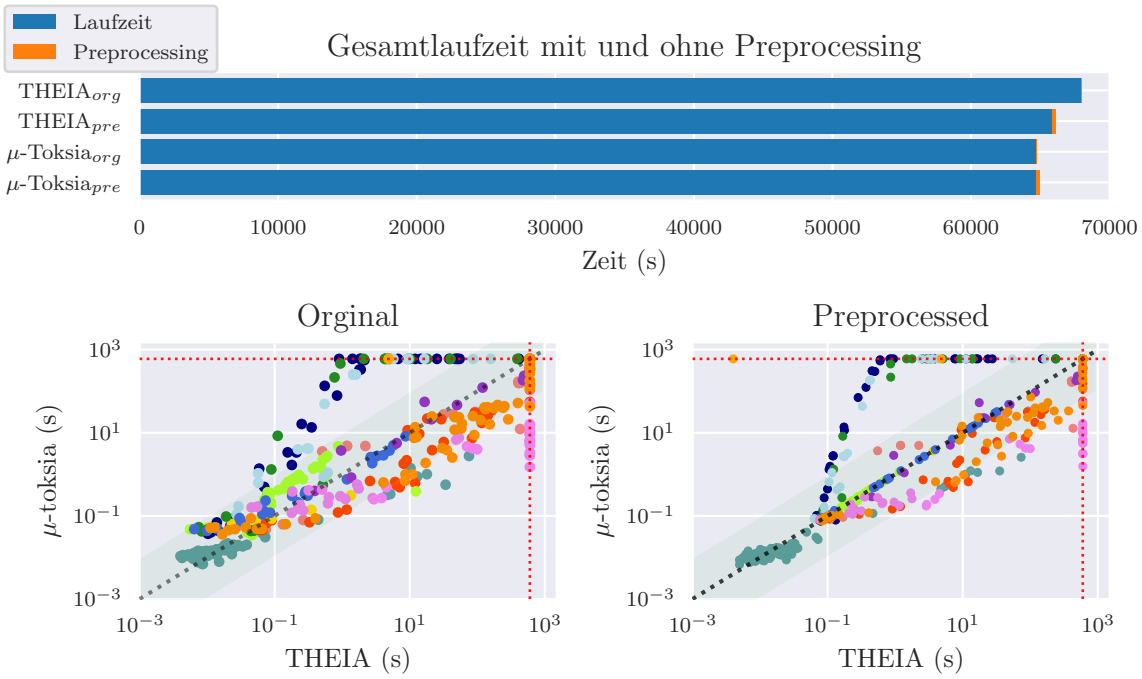


Abbildung 20: EE-CO Aufgabe für die Datensätze der ICCMA 2017 und ICCMA 2019 mit einem Timeout von 600 Sekunden.
Farbzuordnung in den Scatterplots: ABA2AF: ●, Admbuster: ●, AFGen ●, Barabasi-A.: ●, Datalog ●, Erdös-Renyi: ●, GrdGen.: ●, Plan2AF: ●, SccGen.: ●, SemBuster: ●, StableGen.: ●, Traffic: ●, Watts-St.: ●

Preprocessing konnte der Wert für *μ-toksia* um 0,5% und für *THEIA* um 1% verringert werden.

Zusammenfassend lässt sich festhalten, dass sich das Preprocessing für die EE-CO Aufgabe nur positiv auf die Laufzeit des Solvers *THEIA* auswirkt. Für *μ-toksia* konnten nur einzelne Modelle, wie bspw. *ABA2AF*, beschleunigt werden.

Auffallend war, dass die Solver im direkten Vergleich unterschiedlich gut hinsichtlich der verschiedenen Modelle performen. Unter diesem Blickwinkel könnte zukünftig die Funktionweise des *A-folio DPDB* Solvers interessant sein, der basierend auf einer Aufgabenstellung diese an einen Solver delegiert. Dies könnte man weiter entwickeln und das Modell eines AFs als Entscheidungskriterium hinzufügen. Für diesen Gedankengang muss untersucht werden, ob ein Modell bspw. durch die Struktureigenschaften eines AFs klassifiziert werden kann.

7 Muster-Exploration

In diesem Abschnitt wird der Frage nachgegangen, ob neben den bereits bekannten Mustern noch weitere Muster existieren können. Dazu wird den beiden folgenden Fragestellungen nachgegangen:

- Können noch weitere Argumente entfernt werden?
- Wie gut werden AFs, die über keine Extension verfügen, erkannt?

Für die erste Fragestellung wird exemplarisch die Idee des *3-Kegel*-Musters erläutert. Beim Transformationsschritt wird ein Argument entfernt, da dieses in keiner Extension enthalten sein kann (*sceptical rejected*). Genau diese Eigenschaft ist Untersuchungsgegenstand der ersten Fragestellung. Die zweite Fragestellung beschäftigt sich mit dem Fehlen einer Extension und inwieweit das unattackierte *3-Kreis*-Muster in solchen AFs enthalten ist.

Es werden erneut die Datensätze der *ICCMA 2017* und *ICCMA 2019* herangezogen und im Kontext der *stabilen* Semantik untersucht. Die AFs werden dazu durch das Programm *AFPreprocessing* vereinfacht. Es werden alle Muster, einschließlich dem *3-zu-2*-Muster angewendet. Anschließend werden alle Extensionen durch *μ-toksia* mit einem Timeout von 600 Sekunden ermittelt. Die AFs, deren Berechnungen in einen Timeout liefen, werden nicht weiter betrachtet. Die Argumente, die Teil einer Extension sind, werden in einer Extensionsmenge festgehalten. Die Differenz zwischen den Argumenten eines AFs mit der Extensionsmenge ergibt die Menge der Argumente, die *sceptical rejected* sind.

Es wird als erstes die Untersuchung über *sceptical rejected* (*sr*) Argumente betrachtet. In Abbildung 21 ist im unten links befindlichen Diagramm die Verteilung der Anzahl von *sceptical rejected* Argumenten dargestellt. Im unteren rechten Diagramm findet sich Verteilung der durchschnittlichen Angriffe auf die *sceptical rejected* Argumente.

In den Diagrammen fehlen die Modelle *Admbuster*, *Barabasi-Albert*, *Datalog* und *GroundedGenerator*, da keine weiteren *sr*-Argumente gefunden wurden. Das Modell *SccGenerator* fehlt ebenfalls, weil alle AFs des Modells keine stabile Extension besitzen. Im Umkehrschluss lassen sich in AFs von acht unterschiedlichen Modellen *sr*-Argumente finden. Am auffälligsten sind die Ergebnisse für das *Sembuster* Modell. Die AFs enthalten die meisten *sr*-Argumente und diese besitzen die höchste Anzahl an Angreifern. Wie bereits in Kapitel 5.1 ausgeführt, sind die Instanzen des *Sembuster* Modells sehr schematisch aufgebaut. Dieser Aspekt macht eine Untersuchung der Existenz weiterer Muster interessant.

Von den übrigen Modellen sind in dem *StableGenerator* und *Erdös-Renyi* Modell die meisten *sr*-Argumente zu finden, wobei der Unterschied zu den Modellen *AFGen* und *Watts-Strogatz* marginal ist. Bei der Betrachtung der Anzahl der Angreifer besitzen die *sr*-Argumente der Modelle *ABA2AF*, *AFGen*

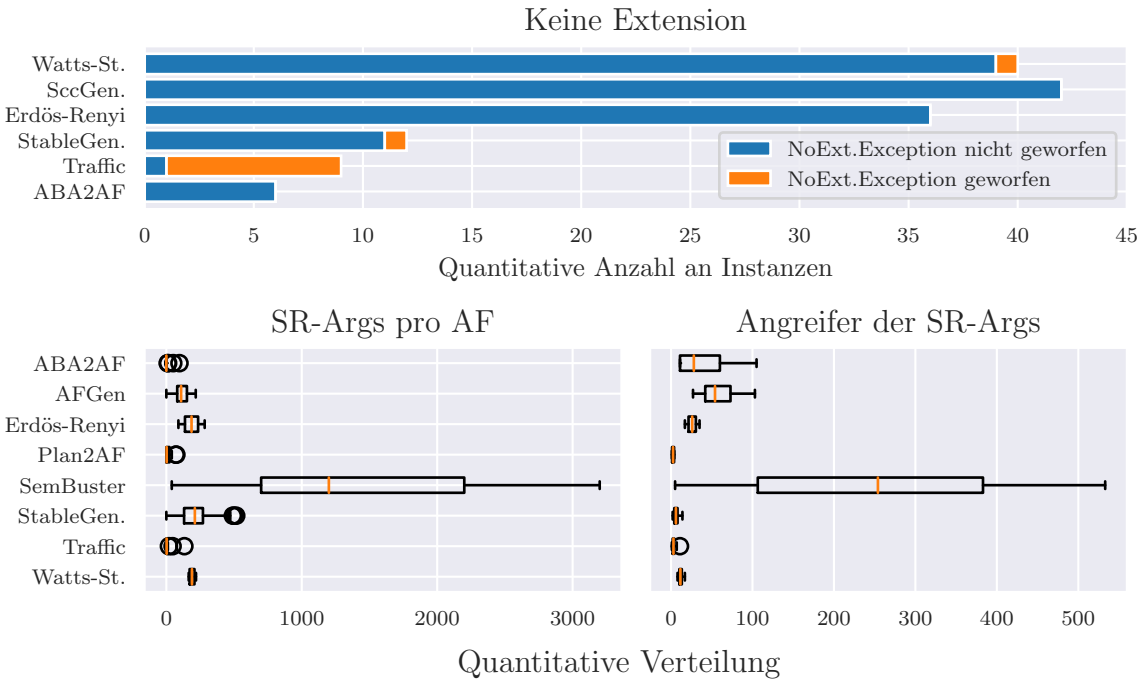


Abbildung 21: Mustersuche: Oben Alle Instanzen die keine stabile Extension besitzen - Blau: unentdeckt, Orange: entdeckt.
Unten links: Verteilung der *sceptical rejected* (*sr*) Argumente,
Unten rechts: Verteilung der Angreifer der *sr*-Argumente

und *Erdös-Renyi* erkennbar mehr Angreifer als die *sr*-Argumente der restlichen Modelle.

Nun wird die Erkennung der Abwesenheit einer *stabilen* Extension überprüft (vgl. Abb. 21, oberes Diagramm). Zur Erinnerung: das Programm *AFPreprocessing* wirft für die *stabile* Semantik eine Ausnahmebedingung, falls es ein unattackiertes *3-Kreis*-Muster identifiziert. Durch diese Methodik werden knapp 7% der AFs, die über keine Extension verfügen, entdeckt. Die Identifzierung war für das *Traffic* Modell am erfolgreichsten, da nur eine Instanz nicht die Ausnahmebedingung ausgelöst hat. Alle AFs des *SccGenerator* Modells besitzen keine Extension. Der Aufbau ist ebenfalls wie das *Sembuster* Modell relativ schematisch, da starke Zusammenhangskomponenten gebildet und anschließend leicht untereinander verbunden werden. Unter diesem Aspekt kann eine genauere Untersuchung hinsichtlich eines Musters, das die Abwesenheit einer *stabilen* Extension erkennt, erfolgversprechend sein. Im Rahmen der probabilistischen Modelle *Erdös-Renyi* und *Watts-Strogatz* besitzen eine signifikante Anzahl von AFs ebenfalls keine Extension. Durch den zufallsgenerierten Aufbau ist es hierbei wesentlich schwieriger Kandidaten hinsichtlich potenzieller Muster zu identifzieren, die zu keiner Extension führen. Das Modell *ABA2AF* konnte auf durchschnittlich acht Argumente verkleinert werden

(vgl. Tabelle 5). Damit wäre es tendenziell möglich, alle Angriffsstrukturen mit relativ geringem Ressourcenaufwand zu analysieren.

Zusammenfassend lässt sich festhalten, dass die Preprocessing-Methode für bspw. die hier betrachteten *Barabasi-Albert* AFs alle *sr*-Argumente entfernt. Es lassen sich allerdings in einigen Modellen noch solche Argumente finden und es bedarf einer genaueren Untersuchung, ob weitere Ersetzungsmuster aus den Angriffsstrukturen der *sr*-Argumente ableitbar sind. Dies gilt analog für die Existenz einer Extension.

8 Fazit und Ausblick

Abschließend werden in diesem Kapitel weitere Arbeiten, die sich mit der Thematik des Preprocessings beschäftigt haben, zusammengetragen und die Ergebnisse dieser Arbeit zusammengefasst.

Related Works

Ein erster Ansatz für die Verwendung von Mustern in AFs findet sich in der Arbeit [4]. In dieser wurden *Multipoles* vorgestellt, die aus einer Menge von Argumenten mit einer wohldefinierten Schnittstelle bestehen. Diese Art von modularen Komponenten erlauben eine Ersetzung durch andere Komponenten und bilden ein erstes Konzept für Ersetzungsmuster. Eine konkrete Ausarbeitung von Ersetzungsmustern wurde allerdings nicht entwickelt.

In [39] wurden Argumente zu Clustern zusammengefügt, falls diese gewisse Eigenschaften untereinander teilen. Diese Methodik soll bspw. bei der Visualisierung von AFs unterstützen. Diese Herangehensweise ähnelt der hier verwendeten Zusammenfügungsoperation. Allerdings gehen durch den Clustering-Prozess Informationen verloren, sodass die klassischen Semantiken für den geclusterten AF nicht gelten. Zusätzlich werden im Rahmen des Ansatzes keine Argumente entfernt, wie dies hier durch die *sceptical rejected* Argumente verfolgt wurde.

Mit weiteren Semantiken sind potenziell neue Muster anwendbar, die einen AF vereinfachen, wie dies bereits in [8] skizziert wurde. Bei diesem Ansatz wurde die *schwach-zulässige* Semantik eingeführt, die es erlaubt selbst-angreifende Argumente zu entfernen. Dementsprechend kann der Transformationsschritt für das *3-Kreis*-Muster unter dieser Semantik angepasst werden.

Die momentan als State-of-the-Art-Solver angesehenen Solver verwenden SAT-Solver. Bei SAT-Solvern sind Preprocessing-Methoden ein wichtiger Vorgang, um Berechnungen zu beschleunigen. In [11] findet sich eine Übersicht über entwickelte Preprocessing-Methoden im Bereich von SAT-Solvern. Da Probleme aus unterschiedlichen Domänen in ein SAT-Problem überführt werden können, kann das SAT-Preprocessing generischer angesehen werden, als die hier verwendete Preprocessing-Methodik. Ein interessanter Untersuchungs-

gegenstand wäre, inwieweit sich die Ziele des SAT-Preprocessings, mit bspw. der *Literal*-Eliminierung und den hier verwendeten Eliminierungsoperationen überschneiden.

Fazit

Ziel dieser Arbeit war die Entwicklung einer effizienten Preprocessing-Methode für die Verkleinerung von AFs. Um dieses Ziel zu erreichen, wurde auf der theoretischen Ebene die Anwendung der Musterreihenfolge untersucht. Es konnte bewiesen werden, dass für die *vollständigen* und *präferierten* Semantiken eine beliebige Anwendungsreihenfolge von Mustern keine Auswirkung auf das Endergebnis entfaltet. Gleiches gilt grundsätzlich auch für die *stabile* Semantik, jedoch mit der Ausnahme eines Überschneidungsszenarios, das zu einem unterschiedlichen Ergebnis führt. Auf der praktischen Ebene wurden die Datensätze der *ICCMA* aus den Jahren 2017 und 2019 genauer studiert, um Einschränkungen für den Suchraum abzuleiten. Die daraus resultierende Implementierung hat den ursprünglichen Prototyp *AFClingo* in allen Testinstanzen hinsichtlich Performance übertroffen. Da der Fokus auf einer effizienten Suchstrategie lag, besteht beim Transformieren weiterhin noch Verbesserungsbedarf wie die Instanzen des *GroundedGenerator* Modells gezeigt haben.

Mittels der Preprocessing-Methodik können einige Modelle signifikant verkleinert werden. Diese Methodik eignet sich allerdings nicht für alle Modelle, die in den *ICCMA* Datensätzen vorkommen. Auch können nicht alle Solver von diesem Verfahren profitieren. Die redukionsbasierten Solver, die momentan als die State-of-the-Art Solver angesehen werden, profitieren auf Grund des Overheads nicht. Bei den direkten Ansätzen sieht das Ergebnis anders aus. Das Entwickeln einer Kombination des direkten Ansatzes mit der Preprocessing-Methodik ist ein Forschungsfeld, das sich für weitere Arbeiten anbieten würde.

Zudem wurde die Suche nach weiteren Mustern in Anfängen untersucht. Im Vordergrund standen dabei die *sceptical rejected* Argumente. Ein Ergebnis zeigte auf, dass z.B. für die hier betrachteten *Barabasi-Albert*-AFs alle eliminierbaren Argumente auch entfernt werden. Bei anderen Modellen lassen sich auch nach der Verkleinerung noch solche Argumente finden. Es gilt im Detail zu untersuchen, ob sich hier noch weitere Muster verbergen. Neben dieser Art von Argumenten bilden die Muster, die zu keiner Extension führen, einen interessanten Untersuchungsgegenstand. Mit dem *3-Kreis*-Muster für die *stabile* Semantik hat die Heuristik bereits ein solches Muster implementiert. Es gilt aber noch zu untersuchen, ob weitere Muster existieren.

Wenngleich die momentanen State-of-the-Art Solver nicht beschleunigt werden konnten, bleibt der Ansatz des Vereinfachens von AFs weiterhin ein interessantes Forschungsgebiet, da auch neue Semantiken entwickelt werden können und hiermit eventuell neue Muster einhergehen.

Anhang

Orginal	Abkürzung
ABA2AF	ABA2AF
AFGen	AFGen
Admbuster	Admbuster
Barabasi-Albert	Barabasi-A.
Datalog	Datalog
Erdös-Renyi	Erdös-Renyi
GroundedGenerator	GrdGen.
Planning2AF	Plan2AF
SccGenerator	SccGen.
SemBuster	SemBuster
StableGenerator	StableGen.
Traffic	Traffic
Watts-Strogatz	Watts-St.

Tabelle 6: Modell-Abkürzungen

Anzahl der Angreifer	Vorkommen in %
1	31,1057
2	68,8348
3	0,0024
4	0,0010
5	0,0008
...	...
1191	0,00007
1194	0,00007

Tabelle 7: Anzahl der Angreifer von 2-zu-1 Muster bei der Initialsuche in Prozent

	σ_{com}		σ_{pre}	
Modell	$\varnothing Arg._{pre}$	Reduzierung (%)	$\varnothing Arg._{pre}$	Reduzierung (%)
ABA2AF	8,7674	98,0445	8,7209	98,0548
Admbuster	1,0	99,9920	1,0	99,9920
AFGen	189,6	0,0	189,6	0,0
Barabasi-Albert	33,0	73,9684	33,0	73,9684
Datalog:	69,0892	9,1673	69,0892	9,1677
Erdös-Renyi	366,5	0,0	366,5	0,0
GroundedGenerator	1,0800	99,9798	1,0800	99,9798
Planning2AF	77,0	78,9795	76,7241	79,0548
SccGenerator	3981,8888	0,0	3981,8888	0,0
SemBuster	2889,2307	0,0	2889,2307	0,0
StableGenerator	355,8035	20,1977	355,6875	20,2047
Traffic	542,3508	55.5577	538,4736	55,8755
Watts-Strogatz	369,5094	2,5671	369,5094	2,5671

Tabelle 8: Durchschnittliche Anzahl der Argumente nach dem Verkleinern für die *vollständigen* und *präferierten* Semantiken.

Algorithmus	Modell	Anzahl der Timeouts
Heuristik	Admbuster	4
	GroundedGenerator	4
Prototype	Admbuster	8
	Erdös-Renyi	24
	GroundedGenerator	22
	SccGenerator	32
	SemBuster	11
	Traffic	5

Tabelle 9: Anzahl der Timeouts für beide Algorithmen differenziert nach Modell

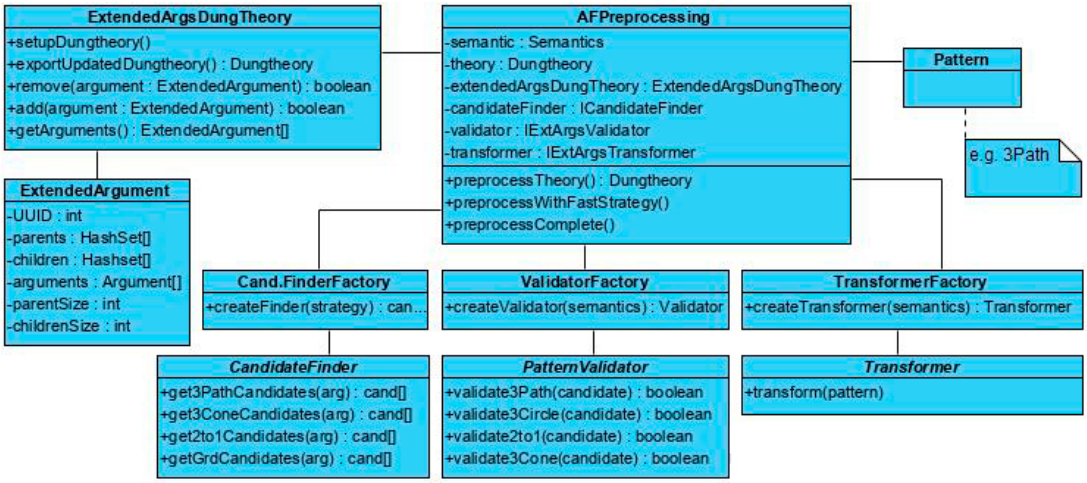


Abbildung 22: Abstrahiertes UML-Diagramm von *AFPreprocessing*

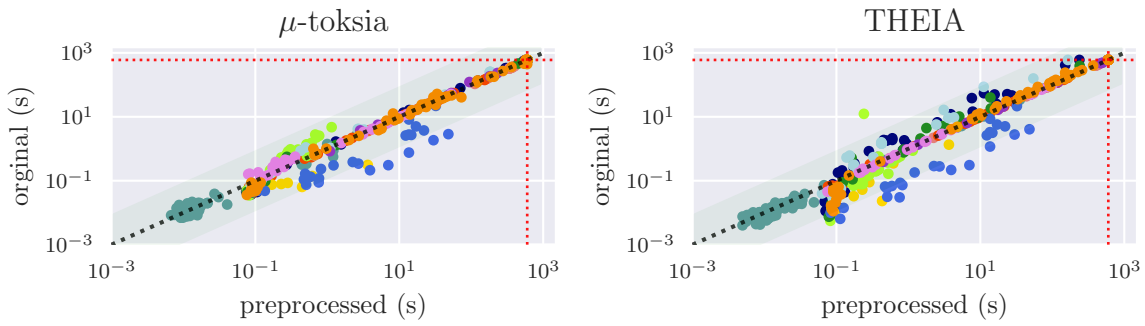


Abbildung 23: Lauzeitanalyse der Solver im Detail für die EE-CO Aufgabe mit einem Timeout von 300 Sekunden für die Datensätze der *ICCMA 2017* und *ICCMA 2019*.
Farbzuordnung in den Scatterplots: Farbzuordnung in den Scatterplots: ABA2AF: ●, Admbuster: ●, AFGen ●, Barabasi-A.: ●, Datalog ●,Erdös-Renyi: ●, GrdGen.: ●, Plan2AF: ●, SccGen.: ●, SemBuster: ●, StableGen.: ●, Traffic: ●, Watts-St.: ●

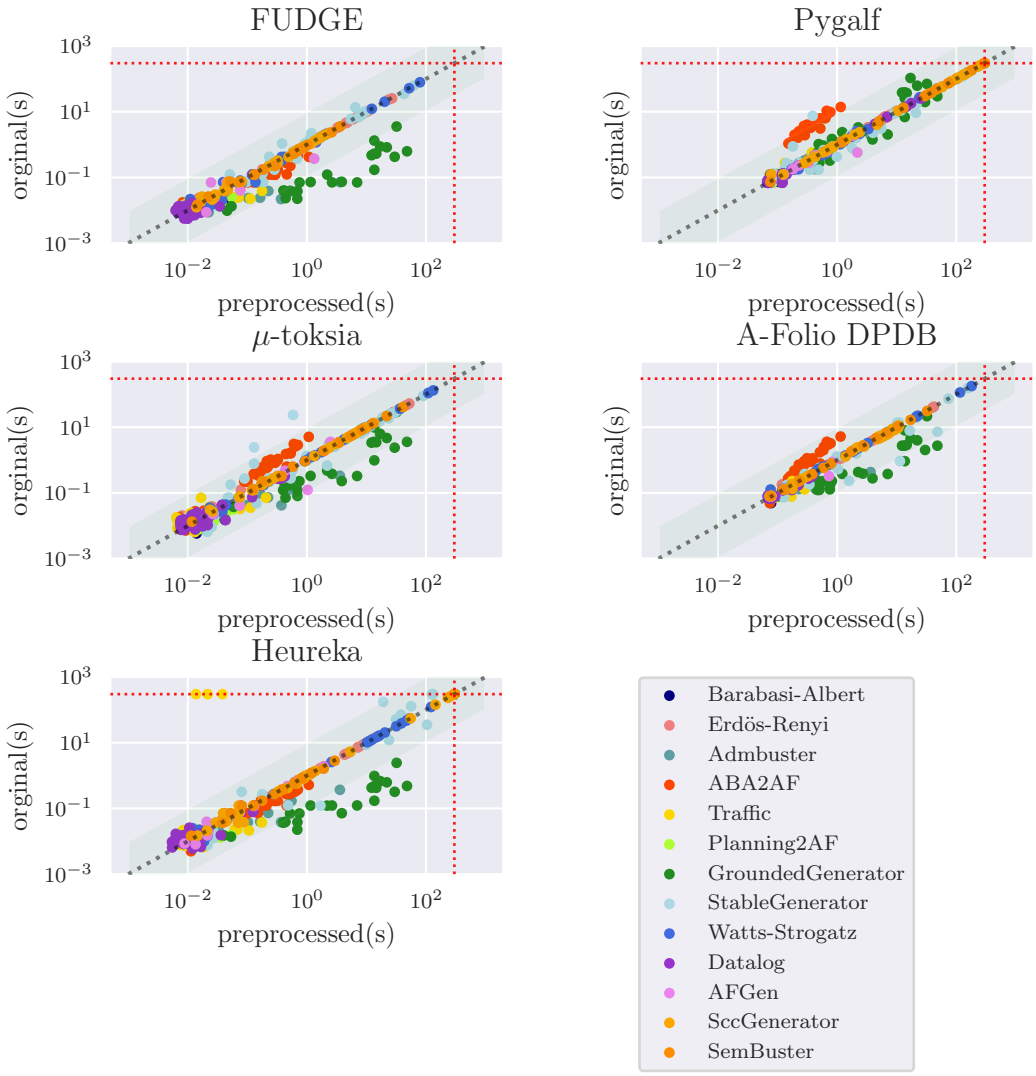


Abbildung 24: Lauzeitanalyse der Solver im Detail für die DC-ST Aufgabe mit einem Timeout von 300 Sekunden für die Datensätze der *ICCMA 2017* und *ICCMA 2019*

Literatur

[1] Mario Alviano. The pyglaf argumentation reasoner (ICCMA2021). *CoRR*, abs/2109.03162, 2021.

[2] Gilles Audemard and Laurent Simon. On the glucose SAT solver. *Int. J. Artif. Intell. Tools*, 27(1):1840001:1–1840001:25, 2018.

[3] Albert-Laszlo Barabasi and Reka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[4] Pietro Baroni, Guido Boella, Federico Cerutti, Massimiliano Giacomin, Leendert W. N. van der Torre, and Serena Villata. On the input/output behavior of argumentation frameworks. *Artif. Intell.*, 217:144–197, 2014.

[5] Pietro Baroni, Martin Caminada, and Massimiliano Giacomin. Abstract argumentation frameworks and their semantics. In *Handbook of Formal Argumentation*, pages 159–232. College Publications, 2018.

[6] Pietro Baroni, Francesca Toni, and Bart Verheij. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games: 25 years later. *Argument Comput.*, 11(1-2):1–14, 2020.

[7] Ringo Baumann. Splitting an argumentation framework. In James P. Delgrande and Wolfgang Faber, editors, *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, volume 6645 of *Lecture Notes in Computer Science*, pages 40–53. Springer, 2011.

[8] Ringo Baumann, Gerhard Brewka, and Markus Ulbricht. Revisiting the foundations of abstract argumentation - semantics based on weak admissibility and weak defense. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 2742–2749. AAAI Press, 2020.

[9] Ringo Baumann, Wolfgang Dvorák, Thomas Linsbichler, and Stefan Woltran. A general notion of equivalence for abstract argumentation. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 800–806. ijcai.org, August 2017.

[10] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the

SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[11] Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in SAT solving. In *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 391–435. IOS Press, 2021.

[12] Stefano Bistarelli, Lars Kotthoff, Francesco Santini, and Carlo Taticchi. A first overview of iccma'19. In *Proceedings of the Workshop on Advances In Argumentation In Artificial Intelligence 2020 co-located with the 19th International Conference of the Italian Association for Artificial Intelligence (AIxIA 2020), Online, November 25-26, 2020*, volume 2777 of *CEUR Workshop Proceedings*, pages 90–102. CEUR-WS.org, 2020.

[13] Martin Caminada and Mikolaj Podlaszewski. Admbuster: a benchmark example for (strong) admissibility. `https://argumentationcompetition.org/2017/AdmBuster.pdf`, 2017.

[14] Martin Caminada and Bart Verheij. Sembuster: a benchmark example for semi-stable semantics. `https://users.cs.cf.ac.uk/CaminadaM/publications/sembuster.pdf`, 2017.

[15] Federico Cerutti, Sarah A. Gaggl, Matthias Thimm, and Johannes P. Wallner. Foundations of implementations for formal argumenation. In *Handbook of Formal Argumentation*, pages 689–767. College Publications, 2018.

[16] Federico Cerutti, Massimiliano Giacomin, and Mauro Vallati. Exploiting planning problems for generating challenging abstract argumentation frameworks. `https://argumentationcompetition.org/2017/Planning2AF.pdf`, 2017.

[17] Federico Cerutti, Nir Oren, Hannes Strass, Matthias Thimm, and Mauro Vallati. A benchmark framework for a computational argumentation competition. In Simon Parsons, Nir Oren, Chris Reed, and Federico Cerutti, editors, *Computational Models of Argument - Proceedings of COMMA 2014, Atholl Palace Hotel, Scottish Highlands, UK, September 9-12, 2014*, volume 266 of *Frontiers in Artificial Intelligence and Applications*, pages 459–460. IOS Press, 2014.

[18] Federico Cerutti, Ilias Tachmazidis, Mauro Vallati, Sotirios Batsakis, Massimiliano Giacomin, and Grigoris Antoniou. Exploiting parallelism for hard problems in abstract argumentation. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 1475–1481. AAAI Press, 2015.

[19] Federico Cerutti, Mauro Vallati, and Massimiliano Giacomin. A generator for random argumentation frameworks. `https://argumentationcompetition.org/2017/AFBenchGen2.pdf`, 2017.

[20] Martin Diller. Traffic networks become argumentation frameworks. http://argumentationcompetition.org/2017/Traffic.pdf, 2017.

[21] Sylvie Doutre, Mickael Lafages, and Marie-Christine Lagasquie-Schiex. A distributed and clustering-based algorithm for the enumeration problem in abstract argumentation. In *Principles and Practice of Multi-Agent Systems - 22nd International Conference, Turin, Italy, October 28-31, 2019, Proceedings*, volume 11873, 2019.

[22] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial intelligence*, 77(2):321–358, 9 1995.

[23] Wolfgang Dvorák and Paul E. Dunne. Computational problems in formal argumenation and their complexity. In *Handbook of Formal Argumentation*, number 631-687. College Publications, 2018.

[24] Wolfgang Dvorák, Matti Järvisalo, Thomas Linsbichler, Andreas Niskanen, and Stefan Woltran. Preprocessing argumentation frameworks via replacement patterns. *Logics in Artificial Intelligence - 16th European Conference, 2019, Rende, Italy, May 7-11, 2019, Proceedings*, 11468:116–132, 2019.

[25] Wolfgang Dvorák, Matthias König, Johannes Peter Wallner, and Stefan Woltran. Aspartix-V21. *CoRR*, abs/2109.03166, 2021.

[26] P. Erdös and A. Rényi. On random graphs i. *Publicationes Mathematicae Debrecen*, 6, 1959.

[27] Johannes K. Fichte, Markus Hecher, Piotr Gorczyca, and Ridhwan Dewoprabowo. A-FOLIO DPDB – System description for ICCMA 2021. 2021.

[28] Sarah A. Gaggl, Thomas Linsbichler, Marco Maratea, and Stefan Woltran. Benchmark selection ICCMA'17. `http://argumentationcompetition.org/2017/benchmark_selection_iccma2017.pdf`, 2017.

[29] Yong Gao. A random model for argumentation framework: Phase transitions, empirical hardness, and heuristics. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 503–509. ijcai.org, 2017.

[30] 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)*, volume 52 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:15, 2016.

[31] Nils Geilen and Matthias Thimm. Heureka: A general heuristic backtracking solver for abstract argumentation. In Elizabeth Black, Sanjay Modgil, and Nir Oren, editors, *Theory and Applications of Formal Argumentation - 4th International Workshop, TAFA 2017, Melbourne, VIC, Australia, August 19-20, 2017, Revised Selected Papers*, volume 10757 of *Lecture Notes in Computer Science*, pages 143–149. Springer, 2017.

[32] Lukas Kinder, Matthias Thimm, and Bart Verheij. A labelling-based solver for computing complete extensions of abstract argumentation frameworks. In *Proceedings of the Fourth International Workshop on Systems and Algorithms for Formal Argumentation (SAFA'22)*, September 2022.

[33] Jonas Klein and Matthias Thimm. probo2: A benchmark framework for argumentation solvers. In *Computational Models of Argument - Proceedings of COMMA 2022, Cardiff, Wales, UK, 14-16 September 2022*, volume 353 of *Frontiers in Artificial Intelligence and Applications*, pages 363–364, 2022.

[34] Markus Kröll, Reinhard Pichler, and Stefan Woltran. On the complexity of enumerating the extensions of abstract argumentation frameworks. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 1145–1152. ijcai.org, 2017.

[35] Jean-Marie Lagniez, Emmanuel Lonca, Jean-Guy Mailly, and Julien Rossit. Design and Results of ICCMA 2021. *CoRR*, abs/2109.08884, 2021.

[36] Tuomo Lehtonen, Johannes P. Wallner, and Matti Jarvisalo. Assumption-based argumentation translated to argumentation frameworks. `https://argumentationcompetition.org/2017/ABA2AF.pdf`, 2017.

[37] Andreas Niskanen and Matti Järvisalo. μ-toksia: An efficient abstract argumentation reasoner. In Diego Calvanese, Esra Erdem, and Michael Thielscher, editors, *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece, September 12-18, 2020*, pages 800–804, 2020.

[38] Samer Nofal, Katie Atkinson, and Paul E. Dunne. ArgTools: a labelling-based solver for abstract argumentation. `https://argumentationcompetition.org/2017/ArgTools.pdf`, 2017.

[39] Zeynep Gözen Saribatur and Johannes Peter Wallner. Existential abstraction on argumentation frameworks via clustering. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021, Online event, November 3-12, 2021*, pages 549–559, 2021.

[40] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257, 2009.

[41] Billy Spelchan and Gao Yong. The afgen argumentation benchmark generator. `http://argumentationcompetition.org/2019/papers/ICCMA19_paper_3.pdf`, 2019.

[42] Matthias Thimm. Tweety: A comprehensive collection of java libraries for logical aspects of artificial intelligence and knowledge representation. In Chitta Baral, Giuseppe De Giacomo, and Thomas Eiter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference*. AAAI Press, July 2014.

[43] Matthias Thimm, Federico Cerutti, and Mauro Vallati. FUDGE: A lightweight solver for abstract argumentation based on sat reductions. *CoRR*, abs/2109.03106, 2021.

[44] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[45] Bruno Yun and Madalina Croitoru. Benchmark on logic-based argumentation framework with datalog$^{\pm}$. `http://argumentationcompetition.org/2019/papers/ICCMA19_paper_2.pdf`, 2019.

[46] Bruno Yun, Madalina Croitoru, Srdjan Vesic, and Pierre Bisquert. DAGGER: datalog+/- argumentation graph generator. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018*, pages 1841–1843. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM, 2018.