Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

An Implementation of Splitting for Dung Style Argumentation Frameworks

Bachelorarbeit

Leipzig, Mai 2011

vorgelegt von Renata Wong
Studiengang Theoretische Informatik

Betreuender Hochschullehrer: Prof. Dr. Gerhard Brewka
Institut für Informatik, Abteilung Intelligente Systeme

Contents

Chapter 1

Introduction

Argumentation and reasoning have been an area of research in such disciplines as philosophy, logic and artificial intelligence for quite some time now. In the area of AI, knowledge needed for reasoning can be represented using different kinds of representation systems. The natural problem posed by this fact is that of possible incompatibility between heterogeneous systems as far as communication between them is concerned. This imposes a limitation on the possibility of extending smaller knowledge bases to larger ones. In order to facilitate a common platform for exchange across the systems unified formalisms for the different approaches to knowledge representation are required.

This was the motivation for Dung [11] to propose in his 1995 paper an approach that later came to be known as an *abstract argumentation framework*. Roughly speaking, Dung's arguments are abstract entities which are related to each other by the means of conflicts between them. An intuitive graphical representation of Dung style framework is a graph whose nodes stand for arguments and whose edges stand for conflicts.

A framework postulated this way is on one hand too general to be used on its own, but on the other hand it is general enough as to allow for varied extensions increasing its expressiveness, which indeed have been proposed. They include value-based argumentation frameworks by Bench-Capon et al. [6], preference-based argumentation frameworks by Amgoud and Cayrol [1] and bipolar argumentation frameworks by Brewka and Woltran [7], to name a few.

The present thesis is concerned with yet another variation of Dung's framework: the concept of splitting. It was developed by Baumann [4] with one of the underlying purposes being that the computation time in frameworks which have been split into two parts and then computed separately may show some improvement in comparison to their variant without splitting. It was one of the main tasks of my work to develop an efficient algorithm for the splitting operation based on the theoretical framework given in [4]. On the other hand I hoped to confirm the expectation that splitting can indeed make a computation perform better.

In order to confirm (or disprove) the expectation of an improvement in runtime if splitting is applied to a framework I designed and developed an application which is the main topic of this thesis. The present work is structured as follows. Chapter 2 describes briefly the theoretical foundations of argumentation frameworks and argumentation se-

mantics. Chapter 3 gives the required introduction to the field of splitting. Chapter 4 deals with the different aspects of the developed software. I will describe the graphical user interface as well as the functionality offered by it. Here belongs also the implementation of various algorithms, i.e. three algorithms borrowed from Modgil and Caminada [15] used for computation of labellings, Tarjan's algorithm for computing of strongly connected components as well as an algorithm for splitting which I developed based on the procedures described in [4]. Some general internal structure of the application is also given. Next, Chapter 5 presents the results of an experimental performance evaluation made on a total of 100 randomly generated frameworks. And finally, Chapter 6 concludes.

Remark As mentioned above, the present thesis is based on and accompanies an application which I developed as partial requirement for the Bachelor's degree. The software - a .jar executable plus a folder containing 3rd party libraries - is available upon request. A DVD-ROM containing application, libraries, Java source files and Javadoc documentation is attached with the paper version of the thesis.

Remark In process of the experimental evaluation presented in Chapter 5 100 files containing framework data were created. They can be read by any .net extension (Pajek) supporting software. These are also available upon request.

Remark All images representing argumentation frameworks given in the thesis were created using the built-in image export function of the developed application. The two diagrams in Chapter 4 were made with the open source software Dia. All screen shots were made with the Windows Print Screen Key.

Chapter 2

Argumentation Frameworks

2.1 Theoretical Foundations

In this section I will provide some basic definitions and reasoning on which the entire thesis is grounded. First of all, we need to start with the notion of an argumentation framework as defined by Dung in [11]:

Definition 2.1 *An argumentation framework* $AF = (A, R)$ *is a pair where* A *is a set of arguments, and* $R \subseteq (A \times A)$ *is a binary relation on* A *called attack relation.*

It is quite intuitive to represent an argumentation framework as a directed graph in which nodes correspond to arguments and edges constitute the attack relation. For convenience, I will refer to the latter simply as *attacks*. Fig. 2.1 shows a simple argumentation framework with two arguments (0 and 1) and an attack from 0 to 1. In other words our framework can be formally written as $AF_{2.1} = (\{0, 1\}, \{(0, 1)\})$.

Fig. 2.1: $AF_{2.1}$

The above example is very general and may therefore represent many different situations. We can put it into a specific context by assigning propositions to its arguments. Given that we currently experience a wave of popular anti-governmental uprisings in North Africa and the Middle East, argument 1 could mean: "The Libyan protests of 2011 are peaceful popular uprising.", and argument 0 could mean: "The Libyan protests are neither peaceful nor popular because on footages we can see only a fistful of militants with heavy weaponry.". In our discourse argument 0 attacks argument 1 as no peaceful protester would carry an anti-aircraft rocket launcher on his or her shoulder.

Having defined the notion of an argumentation framework, we can see that not every argument can be accepted by a rational agent. Rather, the status of each argument is

subject to evaluation. It is understandable that not every set of arbitrarily chosen arguments can be seen as justified. In $AF_{2.1}$, we either accept argument 0 or argument 1. We cannot hold both statements as true at the same time because we cannot believe that a protest is both peaceful and violent. We have also to keep in mind that our information may be incomplete and/or uncertain. There could be some other arguments that could contribute to our discourse, but so far we have to deal with what there is, i.e. the arguments 0 and 1.

2.2 Acceptability of Arguments

Undeniably, one of the main concerns regarding argumentation are the conditions under which an argument can be accepted by a reasoning agent. The most general specification of a set of arguments that can be accepted is given by the notion of a *conflict-free* set. It is outlined in the following definition:

Definition 2.2 *Given an argumentation framework $AF = (A, R)$, a set $S \subseteq A$ is conflict-free, iff $\nexists a, b \in S$ s.t. $(a, b) \in R$.*

In other words, given any two arguments, one being a source and the other being a target of an attack in R, we cannot accept them both at the same time. In $AF_{2.1}$ there exist three *conflict-free* sets: $\varnothing$, $\{0\}$ and $\{1\}$. We can either accept the argument that the Libyan uprising is peaceful and popular or that it is neither peaceful nor popular. We have also the option of accepting neither of the two arguments. However, if our framework were looking like the one in Fig. 2.2, there would exist six *conflict-free* sets: $\varnothing$, $\{0\}$, $\{2\}$, $\{3\}$, $\{0, 2\}$, $\{2, 3\}$.

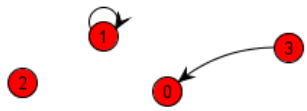

Fig. 2.2: $AF_{2.2}$

Further, it is reasonable for a rational agent to require that a set of arguments fulfills additional conditions in order to be accepted. The most basic of them being that every member of the set is also defended by it. By defense we mean that whenever an argument is a target of an attack the agent is able to counterattack the attacker. In $AF_{2.1}$ it seems quite unnatural to accept the statement that the Libyan protests are peaceful and popular (argument 1) since it is attacked by another statement (argument 0), and is not defended by any other argument. In [11] the requirement is formally given in two steps by defining first the notion of *acceptability* of an argument, and then on its basis defining the notion of *admissibility* of a set:

Definition 2.3 *Given an argumentation framework $AF = (A, R)$, an argument $a \in A$ is acceptable w.r.t. a set $S \subseteq A$ iff $\forall b \in A$ holds that if bRa then $\exists c \in S$ s.t. cRb.*

The concept of *admissibility* can then be formulated as follows:

Definition 2.4 *Given an argumentation framework $AF = (A, R)$, a set $S \subseteq A$ is admissible iff S is conflict-free and $\forall a \in S$ holds that a is acceptable w.r.t. S.*

Example For the framework $AF_{2.2}$ it means that the singleton containing just the argument 0 is not an admissible set since 0 is not an acceptable argument w.r.t. $\{0\}$, i.e. 0 is attacked by 3 but there is no attack from 0 to 3. Also the set $\{0, 2\}$ is not an admissible set because there is no attack from either 0 or 2 to the argument 3. All the other conflict-free sets are also admissible sets of $AF_{2.2}$.

On the basis of this theoretical framework we can now introduce two approaches to status evaluation of arguments commonly identified in literature as *argumentation semantics*.

2.3 Argumentation Semantics

Argumentation semantics comes usually as two approaches - *extension-based* and *labelling-based*. Depending on the approach, a semantics specifies how to extract from an argumentation framework (AF) a set of *extensions* or a set of *labellings*, where:

Definition 2.5 *[2] An extension E of an argumentation framework $AF = (A, R)$ is an admissible subset of A.*

Definition 2.6 *[2] Given a set LAB of predefined labels, a labelling L of an argumentation framework $AF = (A, R)$ is a total function $L : A \to LAB$.*

One of the simplest sets of predefined labels is the one containing three labels: *IN*, *OUT* and *UNDEC*.[1] We could specify a semantics which would assign the label *IN* to every argument that is not attacked, the label *OUT* to every argument that is attacked by an argument with an *IN* label, and the label *UNDEC* to any other argument. A labelling specified by our semantics is a triple: $L = (in, out, undec)$, where *in*, *out* and *undec* are sets containing all arguments labelled *IN*, *OUT* or *UNDEC* respectively. For $AF_{2.2}$ there exists exactly one such labelling, with $in = \{2, 3\}$, $out = \{0\}$ and $undec = \{1\}$.

The extension-based approach can be simulated by the labelling-based approach by defining a semantics for the latter which would assign the label *IN* to every argument belonging to an extension, and the label *OUT* to every other argument. It is obvious that the labelling approach is more expressive than the extension approach as a simulation in the other direction is simply not possible. That was a reason for choosing the labelling approach over the extension approach for implementation.

As it is indirectly stated in Def. 2.5, specification of a semantics may lead to more than one extension or labelling. Since there can exist more than one admissible sets

[1] I will be using the abbreviation *UNDEC* for *UNDECIDED* throughout the thesis.

for an AF, there can exist several extensions corresponding to these sets. And we know that every extension can be simulated by an appropriate assignment of labels in the labelling-approach. On the other hand, a particular semantics may prescribe no extension/labelling for an AF. It is the case when no argument fulfills the conditions defined by the semantics.

In his original paper Dung [11] mentioned four types of semantics considered nowadays "traditional" or "classical": *grounded*, *preferred*, *stable* and *complete*. Since then many other proposals have been introduced in the literature, often with the purpose of overcoming some shortcomings of the traditional approach. They include for example the *semi-stable* [8] and the *ideal* semantics [12]. Nonetheless, the present thesis is concerned primarily with the classical semantics, and precisely with the grounded, preferred and stable semantics. The reason for this is that my implementation of splitting for argumentation frameworks is based on a paper by Baumann [4] which provides proofs for the above mentioned approaches. Other important works will be included in future editions of the software.

As mentioned above, we will be dealing with the labelling-based approach to semantics. The algorithms used for computation of grounded, preferred and stable labellings are presented in Chapter 4. However, some introduction to the area of extensions is required in order to understand better the reasoning behind the splitting concept as described in [4]. Therefore, the reminder of this section provides terminological information about the respective extensions. All of them are grounded in the notion of a *complete extension* given below:

Definition 2.7 *[11] A complete extension of an argumentation framework AF, denoted as CE(AF), is an admissible set which includes all arguments that are acceptable w.r.t. it.*

Example Consider $AF_{2.3}$ in Fig. 2.3. Its admissible sets are $\varnothing$, $\{3\}$, $\{4\}$, $\{2,3\}$, $\{3,4\}$ and $\{2,3,4\}$. However, $\varnothing$ is not a complete extension as the *initial arguments*[2] 3 and 4 are acceptable w.r.t. it, but not contained in it. Alike, $\{3\}$ is not a complete extension as it defends the argument 2 by attacking the argument 1, but does not contain 2. The admissible set $\{3,4\}$ is not a complete extension for the same reason. Again, argument 3 is acceptable w.r.t. the singleton $\{4\}$, but not contained in it. And $\{2,3\}$ does not contain 4 as well. So, in case of Fig. 2.3 we have one complete extension: $\{2,3,4\}$.

Note that the status of an argument may change dynamically. Take for instance the case of $\varnothing$. It is true that it does not constitute a complete extension because 3 and 4 do not belong to it. However, if we accepted these two arguments and obtained the set $\{3,4\}$, we would see that it is not a complete extension either - due to the missing of argument 2 which is acceptable w.r.t. $\{3,4\}$. Nevertheless, the condition for rejection of $\varnothing$ as a complete extension by stating that it does not include 3 and 4 is a sufficient one.

[2]By *initial argument* we mean an argument which is not attacked.

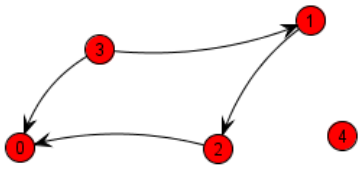

Fig. 2.3: $AF_{2.3}$

Now we can define the three extensions in question:

Definition 2.8 *[11] The grounded extension of an argumentation framework AF, denoted as GE(AF), is the least complete extension w.r.t. set inclusion.*

Definition 2.9 *[11] A preferred extension of an argumentation framework AF, denoted as PE(AF), is a maximal complete extension w.r.t. set inclusion.*

Definition 2.10 *[9] A stable extension of an argumentation framework AF, denoted as SE(AF), is a complete extension that attacks all arguments not belonging to itself.*

All three extensions are the same for the framework in Fig. 2.3: $\{2, 3, 4\}$ since there is only one complete extension for this AF. This kind of argumentation frameworks is defined by Dung as *well-founded*:

Definition 2.11 *[11] An argumentation framework is well-founded if there exists no infinite sequence of arguments $a_0, a_1, ..., a_n, ...$ s.t., for each i, a_{i+1} attacks a_i.*

Theorem 2.12 *[11] Every well-founded argumentation framework has exactly one extension which is grounded, preferred and stable.*

It is easy to see that $AF_{2.3}$ contains no cycles, i.e. fulfills the condition in Def. 2.11, and is therefore a well-founded argumentation framework.

Consider the following framework.

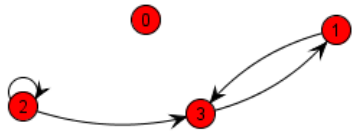

Fig. 2.4: $AF_{2.4}$

Here the grounded extension is the set $\{0\}$, the preferred the set $\{0, 1\}$, and no stable extension exists. There is an infinite sequence of arguments - precisely $1, 3, 1, 3, 1, ...$ - and the framework is not well-founded. However, we have to pay attention to the fact that Dung's definition does not say anything about frameworks containing cycles.

In order to keep this section self contained the following features of the three extensions are noted [11]. Some of them are derivable from their respective definitions:

1. Every argumentation framework possesses exactly one grounded extension;
2. Every argumentation framework possesses at least one preferred extension;
3. Every stable extension is a preferred extension, but not vice versa;
4. Not every argumentation framework possesses a stable extension.

For the labelling-approach, proposed first by Pollock [16], corresponding definitions of *grounded*, *preferred* and *stable labellings* exist. But first, like in the case of extensions, we need to define the notion of a *complete labelling*. Since we will focus solely on labellings as triples $(in, out, undec)$[3], the definition, originated by Caminada and Gabbay [9], assumes that an argument is assigned just to one of the three labels. For the purpose of defining a complete labelling we need to extend further our theoretical framework by introducing the notion of a *legal label*:

Definition 2.13 *[15] Let L be a labelling for an argumentation framework AF = (A, R). We say that $a \in A$ is:*

1. *legally IN iff $a \in in(L)$ and $\forall b : bRa$ holds that $b \in out(L)$*
2. *legally OUT iff $a \in out(L)$ and $\exists b : bRa$ and $b \in in(L)$*
3. *legally UNDEC iff $a \in undec(L)$ and $\exists b : bRa$ and $b \notin out(L)$, and $\nexists b : bRa$ and $b \in in(L)$*

In other words, an argument is *legally IN* in a labelling if and only if all its attackers are labelled *OUT*. Thus, we can see that the set of arguments being *legally IN* in a labelling corresponds to the set of arguments constituting an extension in the extension-based approach. This is stated explicitly by the following theorem:

Theorem 2.14 *[15] Let AF = (A, R) be an argumentation framework, and $E \subseteq A$. For $s \in \{complete, grounded, preferred, stable\}$:*

E is an s extension of AF iff there exists an s labelling L with in(L) = E.

For an argument to be *legally OUT* there must be at least one among its attackers that is assigned the label *IN*. And lastly, an argument is *legally UNDEC*, which means that there is no sufficient ground neither to accept it nor to reject it, if it is attacked by at least one *UNDEC*-argument and by no *IN*-argument.

Next, we need to specify the conditions under which an argument is considered *illegal*:

Definition 2.15 *[15] For a labelling L and $lab \in \{IN, OUT, UNDEC\}$, an argument a is said to be illegally lab iff $a \in lab(L)$, and it is not legally lab.*

[3]This allows us to refer to a labelling in terms of its components as $(in(L), out(L), undec(L))$.

Now, we are ready to define the *complete labelling*:

Definition 2.16 *[9] Let AF = (A, R) be an argumentation framework. A complete labelling is a labelling that does not contain any arguments that are illegally IN, illegally OUT, or illegally UNDEC.*

Given the complete labelling we define grounded, preferred and stable labellings as follows:

Definition 2.17 *[15] Let L be a complete labelling. Then:*

1. *L is a grounded labelling iff there does not exist a complete labelling L' s.t.* $in(L') \subset in(L)$
2. *L is a preferred labelling iff there does not exist a complete labelling L' s.t.* $in(L) \subset in(L')$
3. *L is a stable labelling iff undec(L) =* $\varnothing$

Computation of labellings is described in detail in Chapter 4. Here we will shortly demonstrate the introduced concepts for the framework $AF_{2.4}$. The analysis of the framework shows that argument 1 will always be labeled *IN* as no attackers for it exist. Alike, argument 2 will always be labeled *UNDEC*. 2 cannot be *IN* because then due to the self-loop it would have to be *OUT*. 2 cannot be *OUT* either since then as the only *OUT*-attacker of itself it would have to be labeled *IN*. Therefore it has to be *UNDEC*. The status of 3 and 1 is more complicated. We know that 3 is being attacked by an *UNDEC* argument. Therefore 3 can never be assigned the label *IN*. However, it can be either *OUT* or *UNDEC*. If 3 is labeled *OUT* then 1 has to be labeled *IN*. If 3 is labeled *UNDEC* then 1 also has to be labeled *UNDEC*. In effect, there exist 2 complete labellings for this AF: $(\{0\}, \varnothing, \{1, 2, 3\})$ and $(\{0, 1\}, \{3\}, \{2\})$. According to the given definitions we can see that the first complete labelling is the grounded labelling since its *in* set is minimal. The second complete labelling is the preferred labelling since its *in* set is maximal. The stable labelling does not exist because argument 2 is always labeled *UNDEC* which makes the *undec* set not empty.

At the end of this introductory Chapter it is worth mentioning that all the semantics in question have some drawbacks. For the grounded semantics it is the limitation posed by so-called *floating defeat*. A floating defeat takes place in frameworks with no initial nodes (i.e. those that are not attacked by any argument). The grounded semantics treats all arguments as provisionally defeated (= *UNDEC* in the labelling-based approach). However, a more appropriate way to deal with this kind of frameworks would be to apply some kind of differentiation of the justification status of every single argument. Consider the framework in Fig. 2.5. The grounded extension, as well as the grounded labelling, is an empty set. Nonetheless, we can see that the argument 2 is never in the extension as it is attacked either by 0 or by 1, one of which should be in the extension each time. On the other hand, the argument 3 does always belong to the extension as

it is attacked by 2 which never belongs to the extension. [3, p. 172] This insufficiency of the grounded semantics is solved in the preferred and stable semantics.

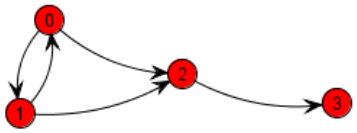

Fig. 2.5: $AF_{2.5}$ (floating defeat)

There are argumentation frameworks where no stable extension (labelling) exists. For instance, the $AF_{2.4}$ is such a framework. As it is argued by Baroni et al. in [3, p. 168], the non-existence of an extension entails that all arguments are not justified. But then again, the argument 0 in $AF_{2.4}$ has no defeaters and should therefore be classified as undefeated. The preferred semantics takes care of this problem as it treats correctly cases like $AF_{2.4}$ by rightly assigning the accepted status to 0.

Yet another problematic issue for the stable and preferred semantics, as Pollock [16, p. 393] has pointed out, is that there is inconsistency in both semantics as to the way they treat frameworks with odd-length cycles and even-length cycles. The stable semantics is notorious in its uniform treatment of frameworks with odd-length cycles, for which it produces no stable extension. However, once we have an even-length cycle, the extensions exist. In this matter, the preferred semantics also fails to treat odd and even cycles the same way. For odd-length cycles it produces an empty extension and a labelling in which all elements are assigned the label *UNDEC*. The problem arising from this fact - called *the witnesses problem* - is detailed in Baroni et al. [3].

Chapter 3

Splitting Argumentation Frameworks

After the introduction into the area of formal argumentation we can proceed to the main topic of the thesis. In 1994 Lifschitz and Turner [14] showed the splitting of a logic program. The authors stated that it works especially well for programs with negation as failure as well. A year later, Dung in his paper [11] pointed out that argumentation can be viewed as a special form of logic programming with negation as failure. The natural link one can establish would be to develop a procedure for splitting an argumentation framework. Such a procedure is presented in Baumann [4]. In this chapter I will present briefly the theoretical framework of splitting as given in [4]. An extensive description of the algorithms developed in the process of implementation is given in Chapter 4.

3.1 Theoretical Foundations

The general idea of splitting is grounded in a thought that we could be able to obtain better computational performance if the framework in question is first split into two parts which are then computed separately. At the end, the outputs of both computations (i.e. a partial extension or a partial labelling) will be combined, thus producing the required result. The following definition formalizes the concept:

Definition 3.1 *[4] Let $AF_1 = (A_1, R_1)$ and $AF_2 = (A_2, R_2)$ be argumentation frameworks s.t. $A_1 \cap A_2 = \varnothing$. Let $R_3 \subseteq A_1 \times A_2$. We call the tuple (AF_1, AF_2, R_3) a splitting of the argumentation framework $AF_0 = (A_1 \cup A_2, R_1 \cup R_2 \cup R_3)$.*

As stated in the above definition, in order to split an argumentation framework certain conditions have to be fulfilled. The first one being that the splitting divides the set of arguments into two disjoint sets. The second being that all attacks lying between the two sets (R_3) have to have their source in the first set (A_1) and their target in the second set (A_2).

Having established the requirement of a single direction, we need to specify the treatment of arguments on both sides of the attacks in R_3. In the labelling-based

approach there are three possible states for an argument: *IN*, *OUT* or *UNDEC*. While computing a labelling prescribed by a particular semantics we are concerned with the immediate environment of every single argument. This means that we decide the status of an argument based on the acceptability status of its attackers. Speaking in general terms, we need to decide what kind of impact the splitting will have on the set A_2. What will happen if the attacker is in the *in*, *out* or *undec* set?[4]

Lets take a look at the situation when the attacker is an element of the *in* set. Assume that our framework looks like the one in Fig. 3.1. If we split it to be $AF_1 = (\{0\}, \varnothing)$, $AF_2 = (\{1,2\}, \{(1,2)\})$ and the singleton $R_3 = \{(0,1)\}$ we will obtain as the unique grounded, stable and preferred labelling [5]

- $(\{0\}, \varnothing, \varnothing)$ for AF_1
- $(\{1\}, \{2\}, \varnothing)$ for AF_2

Fig. 3.1: $AF_{3.1}$

The *IN*-attacker in question is here the argument 0. After applying the union operation to both outputs we get $(\{0,1\}, \{2\}, \varnothing)$ as the grounded, preferred and stable labelling. However, the grounded, preferred and stable labelling of $AF_{3.1}$ is $(\{0,2\}, \{1\}, \varnothing)$. In the end, we cannot reconstruct the original labellings. It is because the splitting did not consider the attack $(0,1) \in R_3$. This attack changes the acceptability status of argument 1 to *OUT* as it is attacked by argument 0 which is *IN*. So, in case when the attacker belongs to the *in* set there exists a problem of incorrectly assigning acceptability status to arguments it attacks. So, it is inappropriate to disregard the internal structure when splitting the framework. This insufficiency can be corrected by the notion of a *reduct*:

Definition 3.2 *[4] Let $AF = (A, R)$ be an argumentation framework, A' a set disjoint from A, $S \subseteq A'$ and $L \subseteq A' \times A$. The (S, L)-reduct of AF, denoted as $AF^{S,L}$, is the argumentation framework*

$$AF^{S,L} = (A^{S,L}, R^{S,L})$$

where $A^{S,L} = \{a \in A |\ \nexists b \text{ s.t. } b \in S \text{ and } (b,a) \in L\}$, $R^{S,L} = \{(a,b) \in R | a, b \in A^{S,L}\}$.

[4] I adopt this approach since we are dealing with labellings in this thesis. However, this differentiation corresponds closely to Baumann's extension-based approach. Arguments labeled *OUT* correspond to Baumann's "arguments that are attacked by the extension". Arguments labeled *UNDEC* correspond to Baumann's "arguments that are not in the extension and are not attacked by the extension". Arguments labeled *IN* correspond naturally to the extension. See also Def. 2.13.

[5] Note that $AF_{3.1}$ is well-defined and therefore the three labellings are the same.

We apply this definition to AF_2 by setting $S = E_1$, where E_1 is the extension of AF_1. In effect, all arguments that are attacked by R_3 and which sources are in the extension (i.e. the *in* set) are removed from AF_2:

$AF_2^{E_1=\{0\},R_3=\{(0,1)\}} = (A_2^{E_1,R_3}, R_2^{E_1,R_3}) = (\{2\}, \varnothing)$

The union of labellings $(\{0\}, \varnothing, \varnothing)$ (for AF_1) and $(\{2\}, \varnothing, \varnothing)$ (for $AF_2^{E_1,R_3}$) reconstructs the original extension. Note that it does not reconstruct the entire labelling as argument 1 is removed by *reduct*. Clearly, in the case when the source of an attack is labeled *IN*, by applying *reduct* we will remove only arguments that would have been labeled *OUT*.[6] No element of an extension will be removed by this operation.

Consider that *reduct* cannot be applied when the attacker is in the *out* set. By removing the target of the attack we could render the extension wrong as some of the removed arguments could have been in it (i.e. in the *in* set as is stated in Theorem 2.14).

The last scenario to check out is the one in which the attacker is assigned the label *UNDEC*. By definition, no argument attacked by an *UNDEC* argument can be *IN*. So, it has to be either *OUT* or *UNDEC*. Here we see that removing such an argument does not affect the extension.

Consider again the argumentation framework given in Fig. 3.1. By adding a self-loop around argument 0 we guarantee that 0 will always (i.e. in any labelling) belong to the *undec* set. As 0 is the sole attacker of 1, 1 also has to be labeled *UNDEC*. The same applies to argument 2 being attacked by only one argument which is *UNDEC*. Thus we obtain as the grounded and preferred labelling the triple $(\varnothing, \varnothing, \{0, 1, 2\})$ and no stable labelling exists. However, if we split the framework on the attack line between the arguments 0 and 1, we obtain:

- The grounded and preferred labelling $(\varnothing, \varnothing, \{0\})$ and no stable labelling for AF_1;
- The grounded, preferred and stable labelling $(\{1\}, \{2\}, \varnothing)$ for AF_2

The union will produce $(\{1\}, \{2\}, \{0\})$ as the grounded and the preferred labelling, and no stable labelling.[7] This result lies far away from the expected $(\varnothing, \varnothing, \{0, 1, 2\})$. An intuitive solution to the problem is given in [4] where it is proposed to introduce a self-loop around the argument being attacked by an *UNDEC* argument. To make the section self contained we present the two definitions given there on which the idea is based:

Definition 3.3 *[4] Let $AF = (A, R)$ be an argumentation framework, E an extension of AF. The set of arguments undefined w.r.t. E is*

[6] It is because by definition no argument which is attacked by an *IN*-argument can belong to the *undec* set.

[7] Depending on the implementation it may be necessary to insert additional code to ensure that no partial stable labelling is being generated. If not implemented correctly it could give rise to the problem of claiming the existence of a labelling while none actually exists. In this particular case, we could be creating a labelling $(\{1\}, \{2\}, \varnothing)$ as union of the stable labelling for AF_2 with the non-existing labelling for AF_1.

$$U_E = \{a \in A | a \notin E, \nexists b \in E \ s.t. \ (b, a) \in R\}.$$

In other words, an argument undefined with respect to an extension is one that is neither in the extension nor attacked by the extension. This definition resembles what we have said about the undecided arguments in previous chapter. If an argument a is in the extension it is labeled *IN*. If a is attacked by an argument in the extension (i.e. belonging to the *in* set), it has to be labeled *OUT*. So, if a is neither in the extension (i.e. a is *IN*) nor attacked by the extension (i.e. a is *OUT*) it has to be *UNDEC*. It follows that the notion of an undefined argument corresponds to the notion of an undecided argument.

Now, we can modify the framework AF_2:

Definition 3.4 *[4] Let $AF = (A, R)$ be an argumentation framework, A' a set disjoint from A, $S \subseteq A'$ and $L \subseteq A' \times A$. The (S, L)-modification of AF, denoted $mod_{S,L}(AF)$, is the framework*

$$mod_{S,L}(AF) = (A, R \cup \{(b, b)\} | a \in S, (a, b) \in L).$$

Taking the argumentation framework in Fig. 3.1 (containing also a self-loop around argument 0) as an example, we search for all undefined arguments w.r.t. an extension of $AF_1 = (\{0\}, \varnothing)$. In our example it will always be 0. Next, we apply the operation of modification to our AF_2:

$mod_{S=U_{E_1}=\{0\},R_3=\{(0,1)\}}(AF_2) = (\{1, 2\}, \{(1, 2), (1, 1)\})$.

With a self-loop added to argument 1 we obtain the labelling $(\varnothing, \varnothing, \{1, 2\})$ for the grounded and preferred semantics and no labelling for the stable semantics. Thus the union renders the expected labelling $(\varnothing, \varnothing, \{0, 1, 2\})$ for the grounded and preferred semantics and no labelling for the stable semantics.

The operation of modification applied to the framework 3.1 gives the correct results. However, the idea of splitting is to compute the *modification of reduct* of an argumentation framework since we need to take into consideration both operations. So the right way to proceed is to compute:

$$mod_{U_{E_1},R_3}(AF_2^{E_1,R_3}) \ ([4]).$$

Remark It is noteworthy that the application of splitting, although removing some of the arguments, allows for the actual reconstruction of the entire labelling. From what is observed above, by assigning all arguments removed by the operation of *reduct* to the *out* set we obtain a labelling in which all arguments are accounted for.

3.2 Strongly Connected Components in Splitting

As pointed out by Baumann in [4] there are many possible ways to split an argumentation framework. Actually, we can do it anywhere as long as the condition of single

direction for R_3 is fulfilled. In the developed application it gave rise to a double approach which is presented in Chapter 4.

In order to ensure that splitting takes place "at a right place" we need to follow the internal structure of the directed graph that underlies each framework. An intuitive way would be to look for any kind of ordering which would give a hierarchical picture of the graph. Baumann [4] is doing it by searching for the set of *strongly connected components* (SCCs).

In graph theory, a strongly connected component (SCC) of a directed graph is a maximal subgraph in which there is a path from each vertex to every other vertex. By contracting each SCC to a single vertex we obtain a directed acyclic graph, i.e. a directed graph that contains no cycles. A directed acyclic graph induces a partial order on the set of vertices thus giving us the required hierarchy. As stated in [4, 5], based on this order any SCC-decomposition can be easily transformed into a splitting. The most obvious is to partition the graph into initial SCCs (i.e. those that are not attacked by any other SCC) and the non-initial SCCs. The union of vertices of all initial SCCs can then be taken as the set A_1 and the union of vertices of all non-initial SCCs would constitute the set A_2.

Example The directed graph in Fig. 3.2 with 13 vertices has 6 SCCs: $\{0, 1\}$, $\{2, 3, 4\}$, $\{5, 6, 7, 8\}$, $\{9\}$, $\{10\}$, $\{11, 12\}$. Splitting is possible on any of the edges joining any two SCCs: $(0, 6)$, $(5, 2)$, $(9, 7)$, $(8, 10)$ or $(10, 11)$. If the splitting is performed on

- the edge $(0, 6)$ then $A_1 = \{0, 1\}$ and $A_2 = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
- the edge $(8, 10)$ then $A_1 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and $A_2 = \{10, 11, 12\}$
- the edges $\{(5, 2), (8, 10)\}$ then $A_1 = \{0, 1, 5, 6, 7, 8, 9\}$ and $A_2 = \{2, 3, 4, 10, 11, 12\}$.

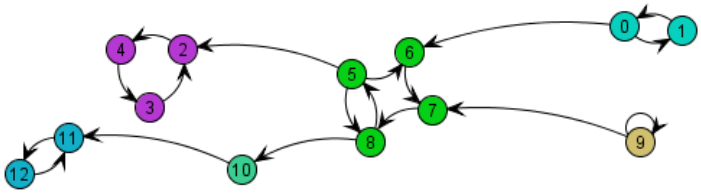

Fig. 3.2: SCCs of a directed graph

The developed software covers two types of splitting. The first one, which can be called *initial splitting*, divides the graph into the initial and the non-initial SCCs. It corresponds to the splitting on the edges $\{(0, 6), (9, 7)\}$ in Fig. 3.2. The other kind of splitting, which can be called *iterative splitting* or *optimized splitting*, is based on arbitrarily chosen cardinality constraints in determining how deep into the partial order we can go. The procedure is described in Chapter 4. Here I just want to mention that the splitting performed by it corresponds to splitting on the edges $\{(5, 2), (8, 10)\}$ in Fig. 3.2.

Chapter 4

The Application (v. 1.0)

This chapter presents the implementation of concepts described in two previous chapters. They were put together to produce a standalone application written in the Java programming language. The current version is 1.0.

The application does not require an installation. However, it may be necessary to install or update the Java Runtime Environment (JRE) which is available on-line at no charge.

The distribution format is zip. The zipped file has the size of 1.3 MB. It contains a .jar executable named AAF and a folder containing third party Java libraries (see remark). In order to use it one has to extract the zip file and then navigate to the executable AAF.jar. A double mouse click should be enough to run the application. Note that the AAF file has to be located on the same hierarchical level as the libraries folder.

Remark The application makes use of the Java Universal Network/Graph Framework [13] and the Apache Commons project's [10] libraries.

The chapter is organized as follows. The first section deals with the most important features of the graphical user interface including the functionality contained in menus, the control panel and the graph display area. The second and third sections are concerned with algorithms for computing labellings and splitting. And the last section talks briefly about the package and class structure of the application.

4.1 Graphical User Interface

The GUI was designed to be as simple and user friendly as possible. As presented in Fig. 4.1, it consists of four main components. On the top there is a menu with four items. On the left is the control panel. On the right there is the display. And finally on the bottom there is a field for outputs.

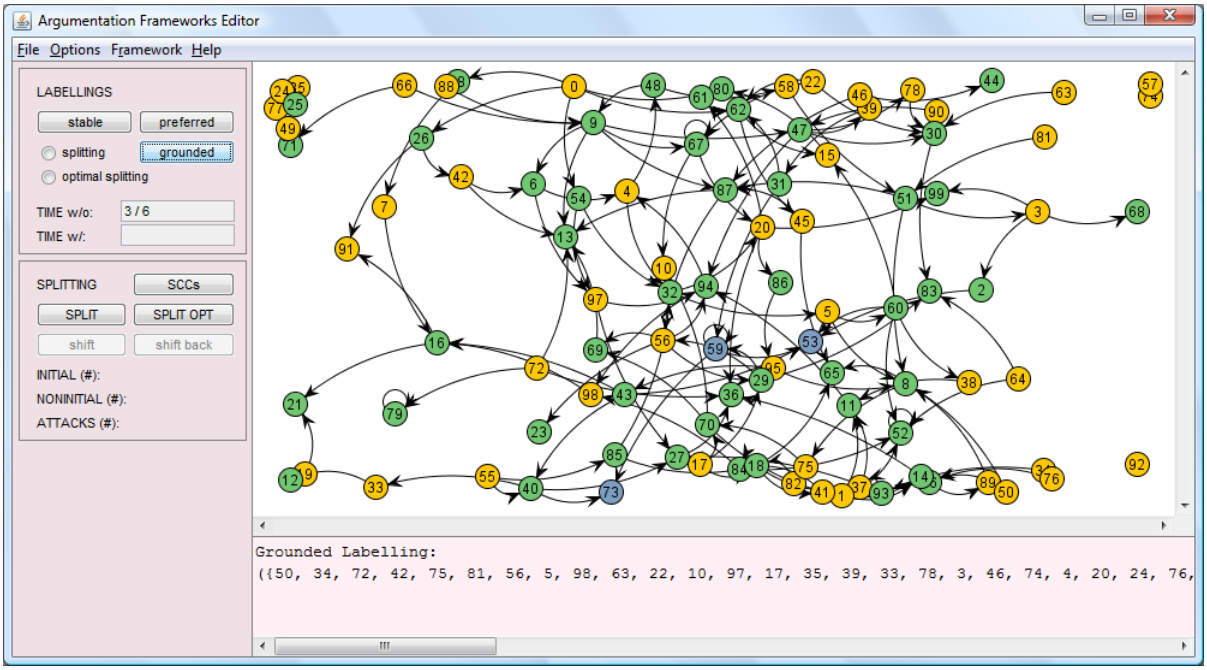

Fig. 4.1: Graphical User Interface

Menu

The *File* drop-down menu provides the following functionality:

- *New Framework*: Creates an empty framework which is yet to be populated. The user will see an empty white sheet in the display area.
- *Open Framework...*: Opens an existing file with framework data. The supported formats include so far **.net*[8] and **.aaf*[9]. Specification of both formats can be found in the help menu.
- *Discard Framework*: Deletes the current framework. The user will see an empty grey sheet in the display area.
- *Save Framework*: Saves the current framework into a file with the extension *aaf* or *net*. As a **.net* file does not contain the information about coordinates the file size is much smaller than an **.aaf* file with data of the same argumentation framework.
- *Export Image*: With this function the current framework can be saved as *gif*, *jpg* or *png* image.

[8]It is a subset of the well-known Pajek format. It specifies arguments (without coordinates) and attacks. Since no coordinates are given an FR layout is applied when a file with this extension is loaded.

[9]It is a file format containing not only specification of arguments and attacks but also the coordinates of arguments; courtesy of Jochen Tiepmar.

- *Generate Random Framework...* (Fig. 4.2): Gives the possibility of creating a random AF by specifying number of arguments and attacks. User input is then checked in two ways. On one hand, any missing input or input containing other characters than numbers is signaled by a corresponding information at the bottom of the input dialog. On the other hand, a check as to the allowable number of attacks for the given number of arguments is performed, e.g. as self-loops are permitted, for n arguments up to n^2 attacks are possible.

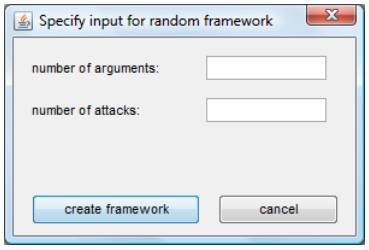

Fig. 4.2: Random framework generator dialog

- *Exit*: If no arguments are detected in the display area the application will exit. If there are arguments detected a dialog window will appear asking the user to confirm the exit commando.

Next to the *File* menu there is a drop-down menu *Options*. It allows currently to activate / deactivate the display of argument and attack names.

The menu *Framework* comprises of two parts. The first one allows for change of layouts. At present implemented are the *circle layout*, *Fruchterman-Rheingold layout*, *Isom layout*, *Kamada-Kawai layout* and the *spring layout*. The second part contains three modes for graph manipulation. These are:

- *editing mode*: Gives the possibility of directly inserting arguments and attacks to the graph. Can also be activated by clicking on the display area and typing the letter *e*.
- *picking mode*: Enables the user to directionally rearrange a subset of arguments. Can also be activated by clicking on the display area and typing the letter *p*.
- *transforming mode*: Encompasses several operations like zooming, panning and rotating of the graph. Can also be activated by clicking on the display area and typing the letter *t*.

The menu item *Help* contains everything that is needed in order to be able to use the software. All instructions regarding the operations on the graph, file format specifications and explanations for reading of the outputs are given here.

Control Panel

The control panel is divided into two sub-panels. The first one is named *Labellings* (Fig. 4.3), the other one is named *Splitting* (Fig. 4.4).

The *Labellings* sub-panel was designed with the purpose of testing and comparing runtimes. It contains three semantics buttons for computing grounded, preferred and stable labellings. Further, two more options are given which implement two variants of splitting. The first variant corresponds to procedure *compute_*$A_1(SCC(AF))$ in Alg. 4. By selecting the radio button *splitting* and then pressing one of the semantics buttons a corresponding set of labellings is computed by applying the splitting algorithm. Analogously, if *optimal splitting* is selected we obtain a set of labellings by applying the optimized variant of the splitting algorithm. This second variant of splitting corresponds to procedure *optimize_*$A_1(SCC(AF), A_1)$ in Alg. 4.

If any of the semantics buttons is pressed without any radio button selected the computation will proceed without splitting the framework.

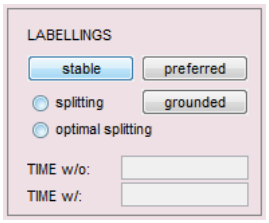

Fig. 4.3: Control Panel: Labellings

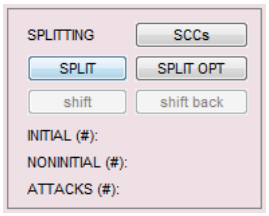

Fig. 4.4: Control Panel: Splitting

The result of a computation is shown in one of the rows below the buttons area. As the names suggest, *TIME w/o* and *TIME w/* will output the time (in *ms*) of a computation without splitting or with splitting (either regular or optimized) respectively. The runtime values can then be compared directly by checking the both outputs.

After performing a computation you will see a forward slash and an integer right behind the runtime value. This integer indicates the number of runs of the main algorithm. Taking the framework of 10 arguments and 100 attacks as example, the computation of preferred labellings will take around 260,000 ms and 9,864,101 steps. So, the output in the field *TIME w/* will read "260000/9864101".

The sub-panel *Splitting* was designed to give a visualization of splitting processes. So, by pressing the button *SCCs* we color the framework graph in accordance with its internal structure, i.e. each SCC is marked in a different color. The second row buttons, *SPLIT* and *SPLIT OPT* allow the user to show on the graph how the regular splitting or the optimized splitting would look like if it were applied. The set R_3, i.e. the attacks which lie between the two sets, is shown in gray. Additionally, we can shift the partitioned arguments. By clicking *shift* we move the initial arguments (set A_1) to the right and the non-initial arguments (set A_2) to the left. The operation can be reverted by pressing the *shift back* button. Each time a splitting is visualized the number of initial arguments, the number of non-initial arguments and the number of attacks between the two sets are printed in the three fields at the bottom of the sub-panel.

Display and Output Area

Finally, the display area is set to contain the graphical representation of an argumentation framework. It is equipped with scroll bars which will be automatically adjusted when the framework is resized or moved. It supports a collection of mouse and keyboard operations. These include:

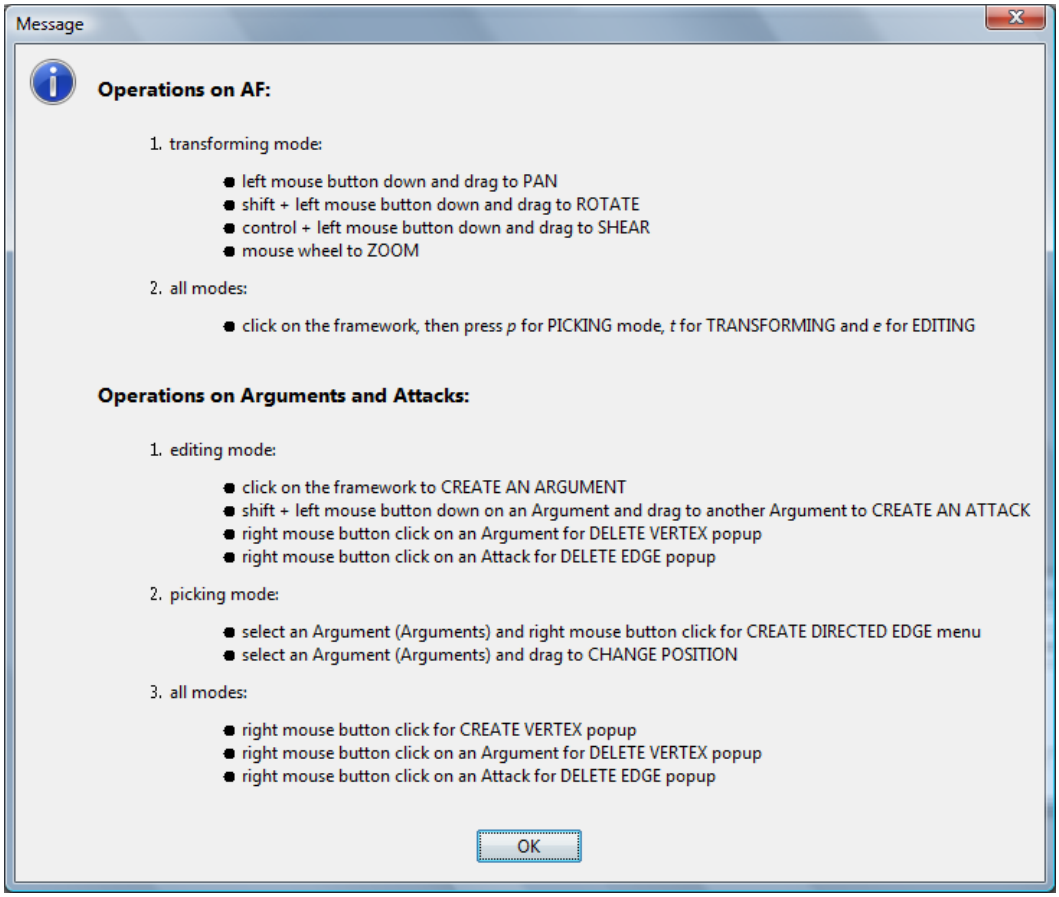

Fig. 4.5: Graph operations

The output area located below the display area shows the result of the last operation executed. When computing a labelling, strongly connected components or a splitting, the situation on display will be mirrored by a textual output of results.

When computing a labelling, the color of arguments contains the following information[10]:

- (a) indicates the default color of an argument. Red is also used to indicate the special case of a non-existing stable labelling.
- (b) indicates that an argument belongs to the *in* set of a particular labelling.

[10]Note that in cases where more than one labelling exists the coloring of arguments and attacks refers only to the last labelling.

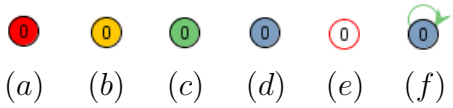

Fig. 4.6: Argument coloring

- (c) indicates that an argument belongs to the *out* set of a particular labelling.
- (d) indicates that an argument belongs to the *undec* set of a particular labelling.
- (e) indicates that an argument was removed in the process of the *reduct* operation. In argumentation frameworks where no stable labelling exists it is possible that some of the arguments are marked like this and not in the default red color. This is not a bug. It is caused by the fact that partitioning of the AF may result in one subset (specifically A_1) containing no *UNDEC* arguments. That means that the non-existence of a labelling is not detected at this stage. In such situation, the operations of *reduct* and *modification* will take place. In effect, some arguments will be removed. Only when the fact that there is no stable labelling is detected early (i.e. for set A_1) the entire framework will have the default color.
- (f) indicates an argument with a self-loop added during the operation of *modification*. Note that the argument has to be either *OUT* or *UNDEC*. It cannot be *IN* since it has to be attacked by an *UNDEC* argument in order to receive a self-loop.

For attacks there are three possible colors. The default is black. The attacks on which splitting took place are indicated in gray. If an attack is marked red it means that it was removed during a *reduct* operation.

4.2 Algorithms I: Labelling

The application implements three of the classical semantics mentioned by Dung: grounded, preferred and stable. It is based on labelling algorithms given by Modgil and Caminada in [15]. In what follows I will describe their working.

4.2.1 Computing the Grounded Labelling

The grounded labelling (L_{gr}) is generated as follows: all arguments which are not attacked are assigned the label *IN*. The next step is to assign the label *OUT* to all those arguments that are attacked by at least one of the arguments just labeled *IN*. We continue assigning the label *IN* to any argument having all of its attackers labeled *OUT*. The iteration stops when no further assignment can be made. The set $undec(L_{gr})$ is the set of arguments from A which were not labeled during the iteration. The exact procedure is given in Alg. 1.

Algorithm 1: Computation of Grounded Labelling

input : $L_0 = (in(L) = \varnothing, out(L) = \varnothing, undec(L) = \varnothing)$

1.1 **repeat**

1.2 $in(L_{i+1}) = in(L_i) \cup \{x | x$ is not labelled in L_i, and $\forall y$: if $(y, x) \in R$ then $y \in out(L_i)$

1.3 $out(L_{i+1}) = out(L_i) \cup \{x | x$ is not labelled in L_i, and $\exists y$: if $(y, x) \in R$ and $y \in in(L_i)\}$

1.4 **until** $L_{i+1} = L_i$

1.5 **return** $L_{gr} = (in(L_i), out(L_i), A - (in(L_i) \cup out(L_i)))$

Fig. 4.8 shows the grounded labelling of the framework $AF_{4.7}$ given in Fig. 4.7. The *in* set is marked orange, the *out* set is marked green, and the *undec* set is marked blue.

We start the computation with an empty labelling. In first step we look for arguments that are not attacked. In $AF_{4.7}$ there is only one such argument: 6. So, 6 is labeled *IN*. Next, we label *OUT* all arguments having at least one attacker that is labeled *IN*. It applies only to argument 5. The labelling contains at this stage two arguments: $L_1 = (\{6\}, \{5\}, \varnothing)$. We then proceed by assigning the label *IN* to all arguments whose attackers are all marked as *OUT*. It is the argument 7 which is attacked only by 5. Then, there is only one argument having all of its attackers assigned the label *OUT*: 8. The labelling $L_2 = (\{6, 7\}, \{5, 8\}, \varnothing)$. At this moment there are no more arguments that can be labeled either *IN* or *OUT*. So, the $L_{gr} = (\{6, 7\}, \{5, 8\}, \{0, 1, 9\})$.

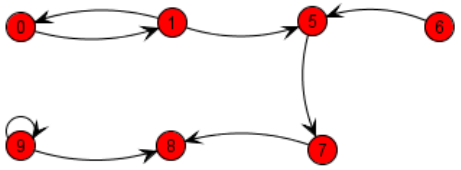

Fig. 4.7: $AF_{4.7}$

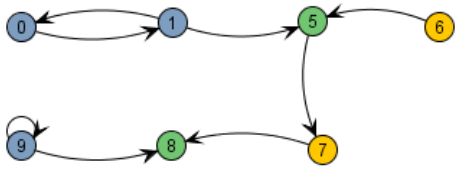

Fig. 4.8: $AF_{4.7}$ (grounded labelling)

It is worth noting that the algorithm runs in polynomial time. According to the tests done during the phase of performance evaluation the runtime for grounded semantics was very short. For instance, for frameworks with 1000 arguments and 2000 attacks it was as low as 40 ms.

4.2.2 Computing Preferred Labellings

In order to present the algorithms for preferred and stable labellings we need to introduce the notion of being *super-illegally IN*:

Definition 4.1 *[15] Given an $AF = (A, R)$ and a labelling L, an argument $a \in A$ is super-illegally IN in L iff it is illegally IN in L and $\exists b : (b, a) \in R$ and b is legally IN or UNDEC in L.*

The algorithm for computing all preferred labellings (Alg.2) starts by assigning to all arguments the label *IN* (labelling L_{IN}), and initializing an empty set in which candidate

labellings are to be stored. Then, by way of the main procedure *find_labellings* arguments that are *illegally IN* in L_{IN} are identified. To each of these arguments a procedure called *transition_step* is applied, by which the label of the given argument is changed from *IN* to *OUT*. If such an argument whose label has been changed from *IN* to *OUT* or if any argument(s) it attacks is *illegally OUT*, it will be relabeled as *UNDEC*. Thus we have obtained a new labelling which contains one less *IN*-argument. Then the entire process repeats again by passing any new labelling onto the main procedure, and the process continues until an acceptance or rejection condition is met. A labelling which does not have any argument which is *illegally IN* will be added to the candidate labellings, unless at any previous stage in the recursion it is detected that a better labelling has been found, i.e. a labelling with a larger *in*-set is already contained in *candidate_labellings*. If such a labelling with a larger cardinality of the *in*-set exists, the current labelling will not be processed.

Algorithm 2: Computation of Preferred Labellings

input : $L_{IN} = (in(L_{IN}) = A, out(L_{IN}) = \varnothing, undec(L_{IN}) = \varnothing)$

```
2.1  candidate_labellings := ∅
2.2  find_labellings(L_IN)

2.3  PROCEDURE find_labellings(L)
2.4  begin
2.5      if ∃L' ∈ candidate_labellings : in(L) ⊂ in(L') then return;
2.6      if L does not contain an argument illegally IN then
2.7          foreach L' ∈ candidate_labellings do
2.8              if in(L') ⊂ in(L) then
2.9                  candidate_labellings := candidate_labellings − {L'}
2.10             end
2.11         end
2.12         candidate_labellings := candidate_labellings ∪ {L}
2.13         return;
2.14     else
2.15         if L has an argument that is super-illegally IN then
2.16             x := some argument that is super-illegally IN in L
2.17             find_labellings(transition_step(L, x))
2.18         else
2.19             foreach x that is illegally IN in L do
2.20                 find_labellings(transition_step(L, x))
2.21             end
2.22         end
2.23     end
2.24 end
```

In order to avoid the situation in which incomplete labellings are being generated by any incorrect assignment of labels, the algorithm is designed to always extract first those arguments that are *super-illegally IN*, i.e. arguments having at least one attacker *legally IN* or *UNDEC*, whenever we try to extract arguments that are *illegally IN*.

An example of a framework where this requirement is needed is given in [15, p. 128] and reproduced in Fig 3.1. Assuming no differentiation was made between arguments *illegally IN* and *super-illegally IN*. There would be two possible runs for the algorithm as both 1 and 2 are *illegally IN* (both are attacked by an argument that is labeled *IN*: 0 and 1 respectively). The first possibility is to first select the argument 1 and then the argument 2. This run results in a complete labelling. However, the run starting with 2 does not. It relabels 2 from *IN* to *OUT*. Then, in the next recursion step, it relabels 1 from *IN* to *OUT* and 2 from *OUT* to *UNDEC*. So, we get a labelling $L = (\{0\}, \{1\}, \{2\})$ which is *illegally UNDEC* since 2 is attacked by an *OUT* argument. As stated in Def. 2.16 a complete labelling is not allowed to have any arguments that are *illegally UNDEC*.

Turning back to our framework in Fig. 4.7, it possesses two preferred labellings given in Figs. 4.9 and 4.10.

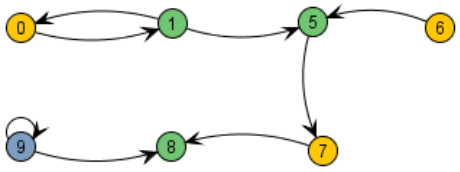

Fig. 4.9: $AF_{4.7}$ (preferred labelling 1)

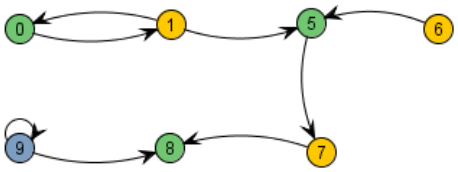

Fig. 4.10: $AF_{4.7}$ (preferred labelling 2)

4.2.3 Computing Stable Labellings

The algorithm for computing all stable labellings is obtained by rewriting line 2.5 of the algorithm for preferred labellings to read "**if** $undec(L) \neq \varnothing$ **then return**". If the set of arguments labeled *UNDEC* in a labelling is not empty, i.e. it violates the requirement for a stable labelling, the labelling will not be further processed.

According to the instruction in [15] we can also skip the lines 2.7 to 2.11 which compare the *in* sets of candidate labellings with those of the already computed labellings.

In the framework $AF_{4.7}$ there exists no stable labelling. This is clear even from the look at the two preferred labellings in previous subsection.[11] Both have an argument which is labeled *UNDEC*. However, the algorithm for computing preferred labellings after modification of the line 2.5 will not allow for generation of a labelling containing any *UNDEC* element.

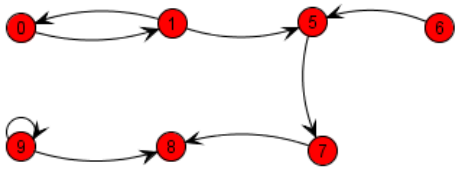

Fig. 4.11: $AF_{4.7}$ (no stable labelling exists)

[11] We have said in Chapter 2 that a stable labelling is also a preferred labelling.

4.3 Algorithms II: Splitting

This section contains two algorithms. Due to the requirement of single directionality, we need to find *the strongly connected components* of an argumentation framework prior to splitting it. Undoubtedly, one of the most known and most efficient ones is the algorithm proposed by Tarjan in 1972 [17]. The introduction of splitting brings an additional computational overhead in form of time needed for dividing the framework. That is why it is especially important for us to use an efficient algorithm and Tarjan's with its linear complexity is a good choice.

The other algorithm was developed based on the procedures and definitions given in [4]. It is concerned with the problem of splitting of an argumentation framework.

4.3.1 Tarjan's Algorithm for Computing SCCs

The set of strongly connected components as presented in Tarjan's paper is computed by means of depth-first search (Alg. 3). We start by initializing variables *index* and *lowlink* to -1 for every argument of the framework. *index* will number the arguments consecutively in the order in which they are discovered by the algorithm. *lowlink* will be updated during computation and will always be smaller or equal to the *index* value for a particular argument. At the end of the computation all the arguments of a SCC will have the same *lowlink* value which is equal to the *index* of the first discovered argument of the SCC. Arguments which have been visited during the search but have not yet been placed in a SCC are stored on a stack.

Starting with an arbitrary argument, we push it on the stack. Then, if any of the arguments it attacks has not yet been visited (i.e. its $index = -1$), we apply the algorithm to that argument. When the algorithm identifies an argument as having its $index = lowlink$ (i.e. an SCC was found), that argument and all the arguments on top of it are removed from the stack and added to the set $SCC(AF)$ as a strongly connected component.

Example The algorithm of Tarjan finds six strongly connected components for the argumentation framework $AF_{4.7}$ (each receiving a different color):

$SCC(AF_{4.7}) = \{\{0, 1\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\}\}$

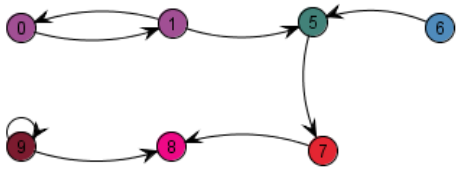

Fig. 4.12: $AF_{4.7}$ (the strongly connected components)

Algorithm 3: Computation of SCCs

input : an argumentation framework $AF = (A, R)$
output : set of strongly connected components ($SCC(AF)$)

```
3.1  foreach x ∈ A do index(x) := lowlink(x) := −1
3.2  SCC(AF) := ∅
3.3  stack := ∅
3.4  counter := 0
3.5  while index(x) = −1 do compute_SCCs(x)
3.6  print SCC(AF)

3.7  PROCEDURE compute_SCCs(a)
3.8  begin
3.9      push a on the stack
3.10     index(a) = counter
3.11     lowlink(a) = index(a)
3.12     counter = counter + 1
3.13     foreach (a, b) ∈ R do
3.14         if index(b) = −1 then
3.15             compute_SCCs(b)
3.16             lowlink(a) = min(lowlink(a), lowlink(b))
3.17         else if b on stack then
3.18             lowlink(a) = min(lowlink(a), index(b))
3.19         end
3.20     end
3.21     if lowlink(a) = index(a) then
3.22         SCC = ∅
3.23         repeat
3.24             pop argument c from stack
3.25             add c to SCC
3.26         until c = a
3.27         add SCC to SCC(AF)
3.28     end
3.29 end
```

4.3.2 The Splitting Algorithm

Our splitting algorithm consists of two parts: The first part (Algorithm 4) is executed prior to the first call of a labelling algorithm for a semantics and computes AF_1, AF_2 and the set R_3. The second part (Algorithm 5) is executed after receiving an extension from the labelling algorithm. The tuple AF_2 is then modified in accordance with the extension.

The first task is set to look for all the initial arguments (A_1) of our framework (AF). We use the set of *strongly connected components* returned by the Tarjan algorithm. The algorithm starts by introducing a Boolean variable *scc_attacked* which will be initialized to *false* for every SCC in $SCC(AF)$. Given an SCC, once an argument in this SCC is attacked by some argument in another SCC, the variable will be set to *true* and the

execution of the algorithm for this SCC stops. Then the algorithm starts processing the next SCC. Only if *scc_attacked* remains *false*, which means that the corresponding SCC is not attacked, will all the arguments of this SCC be added to A_1.

The splitting operation described above may result in subframeworks which differ a lot in size. We also provide a possibility to equalize the cardinalities along the partial ordering dictated by $SCC(AF)$.[12] The algorithm, called *optimize*, accepts the already computed arguments of A_1 and adds new ones under certain conditions. The first criterion used is the cardinality of A_1. Since the addition of new arguments relies on the partial order, it may not always be possible. Therefore, choosing 45% as a starting condition for equalization was an attempt to optimally equalize the numbers of arguments on the one hand, and on the other not to slow down the splitting process unnecessarily. Another condition limits the number of arguments added to A_1 by imposing a relative restriction on the added SCC's cardinality, i.e. if $|SCC| + |A_1| > |A| * 60\%$, the SCC will not be accepted. The algorithm runs recursively until no further arguments can be added (i.e. when $|optimal\_set| = |A_1|$).

On the basis of the set A_1 we can then compute the sets R_1, A_2, R_2 as well as the set of attacks along which the framework is split (R_3). The pseudo code for these operations is not included here due to their obvious simplicity.

The processing of the tuple AF_1 by a labelling algorithm may return an extension as part of a labelling, if it exists. This extension (E_1) will in turn be used for modifying AF_2 in the second part of the splitting algorithm. We start with the set A'_2 which is A_2 minus all the arguments in A_2 that are attacked by E_1, and we call it the modified set of A_2.

Next we apply the second algorithm on E_1, starting with an empty set, in order to compute a reduced set of undefined arguments (U_{E_1}). Note that we are not concerned with all the undefined arguments as stated in Def. 3.3, but only with those that are sources of an attack in R_3. Whenever an argument is a source of an attack in R_3, if it neither is an element of the extension E_1 nor is attacked by E_1, it will be added to the set U_{E_1}.

We then proceed to the final step in the modification of AF_2. Given U_{E_1}, for every argument of A'_2 which is attacked by U_{E_1}, a loop is added. By this addition, we have modified the set R_2. We call this modified set R'_2, and now we can define AF'_2 as the tuple (A'_2, R'_2). With the given definition, AF'_2 is to be processed by a labelling algorithm.

[12]The motivation behind this kind of splitting was that we expected a better performance in comparison with the *initial splitting*. However, as indicated in the experimental evaluation (Chapter 5) the results do not show a significant improvement. A reason for this is that equalization of sets poses an additional computational overhead to the already existing overhead of *initial splitting*.

Algorithm 4: Computation of Splitting, part 1

input : set of strongly connected components ($SCC(AF)$)
output : A_1, R_1, A_2, R_2, R_3

```
4.1  PROCEDURE compute_A1(SCC(AF))
4.2  begin
4.3      foreach SCC ∈ SCC(AF) do
4.4          scc_attacked := false
4.5          loop:
4.6          foreach a ∈ SCC do
4.7              foreach b s.t. (b, a) ∈ R do
4.8                  if b ∉ SCC then
4.9                      scc_attacked = true
4.10                     break loop;
4.11                 end
4.12             end
4.13         end
4.14         if scc_attacked = false then  add SCC to A1
4.15     end
4.16     return A1
4.17 end

4.18 PROCEDURE optimize(SCC(AF), A1)
4.19 begin
4.20     optimal_set := A1
4.21     illegal_attacks := false
4.22     foreach SCC ∈ SCC(AF) do
4.23         if |A1| < |A| * 0.45 then
4.24             pick an a ∈ SCC
4.25             if a ∉ A1 and |A1| + |SCC| < |A| * 0.6 then
4.26                 illegal_attacks = false
4.27                 loop:
4.28                 foreach a ∈ SCC do
4.29                     foreach (b, a) ∈ R do
4.30                         if b ∉ A1 and b ∉ SCC then
4.31                             illegal_attacks = true
4.32                             break loop;
4.33                         end
4.34                     end
4.35                 end
4.36                 if illegal_attacks = false then  add SCC to A1
4.37             end
4.38         end
4.39     end
4.40     if |A1| < |A| * 0.45 and |optimal_set| ≠ |A1| then
4.41         optimize(SCC(AF), A1)
4.42     end
4.43     return A1
4.44 end
```

Algorithm 5: Computation of Splitting, part 2

input : an extension of AF_1 (E_1), A_2, R_1, R_2, R_3
output: $AF'_2 = (A'_2, R'_2)$

5.1 $compute\_modified\_A_2(E_1, A_2, R_3)$
5.2 $compute\_U_{E_1}(E_1, R_1, R_3)$
5.3 $compute\_modified\_R_2(U_{E_1}, R_2, R_3)$

5.4 **PROCEDURE** $compute\_modified\_A_2(E_1, A_2, R_3)$
5.5 **begin**
5.6 $A'_2 := A_2$
5.7 **foreach** $a \in E_1$ **do**
5.8 **foreach** $(a, b) \in R_3$ **do**
5.9 **if** $b \in A'_2$ **then** remove b from A'_2
5.10 **end**
5.11 **end**
5.12 **return** A'_2
5.13 **end**

5.14 **PROCEDURE** $compute\_U_{E_1}(E_1, R_1, R_3)$
5.15 **begin**
5.16 $U_{E_1} := \varnothing$
5.17 **foreach** $(a, b) \in R_3$ **do**
5.18 **if** $a \notin E_1$ *and* $\nexists c \in E_1$ *s.t.* $(c, a) \in R_1$ **then** add a to U_{E_1}
5.19 **end**
5.20 **return** U_{E_1}
5.21 **end**

5.22 **PROCEDURE** $compute\_modified\_R_2(U_{E_1}, R_2, R_3)$
5.23 **begin**
5.24 $R'_2 := R_2 - \{(x, y) | (x, y) \in R_2$ and $(x \notin A'_2$ or $y \notin A'_2)\}$
5.25 **foreach** $a \in U_{E_1}$ **do**
5.26 **foreach** $(a, b) \in R_3$ **do** add (b, b) to R'_2
5.27 **end**
5.28 **return** R'_2
5.29 **end**

The application of splitting to framework $AF_{4.7}$ renders the following sets (Fig. 4.15):

- $AF_1 = (\{0, 1, 6, 9\}, \{(0, 1), (1, 0), (6, 9)\})$ (brown)
- $AF_2 = (\{5, 7, 8\}, \{(5, 7), (7, 8)\})$ (blue)
- $R_3 = \{(1, 5), (6, 5), (9, 8)\}$ (gray)

The result of computing the grounded labelling is given in Fig. 4.16. Several things have changed compared with the original framework, as shown in Fig. 4.8:

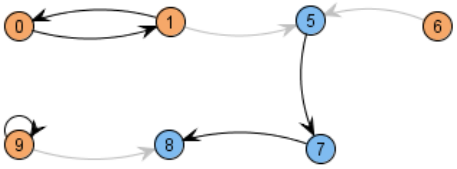

Fig. 4.13: $AF_{4.7}$ (with splitting indicated)

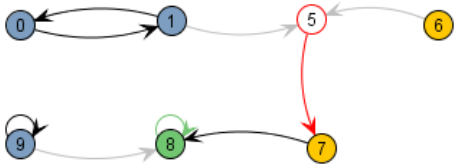

Fig. 4.14: $AF_{4.7}$ (gr. lab. with spl.)

- Argument 5 was removed as is signaled by white background and red edging. It is due to the fact that 5 is attacked by the extension (argument 6) which requires application of the reduct operation.
- Attack $(5, 7)$ was removed as signaled by its red coloring.
- Argument 8 received a self-loop. It is because argument 9 is undefined. [13]

Note that 5 does not have a self-loop although it is attacked by the undefined argument 1. It has to do with the order of applying both *reduct* and *modification*. Because *reduct* was used prior to *modification* we removed 5 before it could receive a self-loop.

To keep the section complete the splitting results for the preferred semantics are included below.

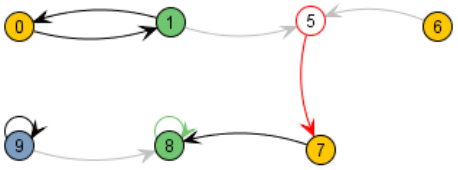

Fig. 4.15: $AF_{4.7}$ (with splitting indicated)

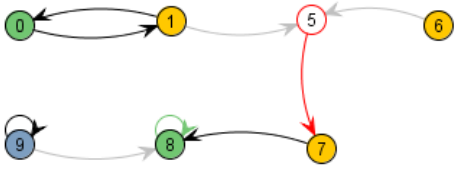

Fig. 4.16: $AF_{4.7}$ (gr. lab. with spl.)

4.4 Java Source Structure

The software was written in the Java programming language. One of the advantages of the language is that it allows for good modularization. The present project was divided into four packages, each of which has a different functionality. In the following sections I will describe shortly the structure of the program.

A general overview of dependencies between the packages is given in Fig. 4.17.

[13] Note that 8 is also attacked by the *IN* argument 7 which does not have any influence on 8 as 7 does not belong to AF_1.

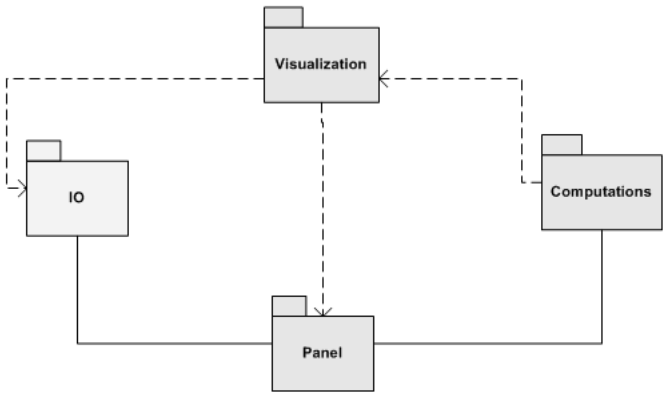

Fig. 4.17: Package Diagram

The package "Panel" represents the graphical user interface. As such it is associated with any external communication that a user can perform. The communication is concerned with importing and saving a file with framework data or exporting a framework image, etc. All this is organized by the package "IO". All computations are executed by the user via the GUI, hence the association with the package "Computations". Seen from the side of visualization, representation of a framework is dependent on the data received via "IO" as well as any user interaction on the panel. Any computation is dependent on the data represented graphically on the display (Package "Visualization").[14]

Package "Panel"

The Java package *Panel* contains three classes:

- Main.java
- FrameworkPanel.java
- HelpInstructions.java

Main.java is the class responsible for the startup of the application. It contains only one method which creates an instance of the FrameworkPanel class. A FrameworkPanel initializes the user interface and menus. It also keeps the memory of settings saved in the previous session by extending the *java.util.Observable* Interface. The class has several internal *listener* classes for the purpose of notifying various parts of the GUI of user initiated events like change of modes or layouts, and any kind of mouse, keyboard or button actions like for instance the computation of a labelling.

The class HelpInstructions as the name already suggests provides all the help related materials like the operation of the graph or the way of reading graphical and printed results.

[14]I decided upon this approach as splitting entails many operations directly on the graph. Hence most of information requests about arguments and attacks, adding and deleting nodes and edges, is performed by querying the graph directly.

Package "Visualization"

Here the most important elements of the graphic representation are located:

- Argument.java
- Attack.java
- Labelling.java
- FrameworkViewer.java

This part of the application was a novel one as it goes away from the fundamental Java libraries and ventures into the world of third party libraries which provide extended opportunities at the cost of time spent to learn a usually different conceptual approach. Most of the code of the FrameworkViewer class implements JUNG and Apache Commons functionality.

FrameworkViewer is the class steering the appearance on the display. It is responsible for creating arguments and attacks with help of *org.apache.commons.collections15's* argument / attack *factories*. The graph format used is an instance of JUNG *DirectedSparseGraph* class. An Apache Commons *transformer* dynamically translates an argument into its Point2D location. Yet another *transformer* (the ArrowDrawPaintTransformer) dynamically maps an attack into an arrow color or an argument into a fill color. JUNG's various *renderers* control for example the way an argument's label is displayed.

The two components of a graph - the argument and the attack - are characterized as follows. An arguments is identified by its name. Beyond that, each arguments has its

- *lowlink* and *index* number used for computing strongly connected components,
- fill and draw color responsible for its appearance,
- location coordinates specifying where on the display area it lies at a particular moment.

For an attack there is a name and the color. There is no need to save its source and target since those can be easily queried using methods provided by JUNG.

Package "IO"

Package "IO" contains classes responsible for external communication:

- FrameworkFileFilter.java
- FrameworkFileSelector.java
- FrameworkImageFilter.java

- RandomFrameworkDialog.java

FrameworkFileFilter and FrameworkFileSelector guide the user in selecting, opening, saving or discarding files with framework data. The filter is exclusively filtering out only those files that have an appropriate extension, i.e. .net or .aaf. The same does the FrameworkImageFilter to framework images exported from the editor. It indicates the supported formats which at present consist of .png, .jpg and .gif.

The RandomFrameworkDialog is the dialog windows that opens when in the File menu the command "Generate Random Framework" is chosen (see Fig. 4.2).

Package "Computations"

This package is the main component of the application. Algorithms for computation of strongly connected components, the splitting algorithm, and the three semantics algorithms are all located here:

- StronglyConnectedComponent.java
- Splitting.java
- GroundedSemantics.java
- PreferredSemantics.java
- StableSemantics.java

Since the working of these classes were presented in Chapter 2, here I am attaching only the class diagram indicating the connections between them (Fig. 4.18). For clarity, class attributes and methods are omitted.

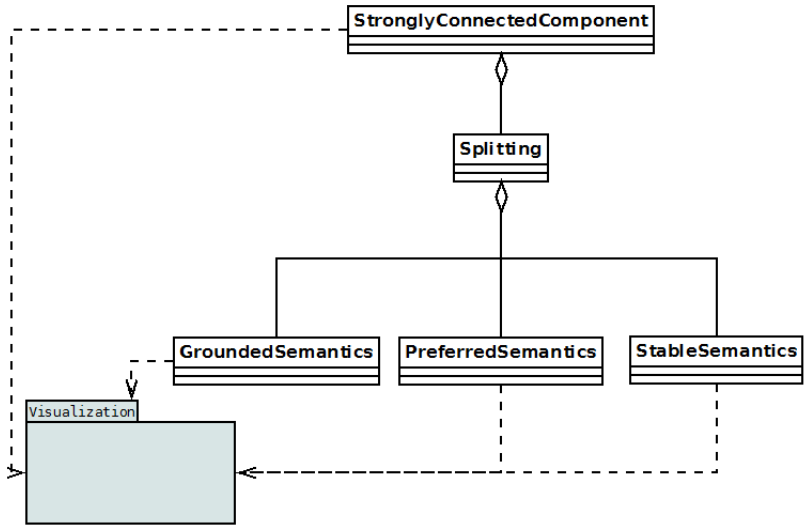

Fig. 4.18: Class Diagram

Computation with splitting: The class StronglyConnectedComponent.java receives its input directly from the package "Visualization". After finding the set of SCCs the splitting takes place. With the first part of splitting computed we can execute a semantics algorithm.

Computation without splitting: Any semantics execution depends directly on the inputs from the "Visualization" package.

Chapter 5

Experimental Evaluation

Our evaluation[15] of the runtime for grounded, preferred and stable semantics is based on the sampling of 100 randomly generated frameworks. The tests were performed on a Samsung P510 notebook with a Pentium Dual Core Processor, CPU speed: 2.0 GHz, CPU Caches: 32 KB (L1) and 1024 KB (L2), RAM: 2 GB. We focused in our experiments on frameworks where the number of attacks (n) exceeds the number of arguments (m) by a factor between 1.5 and 3.

The reasons for this restriction are as follows. First of all, even leaving execution times aside[16], by further increasing the number of attacks the probability of generating frameworks consisting of a single SCC grows, thus rendering the experiment inconclusive as splitting has no effect on AFs with a single SCC. For example, initial tests showed that if 500 or more attacks (n) are given for 100 arguments (m), then almost all of the randomly generated frameworks will consist of only a single SCC and no effect of splitting is to be expected.

Second, an even stronger limitation lies in the fact that we are bound to limit the number of attacks even more due to long execution time required for computing labellings in AFs when the number of attacks exceeds far too much the number of arguments. It is especially true of those frameworks which already contain large number of arguments since the larger the number of arguments is the longer it takes to compute labellings. For example, our preliminary testing showed that for AFs with 100 arguments, if 200 attacks are specified, the percentage of frameworks with runtime over 3 *min* for preferred semantics without splitting was about 70%.[17]

On the other hand, choosing an n smaller than m would not lead to significant differences in execution time between AFs with and without splitting as execution times tend to be fast under such conditions anyway.

With the above limitations in mind, a total of 100 examples were collected, with

[15] The evaluation presented here is an extended version of Section 4 ("Experimental Results") given in [5].

[16] For example, our preliminary testing showed that for AFs with 100 arguments, if 200 attacks are specified, the percentage of frameworks with runtime over 3 *min* for preferred semantics without splitting was about 70%.

[17] Out of 15 tests only 4 had runtime under 3 *min*.

20 examples extracted from each of the following m/n combinations: 10/30, 50/100, 100/175, 200/375 and 500/750. A brief description of the results obtained will be presented below together with a tabular summary of statistical data for each combination. Each table contains average-runtime results (in milliseconds) and gain-in-time results (in %)[18] for the grounded, preferred and stable semantics. Under "average runtime", the first column contains results from executing without splitting, the second from executing with non-optimized splitting and the third from executing with optimized splitting. Under "gain in time", minimal, maximal and average gain results, each in relation to non-optimized and optimized splitting, are distinguished.

The 10/30 combination was the only case in which we experienced no runtime that was over 3 min.[19] Thanks to the low number of arguments we were given a possibility of structural analysis. Although 20 examples is a small sample size, we were able to distinguish 4 characteristics based on the structure of the framework and the corresponding difference in runtime between executions without and with splitting. The analysis below applies to the preferred and stable semantics as the execution of the grounded semantics did not show any difference.

First, in 3 cases out of 20 a single SCC was generated. As splitting has no effect on AFs consisting of a single SCC, there was no runtime improvement for all 3 semantics. However, no noticeable runtime delay in relation to the splitting process was recorded either.

Second, 3 further examples had the form of a single argument SCC attacking a large SCC (A). Here we recorded no improvement or only a slight improvement in the runtime when splitting was applied: 0-20%.

An example:

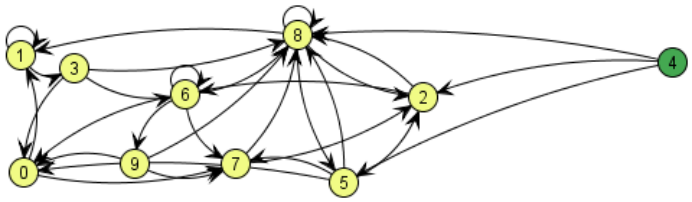

Fig. 5.1: Framework structure A (SCC {4} attacks the rest of the framework)

Third, yet 3 further cases consisted of a single argument SCC with a self-loop attacking a large SCC (B). The only difference regarding the single argument between this form and the previous one was that we now had a loop attack. However in terms of runtime the gap was significant. In the second case it was between 68-71% for preferred semantics and between 99-100% for stable semantics.

An example:

[18]For convenience, in the presented data we use "0 ms" to mean "close to 0 ms" and "100%" to mean "close to 100%".

[19]It comes as no surprise since the computation of preferred labellings for an AF with 10 arguments and 100 attacks takes around 260,000 ms

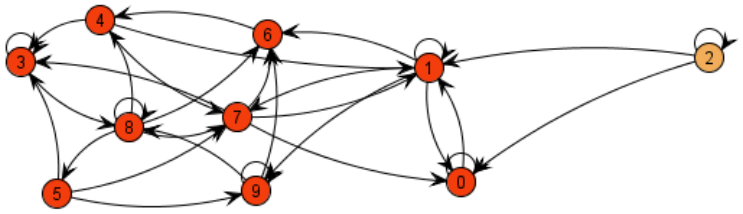

Fig. 5.2: Framework structure B (SCC {2} attacks the rest of the framework)

And last, 11 of the random AFs had the form of a larger SCC attacking a single argument SCC (C1), a single argument SCC with a self-loop (C2) or two SCCs (C3); or the form of two SCCs, with at least one attack each, attacking the rest of the framework (C4). The difference in execution without and with splitting ranged here between 80-99% for preferred semantics and between 59-100% for stable semantics.

Examples:

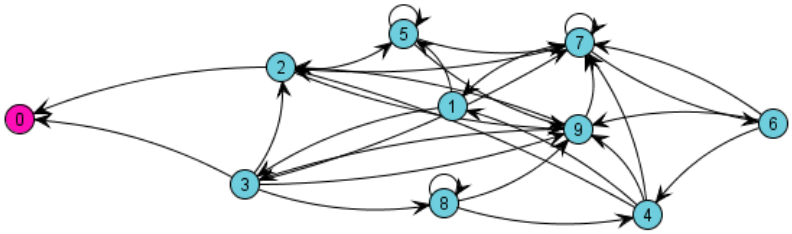

Fig. 5.3: Framework structure C1 (SCC {0} is attacked by the rest of the framework)

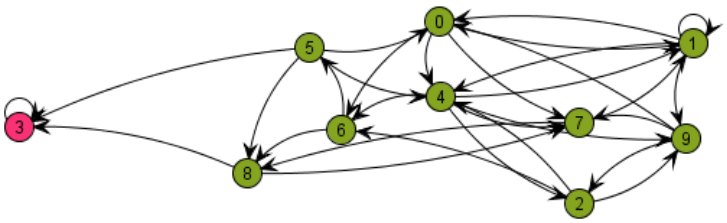

Fig. 5.4: Framework structure C2 (SCC {3} is attacked by the rest of the framework)

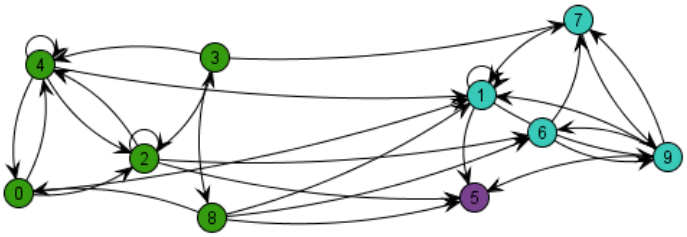

Fig. 5.5: Framework structure C3 (SCC $\{0, 2, 3, 4, 8\}$ attacks $\{5\}$ and $\{1, 6, 7, 9\}$)

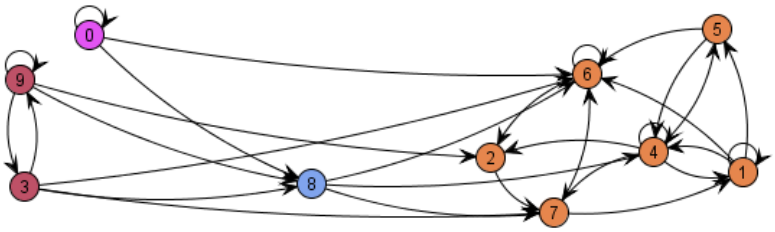

Fig. 5.6: Framework structure C4 (SCCs $\{0\}$ and $\{3, 9\}$ attack the rest of the framework)

The limited data suggest that splitting can render computation significantly faster for frameworks with certain characteristics. It seems that the most relevant are those AFs having one or more SCCs, each with at least one attack (i.e. a single argument SCC with a loop or an SCC with at least 2 arguments), attacking one or more SCCs whose structure in itself is not relevant.

An additional test on an AF of 10 arguments, of which 9 constituted an SCC with 81 attacks and all 9 attacked the 10th argument (Fig. 5.8), recorded a 90% runtime difference for both preferred and stable semantics. Alike, 90% gain was obtained after adding a self-loop to the SCC being attacked (Fig. 5.7). These additional results lie nicely within the ranges of the previously obtained 80-99% and 59-100% respectively.

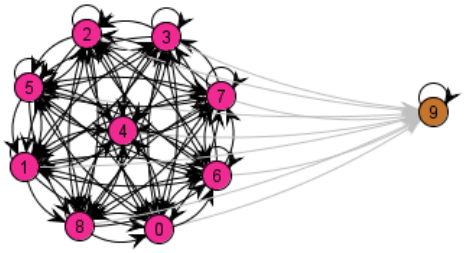

Fig. 5.7: 90% gain for preferred and stable

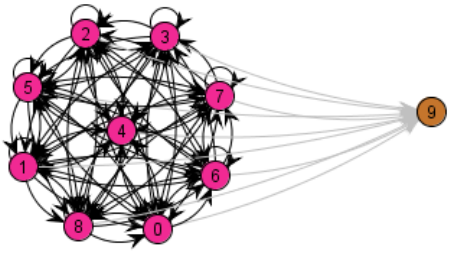

Fig. 5.8: 90% gain for preferred and stable

A further test of a single argument with a self-loop attacking each argument of an SCC with 9 arguments and 81 attacks (Fig. 5.9) showed a 90% runtime difference for preferred semantics and 100% for stable semantics. The performance was evidently better than the previously obtained result for preferred semantics (68-71%). Having removed the loop attack (Fig. 5.10) we obtained a runtime of 1 *ms* for preferred and stable semantics, both with and without splitting. Again, these results are also in compliance with the ones obtained in the sample test using 20 examples.

In general we obtained an average acceleration of 60% for both types of splitting in comparison to an execution without splitting. It is partly due to the fact that for the 10/30 combination both non-optimized splitting and optimized splitting usually overlap, which in turn is a result of the existence of large SCCs that limits the possibility of having different splittings. In no case was the execution with splitting slower than the one without.

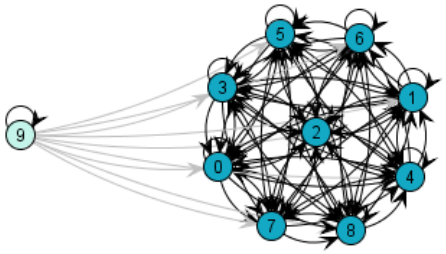

Fig. 5.9: 90% gain for pref., 100% for stable

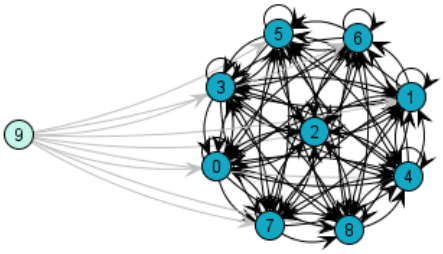

Fig. 5.10: 0% gain for preferred and stable

Tab. 5.1: Evaluation results for 10 arguments and 30 attacks

m = 10	**average runtime (in ms)**			**gain in time (in %)**					
n = 30	**w/o spl.**	**w/ spl.**	**opt. spl.**	**min**	**min/op**	**max**	**max/op**	**avg**	**avg/op**
grounded	1	1	1	0	0	0	0	0	0
pref.	3871	886	890	0	0	99	99	60	61
stable	1040	267	262	0	0	100	100	59	60

The runtimes for the 50/100 combination were very diversified: from 1 *ms* (for stable) and 2 *ms* (for preferred semantics) to 381,512 *ms* (preferred)[20] and 4,456 *ms* (stable). The grounded labelling was computed at the speed of 1-3 *ms* in each case, no improvement nor delay was recorded for executions with splitting in comparison to those without.

In 9 out of the 20 cases, the computation time for preferred and stable labellings without and with splitting was very short (below 20 *ms*). No significant difference was observed. The time gain for these cases was given as 0%, which had a negative effect on the average gain in time as shown in Table 5.2: it dropped to only 26-29%. Note that the maximal gain in time for both semantics was at 99%.

For the stable semantics we observed dramatic improvements in cases where no labellings existed. Through splitting of the framework, the time needed to find the first argument of the *undec* set, hence breaking the execution of the labelling algorithm, was at times very short. In 8 out of 15 cases where no labelling existed, the execution times lay below 20 *ms* which as mentioned above had 0% gain. Among the remaining 7 cases, 2 recorded an improvement of 99%, the rest between 17-75%. In none of the 20 examples was the execution without splitting faster than the one with splitting. Neither significant improvement nor delay was found for optimized splitting as compared to regular splitting.

Tab. 5.2: Evaluation results for 50 arguments and 100 attacks

m = 50	**average runtime (in ms)**			**gain in time (in %)**					
n = 100	**w/o spl.**	**w/ spl.**	**opt. spl.**	**min**	**min/op**	**max**	**max/op**	**avg**	**avg/op**
grounded	2	2	2	0	0	0	0	0	0
pref.	35860	23237	23352	0	0	99	99	29	26
stable	663	487	480	0	0	99	99	29	29

[20]This example had already been included in the data before the imposition of the 3-minute limit, and so this is the only example with a runtime above 3 *mins*.

Some 40% of the frameworks generated with 100 arguments and 175 attacks had a computation time of at least 3 *min* for the preferred semantics without splitting. They were not taken into consideration for the reason stated at the beginning of this section. In the collected examples, the runtimes varied from around 20 *ms* to slightly below 40,000 *ms*. No stable labelling existed in 19 out of the 20 examples. In 9 out of these 19 examples, we obtained an improvement of 90-100% for the stable semantics and 0-50% for the remaining 10. No slow down due to the process of splitting was noticeable.

Here, for the first time, we recorded a significant improvement in runtime when the optimized version of splitting was applied. It was 13% for the preferred semantics and 5% for the stable semantics, both of which were better than the non-optimized variant. On average, an execution with splitting was better than one without splitting by 56-69% for the preferred semantics and by 60-65% for the stable semantics.

Tab. 5.3: Evaluation results for 100 arguments and 175 attacks

m = 100	**average runtime (in ms)**			**gain in time (in %)**					
n = 175	**w/o spl.**	**w/ spl.**	**opt. spl.**	**min**	**min/op**	**max**	**max/op**	**avg**	**avg/op**
grounded	2	2	2	0	0	0	0	0	0
pref.	8335	3701	2502	0	0	93	99	56	69
stable	499	297	262	0	0	100	99	60	65

The computation time for preferred and stable labellings without splitting in frameworks of 200 arguments and 375 attacks was in general above 15 *ms*, thus making a more precise comparison possible. All the generated AFs showed a runtime improvement of at least 14% (pref.) and 26% (stable) when the execution with splitting is compared to the execution without splitting. Here too the gain in time reached in some cases 99% for the preferred labellings and 96% for the stable labellings.

With an average runtime of 3 *ms* for the grounded semantics, no difference between execution without and with splitting was found. The computation of stable labellings with applied splitting took on average 56% less time than that without. For the preferred semantics, the gain was somewhat less, it was 45% with optimized splitting and 47% with non-optimized splitting.

Tab. 5.4: Evaluation results for 200 arguments and 375 attacks

m = 200	**average runtime (in ms)**			**gain in time (in %)**					
n = 375	**w/o spl.**	**w/ spl.**	**opt. spl.**	**min**	**min/op**	**max**	**max/op**	**avg**	**avg/op**
grounded	3	3	3	0	0	0	0	0	0
pref.	9333	6531	6296	14	16	99	98	47	45
stable	352	236	222	26	26	96	93	56	56

It was relatively comfortable testing the 500/750 combination since only about 20% of the randomly generated frameworks had a runtime above 3 *min* for preferred labellings without splitting. The execution time was quite steady. The lowest runtime for preferred semantics without splitting was 53 *ms* and 58 *ms* for stable semantics without splitting. The absence of drastic highs and lows was mirrored in all the average runtimes for preferred semantics, which were much lower than the average runtimes

measured for 200/375. Here we observed also a steady improvement after splitting was applied. The lowest of which was 35% for preferred semantics and 33% for stable. The upper range was also less drastic with up to 86% for preferred and 97% for stable. The average differences were quite high with 57-61% for preferred labellings and 62-66% for stable. There was a drop in efficiency for the optimized type of splitting as compared to the non-optimized type (by 4% for both preferred and stable labellings). However, in AFs with a runtime above 700 ms, the optimized type ran faster than the one without optimization. In no case though was an execution with splitting slower than the one without splitting.

While in frameworks with 200 arguments and lower the grounded semantics did not perform worse after splitting, here we observed a visible slowdown. There was an average loss of 2% in the case of the non-optimized variant and an average loss of 36% in the case of the optimized variant.

Tab. 5.5: Evaluation results for 500 arguments and 750 attacks

m = 500	**average runtime (in ms)**			**gain in time (in %)**					
n = 750	**w/o spl.**	**w/ spl.**	**opt. spl.**	**min**	**min/op**	**max**	**max/op**	**avg**	**avg/op**
grounded	10	10	13	-12	-60	15	-15	-2	-36
pref.	2785	1697	1168	36	35	86	78	61	57
stable	232	120	99	33	47	97	89	66	62

Chapter 6

Conclusion

Based on our evaluations of 100 randomly generated AFs, we have made the following observations:

1. Among the 100 AFs, we observed an average improvement by 50-51% and by 54% for preferred and stable semantics respectively. The data contained some inconclusive examples which had "marred" the results to some extent.

2. No instance, neither for preferred semantics nor for stable, was found in which the execution with splitting lasted longer than the one without. This shows that the additional overhead introduced by splitting is negligible.

3. The optimized type of splitting did better than the non-optimized type in cases when the AF without splitting had a relatively long runtime. When the runtime was relatively short, the type without optimization usually performed better. The reason for that is probably the additional overhead introduced by the optimization process. On the other hand, this would also suggest that factors other than the size of the two sets after the partition may play an important role in the performance of splitting. Our expectation is that this factor is the internal structure we talk about in point 5.

4. Splitting may significantly improve runtime for stable semantics in frameworks where no stable labellings exist. By splitting the framework, we were able to complete the execution of the algorithm a lot faster because it took less time to find a labelling with the *undec* set that was not empty.

5. It seems that there exist certain regularities between the structure of frameworks and the corresponding runtime. Having an SCC with at least one attack (or several SCCs with at least one attack each) attacking the rest of the framework can improve runtime significantly. We especially hope that this will greatly affect computation of large frameworks with large SCCs, which so far we were unable to test due to the required long computation time.

The largest argumentation framework tested contained 500 arguments and 750 attacks. It would be interesting to test whether the results can be translated into frameworks of much larger sizes. That would however require a much faster processor to test

on. Some frameworks with just 100 arguments and 200 attacks run for over 30 mins. without conclusion (for preferred semantics). Those runs had to be stopped and were therefore not tested.

Bibliography

[1] Leila Amgoud and Claudette Cayrol. On the acceptability of arguments in preference-based argumentation. In *Proc. Fourteenth Conference on Uncertainty in Artificial Intelligence, UAI-98*, pages 1–7, 1998.

[2] Pietro Baroni and Massimiliano Giacomin. Semantics of abstract argument systems. In Iyad Rahwan and Guillermo R. Simari, editors, *Argumentation in Artificial Intelligence*, pages 25–44. Springer, 2009.

[3] Pietro Baroni, Massimiliano Giacomin, and Giovanni Guida. Scc-recursiveness: a general schema for argumentation semantics. *Artif. Intell.*, pages 162–210, 2005.

[4] Ringo Baumann. Splitting an argumentation framework. In *Proc. LPNMR-11, to appear*, 2011.

[5] Ringo Baumann, Gerhard Brewka, and Renata Wong. Splitting argumentation frameworks: An empirical evaluation. accepted for TAFA-11.

[6] Trevor J. M. Bench-Capon, Sylvie Doutre, and Paul E. Dunne. Value-based argumentation frameworks. In *Artificial Intelligence*, pages 444–453, 2002.

[7] Gerhard Brewka and Stefan Woltran. Abstract dialectical frameworks. In *Proc. KR-2010*, pages 102–111, 2010.

[8] Martin Caminada. Semi-stable semantics. In *Proc. Computational Models of Argument, COMMA-06*, pages 121–130, 2006.

[9] Martin Caminada and Dov Gabbay. A logical account of formal argumentation. In *Studia Logica*, pages 347–374. Springer, 2009.

[10] Apache Commons. *http://commons.apache.org*.

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

[12] Phan Minh Dung, Paolo Mancarella, and Francesca Toni. A dialectic procedure for sceptical, assumption-based argumentation. In *Proceeding of the 2006 conference on Computational Models of Argument: Proceedings of COMMA 2006*, pages 145–156, Amsterdam, The Netherlands, The Netherlands, 2006. IOS Press.

[13] Java Universal Network/Graph Framework. *http://jung.sourceforge.net*.

[14] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In *ICLP*, pages 23–37, 1994.

[15] Sanjay Modgil and Martin Caminada. Proof theories and algorithms for abstract argumentation frameworks. In Iyad Rahwan and Guillermo R. Simari, editors, *Argumentation in Artificial Intelligence*, pages 105–132. Springer, 2009.

[16] John L. Pollock. Justification and defeat. *Artificial Intelligence*, 67:377–407, 1994.

[17] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.

Ort	Datum	Unterschrift