Imperial College London

Interim Report

Web-Based Argumentation

Author:
Michael Procopiou

Supervisors:
Dr. Xiuyi Fan
Prof. Francesca Toni

June 9, 2014

Contents

1 Introduction **1**
- 1.1 Incorporating derivation engines in a web application. 2
- 1.2 Visualising the derivation trees. 3
- 1.3 Building a Context Free Grammar for the input of the web application. 4
- 1.4 Grounding a framework over a domain. 4
- 1.5 Implementing derivations using Ideal Semantics. 5

2 Background **7**
- 2.1 Argumentation 7
- 2.2 Assumption-Based Argumentation 7
- 2.3 The ABA framework 8
- 2.4 Proving a claim using ABA 10
- 2.5 Computational mechanisms for ABA 11
- 2.6 Proxdd 12
- 2.7 Grapharg 13
- 2.8 ASPARTIX 15
- 2.9 Implementing Ideal semantics dispute derivations 16
- 2.10 Decision Making with ABA 18
- 2.11 Mapping AA to ABA 20

3 System Overview. **22**
- 3.1 Providing a web application based interface for derivation engines. 22
  - 3.1.1 Making ABA more accessible. 22
  - 3.1.2 Common gateway to various engines. 22
- 3.2 Flow of the System. 23
  - 3.2.1 Flow overview and Diagram. 23
  - 3.2.2 Overview of functionalities. 24
- 3.3 System Design choices. 27
  - 3.3.1 Plug—In and Plug—Out. 27

3.3.2 Need for Generator functions. 27

4 Creating an Input Language. 28

4.1 Creating a Context Free Grammar for the Input. 28

4.2 The need for a new input. 28

4.2.1 Formal definition of Input. 29

4.3 Parsing in the input. 30

4.3.1 Building a simple parser for the language. 31

4.3.2 Verifying the correctness of the input. 32

4.3.3 Forcing the existence of a contrary. 33

4.4 Error checking and feedback for user. 33

4.4.1 Detecting an Invalid Statement. 34

4.4.2 Validate form of user defined target. 34

4.4.3 Validate the existence of a target in the grounded framework. 35

4.4.4 Check if derivation for target was found. 36

4.5 Example of an input. 37

5 Grounding a framework over its domain. 40

5.1 The need of specifying a domain for a framework. 41

5.2 Implementing a grounder. 41

5.2.1 Description of the algorithm. 42

5.2.2 Advantages of implementation. 47

5.2.3 Disadvantages of implementation. 48

5.3 Example of expected output. 49

6 Implementing Ideal Semantics dispute derivations. 51

6.1 Extending Proxdd to include ideal semantics. 51

6.2 Implementing ideal semantics. 52

6.3 Example of ideal derivation. 57

7 Building a web application interfacing the derivation engines. 58

7.1 Overview of the Solution. 58

7.1.1 Client 58

7.1.2 Server 59

7.1.3 Derivation Engines 59

7.2 User Interface with Application. 59

7.2.1 Setting up the framework. 60

7.2.2 Configuration of the engines. 60

7.2.3 Visualisation the derivation tree. 61

7.3 Server - Processing the Input. 62
7.4 Server - Interfacing with the Prolog Engine. 63
7.4.1 Need for an interface between C# and Prolog Engines. 63
7.4.2 Using the SwiPIC.dll library as an interface. 64
7.5 Choosing the current implementation. 64
7.5.1 Advantages of current implementation. 65
7.5.2 Disadvantages of current implementation. 65
7.5.3 Proposed alternative (Prolog Web Server). 66

8 Evaluation of the System. 68
8.1 Evaluating GUI Usability. 68
8.1.1 User testing and feedback. 68
8.1.2 Initial Concept. 69
8.1.3 Changes to GUI. 70
8.2 Evaluating the Grounder. 72
8.2.1 Checking validity through examples. 73
8.2.2 Random Framework generator. 74
8.2.3 Performance of Grounder. 77
8.3 Evaluating the ideal semantics implementation. 77
8.3.1 Testing in Swi—Prolog. 77

9 Conclusions 84
9.1 What comes next? 84
9.1.1 Building on top of the Web—Application. 84
9.1.2 Real World Application. 85
9.2 Concluding remarks. 86

Abstract

Assumption Based Argumentation (ABA) frameworks can be used to represent a knowledge based systems or decision making problems. Through various dispute derivation processes one can then deduce the validity of a conclusion they have suggested, in accordance to the framework defined.

In this project we propose the implementation of a web application allowing the users to define ABA frameworks on-line and then carry out the necessary derivation of the conclusions they specify. We suggest the a flexible implementation allowing the application to be considered as a middle-ware solution on top of which further application can be developed. Additionally, the system abstracts its functionality from the derivation engines allowing for the support of an array of different argumentation derivation engines.

We begin by defining a unified input mechanism, in the form of a formally defined language, that allows for the definition of ABA frameworks irrespective of the underlying derivation engine. A parser was built for the input language that interprets and validates the input.

Additionally, we enable the definition of a domain over which a framework exists. This enables us to ground frameworks over the domain using a grounder we have developed. This extends the use of ABA frameworks by accounting for domains and variables.

Furthermore, we extend an existing derivation engine to accommodate dispute derivations based on ideal semantics, thus further extending the diversity of choices available to the user.

Lastly, the web application in its entirety is thoroughly evaluated through a serious of test runs with predefined examples.

Acknowledgements

I would like to sincerely thank Dr Fan Xiuyi for his constant support and guidance during the duration of this project, along with his much appreciated advice and feedback. I would also like to extend special thanks to Professor Francesca Toni for her invaluable support and constructive feedback throughout the extend of the project.

Special thanks to Professor Krysia Broda for her advice and guidance in accordance to the interim report.

Lastly, I would like to extend my warmest thanks to my parents, my sister and my friends for their much appreciated and persistent guidance, support, patience and love.

Without these people the completion of this project would not have been possible.

Chapter 1

Introduction

Argumentation is increasingly becoming a popular aspect of Artificial Intelligence that deals with the creation of knowledge systems and evaluation of decision problems. Past and current research has allowed for the development of various argumentation frameworks that allow the analysis of arguments by formulating them in a formal manner and evaluating them. This project focuses mostly on Assumption-Based Argumentation (ABA) and its implementation through a practical argumentation system. ABA is a form of argumentation where arguments and attacks are notions derived from primitive notions of rules in a deductive system, assumptions and contraries thereof [9]. This framework allows for the evaluation of a conclusion based on whether it is supported by a "winning" set of arguments. Such sets of arguments and assumptions can be determined as "winning/acceptable" based on a number of different semantics that ABA supports, which are discussed in more detail later on in the report.

Therefore, ABA is a potentially powerful tool which one could use to establish the validity of the conclusion put forward. Early successful application of argumentation theory and the ABA framework have occurred in fields such as legal-reasoning, medical diagnosis and decision theory. Within these fields ABA has been used not only because of its ability to determine propositions as acceptable, but also due to its ability to convey the derivation process to the user. There are various computational mechanisms that have been devised to algorithmically compute the acceptability of a claim. Using these mechanisms argumentation engines, such as proxdd [8] and grapharg [3], have been implemented thus allowing for the computation of acceptability of a claim under a defined ABA framework. Nonetheless, these engines are still at a primitive stage and require enhancing for ABA to become a widely accepted and applied tool in the real world.

Namely, there is a lack of an application that is easily accessible to users and provides both satisfactory performance and useful visualisations of the argument. The difficulties involved with creating such an application are multifold. These involve the performance of the underlying engine when dealing with large scale real problems, the difficulty a user might face in devising a correct ABA framework for their problem and devising a visualisation system to accurately convey the derivation process to the user. Additionally, the engines themselves are still at an early stage and need to be extended. For example the proxdd and grapharg systems lack the ability to derive the acceptability of a claim based on "ideal" semantics. Lastly, there is a lack of generalisation of these systems. At their current state they aim at simply resolving ABA frameworks directly implemented into the engine. This restricts the usability of the system to users who intend to use just ABA and have significant knowledge of ABA to device the initial framework of their problem.

The project's objectives centre around providing solutions to the problems mentioned above by fulfilling the requirement for a web-based argumentation application that will allow users to exploit the existing ABA systems. The web-application should act as a platform that will provide the users with good user experience and the ability to easily and as seamlessly as possible interact with the ABA systems in the back end. Having computed the necessary derivation the user will be provided with the output in a useful and meaningful manner in the form of a visualisation. The end system would be similar to the ASPARTIX system implemented by TU Wien [6]. Having established the web-application, the project will then seek to enhance the existing systems by enabling them to compute based on more semantics mentioned above. Potentially the project might look in the development of a simple API for these engines. Lastly, the web-application might be extended to support Abstract Argumentation through its mapping to ABA (as described by [7]) and allow the direct input of decision problems (as described in [11]).

1.1 Incorporating derivation engines in a web application.

We looked into designing a web application that would provide an online interface to the derivation engines. This includes a clean and simple interface by which the user is able to choose both the engine and the semantics required and would also allow them to define their ABA framework. Figure [TODO] provides a screen shot of the GUI used by the user to interact with the derivation engines. The implementation and design choices are detailed in sections [TODO].

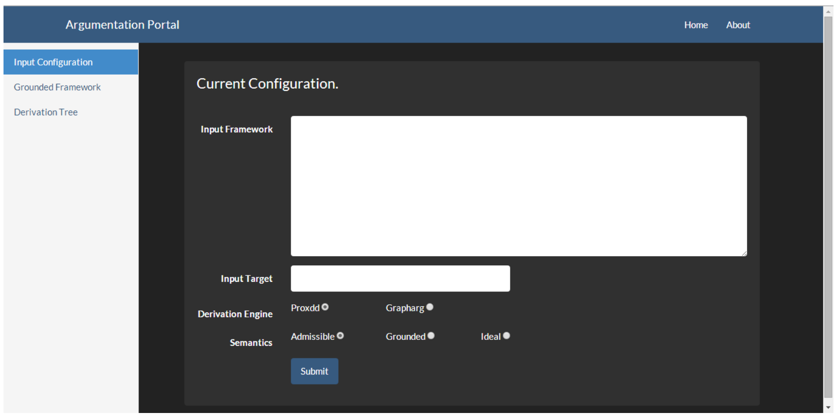


Figure 1.1: Input configuration panel.

1.2 Visualising the derivation trees.

We then looked into a suitable output method of the solution received from the derivation engines. This involves the visualisation of the derivation tree on a canvas environment as illustrated in figure [TODO]. The visualisation provides a clear and concise method for the user to evaluate the derivation of the claim they submitted. The implementation of the visualisation is discussed further in section [TODO].

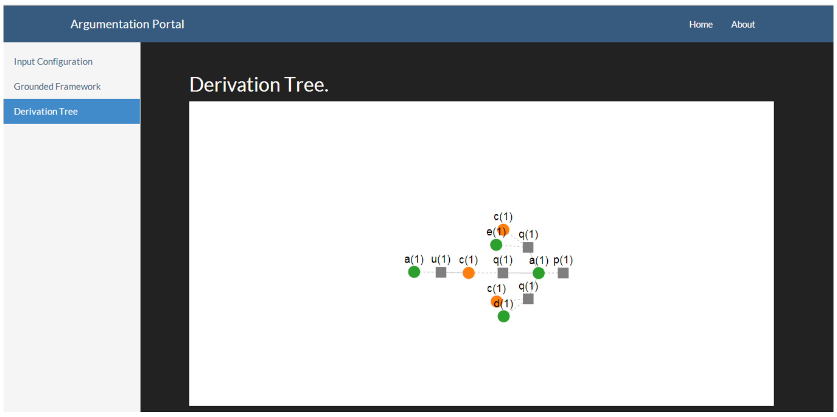


Figure 1.2: Derivation Tree panel.

1.3 Building a Context Free Grammar for the input of the web application.

We formally defined a Context Free Grammar for the expected user input and built a parser for it. An example of a valid input is given in [TODO].

```
asm(a(X),q(X)){X=1,2;}.
asm(b(X),f(X)){X=1,2;}.
asm(c(X),u(X)){X=1,2;}.
asm(d(X),v(X)){X=1,2;}.
asm(e(X),v(X)){X=1,2;}.
asm(f(X),v(X)){X=1,2;}.

p(X)<-[a(X),u(X)].
q(X)<-[b(X),r(X)].
q(X)<-[c(X),s(X)].
q(X)<-[c(X),t(X)].
u(X)<-[a(X)].
s(X)<-[]{X=1,2;}.
t(X)<-[d(X)].
t(X)<-[e(X)].
```

In section [TODO] we elaborate on the semantics of our input language, along with how the parser forced the validity of the user's input.

1.4 Grounding a framework over a domain.

We then extended the language to allow for the specification of a domain over which assumptions, contraries or rules of our framework are valid. In example [TODO] we see how the notation is used to define a domain and the viable variable values.

```
asm(a(X,Y),b(X)){X=1,2,3;Y=a,b;}.
b(X)<-[]{X=1;}.
```

We also built a grounder that grounds the framework over the domain specified by the user. The grounder acts as a pre-processing step and allows us to generate the grounded version of our framework as in example [TODO]

```
asm(a(1,a),b(1)). asm(a(1,b),b(1)).
asm(a(2,a),b(2)). asm(a(2,b),b(2)).
asm(a(3,a),b(3)). asm(a(3,b),b(3)).

b(1)<-[].
```

An overview of the impementation of the grounder is provided in section [TODO]

1.5 Implementing derivations using Ideal Semantics.

Using the existing implementation of the admissible based dispute derivations in Proxdd, we introduced the necessary functionality that enables us to derive dispute derivations based on ideal semantics. This includes the introduction and correct updating of the "F" set , along with the implementation of the Fail(S) check as described in section [TODO]. An example of a successful dispute derivation under the ideal semantics is provided in example [TODO].

Example 1. Consider the following assumption based framework:

R - includes the following rules:

z←a
z←b
y←a
x←d
v←c

A = {a,b,c,d} and $\bar{a} = z, \bar{b} = y, \bar{c} = x, \bar{d} = v$.

Step	P	O	A	C	F
1	{z}	{}	{}	{}	{}
2	{b}	{}	{b}	{}	{}
3	{}	{{x}}	{b}	{}	{}
4	{z}	{}	{b}	{a}	{{a}}
5	{}	{}	{b}	{a}	{{a}}
6	{}	{}	{b}	{a}	{}

Table 1.1: Example of IB-derivation of example [TODO]

The details of the implementation are explored in section [TODO].

Chapter 2

Background

2.1 Argumentation

Argumentation in general involves studying how claims and conclusions can be reached by applying logical reasoning. It is useful when analysing arguments in the form of dialogue and persuasion, especially under the context of philosophy, medicine and law. Research in artificial intelligence is looking to exploit the underlying logic in analysing arguments to allow for this analysis to be carried out in a computerised manner. In order to do so general purpose argumentation frameworks have been devised to allow for analysis of an argument as a computerised argumentation problem. Two such argumentation frameworks are Abstract Argumentation (AA) and Assumption-Based Argumentation (ABA). Argumentation frameworks such as these can provide the general basis required to implement specific frameworks for real-life problems where argumentation could help in decision making and argument analysis. For the web-based application proposed by this project the underlying framework will be ABA.

2.2 Assumption-Based Argumentation

Although ABA is an instance of AA, there are core differences in the way ABA perceives and evaluates arguments. In ABA arguments and attacks are derived from given rules in a deductive system, assumptions and their contraries [9]. An argument is supported by assumptions and is attacked by other arguments when the contrary of an assumption of the initial argument can be deduced by the opposing argument. However, in order for an argument to be deemed “acceptable” or “winning”, ABA is equipped with numerous semantics that allow us to determine an “acceptable/winning” set of assumptions, from which we can deduce an “acceptable” set of arguments. Additionally several computational mechanism have been developed for ABA (see [8] and [3]) that allow a conclusion or claim to be algorithmically evaluate in terms of a “acceptable” set of arguments. The computational nature of these mechanisms allow us to programmatically implement the semantics and allow for computerised argumentation analysis of a framework.

2.3 The ABA framework

The potential power of ABA lies with its ability to represent and evaluate real-world decision problems. An implementation of ABA could prove to be an invaluable tool when it comes to evaluating potential claims, providing support to existing claims or disputing existing claims, in any context as long as it can be generalised and implemented as part of the ABA framework.

The following is an example taken from [9] to demonstrate how ABA can interpret a simple real-world problem, while introducing the ABA framework and its elements.

“You like eating food, this makes you really happy. But you are very health-conscious too, and don’t want to eat without a fork and with dirty hands. You are having a walk with some friends, and your friend Nelly is offering you all a piece of lasagne, looking really delicious. You are not quite sure, but typically you carry no cutlery when you go for walks, and your hands will almost certainly be dirty even if the may not look so. Should you go for the lasagne, trying to snatch it quickly from the other friend? You may argue with yourself as follows:

α : let me go for the lasagne, it looks delicious, and eating it will make me happy!

β : but I am likely to have no fork and my hands will be dirty, I don’t want to eat it in these circumstances, let the others have it!

α and β can be seen as arguments, and β disagrees with (attacks) α. If these are all the arguments put forward, since no argument is proposed that defends α against β , the decision to go for the lasagne is not dialectically justified. If, however, you put forward the additional argument

γ : who cares about the others, let me get the lasagne, I may have a fork after all

This provides a defence for α against β, and the decision to go for the lasagna becomes dialectically (although possibly not ethically) justified.”

This example is illustrative of the breadth of problems that could potentially be analysed by an ABA system. Granted the example is overly simplistic. It has a limited amount of arguments and accounts for just a few of the possible parameters that could affect whether we are happy or not. Nonetheless, being a general framework the simplicity of the problem does not affect our ability to device a specific framework for this problem from which we can evaluate our claim of being happy (or not being happy).

The ABA frame work is defined by [9] as follows:

Definition 1. An ABA framework is a tuple $\langle \mathcal{L}, \mathcal{R}, \mathcal{A}, \bar{\ } \rangle$ where

- $\langle \mathcal{L}, \mathcal{R} \rangle$ is a deductive system, with $\mathcal{L}$ the *language* and $\mathcal{R}$ a set of *rules*, that we assume of the form $\sigma_0 \leftarrow \sigma_1, \ldots, \sigma_m (m \geq 0)$ with $\sigma_i \in \mathcal{L}(i = 0, \ldots, m)$; σ_0 is referred to as the *head* and $\sigma_i, \ldots, \sigma_m$ as the *body* of the rule $\sigma_0 \leftarrow \sigma_1, \ldots, \sigma_m$;
- $\mathcal{A} \subseteq \mathcal{L}$ is a (non-empty) set, referred to as *assumptions*;
- $\bar{\ }$ is a total mapping from $\mathcal{A}$ into $\mathcal{L}$; $\bar{\alpha}$ is referred to as the *contrary* of α.

Referring back to the example used let us assume that the following mapping exists:

- *p - happy*
- *r - not happy*
- *a - eating*
- *q - good food*
- *b - no fork*
- *c - dirty hands*
- *s - fork*
- *t - clean hands*

Based on this mapping and the definition of the an ABA framework we can now construct a framework that represents the example, as shown in example 2:

Example 2. $\langle \mathcal{L}, \mathcal{R}, \mathcal{A}, \bar{\ } \rangle$ may consist of
$\mathcal{L} = \{a, b, c, p, q, r, s, t\}$
$\mathcal{R} = \{p \leftarrow q, a, q \leftarrow, r \leftarrow b, c\}$
$\mathcal{A} = \{a, b, c\}$
$\bar{a} = r, \bar{b} = s, \bar{c} = t$

The framework constructed in example 2 displays the various elements that allow one to represent a real-life decision problem in ABA. It can be seen that the rules related to our problem are defined in terms of inference rules. Additionally, the framework also includes the following elements:

Assumptions are, in our example, actions that can be taken or disputable observations about the argument. In general assumptions can be considered as vulnerable points of an argument that can be used to support or dispute that argument. The aim of ABA is to establish a set of "strong" assumptions that support or dispute the underlying argument.

Non-Assumptions represent goals we wish to achieve or avoid in accordance to the claim we are trying to analyse. These can be based on certain assumption we are considering or might be stand-alone goals.

Contraries are sentences that are opposing to the assumptions of our framework. ABA imposes that each assumption must be mapped to a contrary. For example, the assumption "no fork" in our framework has a contrary of "fork". Thus our framework is now equipped with both the assumptions necessary but also the opposing assumptions that can dispute them.

The description provides a simple overview of what these elements of the framework are and are used in the example. This will allow us to explore how claims are analysed. For the purpose of this report there is no need to explore to a deeper level the framework, however there are numerous publications that explain in these elements in more detail (see [9]).

2.4 Proving a claim using ABA

The ability of ABA to evaluate the support of a claim, lies in its ability to formulate and analyse arguments. Arguments are the deduction of a certain claim from the rules and assumptions provided. The following is a definition of an argument (according to [9]):

Definition 2. an argument is defined as follows:
An argument for (the claim) $\sigma \in \mathcal{L}$ supported by $A \subseteq \mathcal{A}$ ($A \vdash \sigma$ in short) is a deduction for σ supported by A (and some R $\subseteq \mathcal{R}$)

However, within the ABA framework there is also the notion of attack (similarly to AA). An attack between arguments under ABA occurs when one argument deduces the contrary of an assumption of another argument. Attacks can be defined as follows (from [9]):

Definition 3. an attack is defined as follows:
An argument $A_1 \vdash \sigma_1$ attacks an argument $A_2 \vdash \sigma_2$ iff σ_1 is the contrary of one of the assumptions in A_2.

ABA is equipped with various semantics that allow it to determine "winning/acceptable" sets of arguments and assumptions. This enables us to validate whether a claim can be deduced from these "acceptable" arguments and whether the claim is supported by these "winning" assumptions. These semantics can define the "acceptability" of both a set of arguments and a set of assumptions. Both approaches (or views) are defined below (in accordance to [9]):

Argument View Semantics

- *admissible* iff it does not attack itself and it attacks all arguments that attack it;
- *preferred* iff it is maximally (w.r.t. $\subseteq$) admissible;
- *sceptically preferred* iff it is the intersection of all preferred sets of arguments;
- *complete* iff it is admissible and contains all arguments it *defends*, where A defends α iff A attacks all arguments that attack α;
- *grounded* iff it is minimally (w.r.t $\subseteq$) complete;
- *ideal* iff it is maximally (w.r.t $\subseteq$) admissible *and* contained in all preferred sets of arguments;
- *stable* iff it does not attack itself and it attacks all arguments it does not contain.

Assumption View Semantics

- *admissible* iff it does not attack itself and it attacks all assumptions that attack it;
- *preferred* iff it is maximally (w.r.t. $\subseteq$) admissible;
- *sceptically preferred* iff it is the intersection of all preferred sets of assumptions;
- *complete* iff it is admissible and contains all assumptions it *defends*, where A defends α iff A attacks all assumptions that attack α;
- *grounded* iff it is minimally (w.r.t $\subseteq$) complete;
- *ideal* iff it is maximally (w.r.t $\subseteq$) admissible *and* contained in all preferred sets of assumptions;
- *stable* iff it does not attack itself and it attacks all assumptions it does not contain.

2.5 Computational mechanisms for ABA

The power of ABA is unleashed through the various computational mechanisms and tools that have been developed. These allow for ABA frameworks to be analysed in an algorithmic manner, thus allowing the creation of computerised systems that can carry out this analysis. The underlying features of these mechanisms are as follows (as described by [9]):

- they can be abstracted away as disputes between a *proponent* and an *opponent*, in a sort of (fictional) zero-sum, two-player game: the proponent aims at proving (constructively) that an initially given *sentence* is "acceptable"/"winning", the opponent is trying to prevent the proponent from doing so;
- these disputes are defined as a *sequence of tuples* (referred to as *dispute derivations*), representing the state of the game while it is being played;
- these disputes interleave the construction of arguments and identification of attacks between them with testing whether the input sentence is "acceptable"/"winning";
- the rules of the game allow proponent and opponent to perform various kinds of *filtering* during disputes, but different kinds for different argumentation semantics (for determining whether the input sentence is "acceptable"/"winning";
- the possible outcomes of dispute derivations are as follows:
  - the input sentence is proven to be "acceptable"/"winning" (the dispute derivation is *successful*) or is not proven to be so; and
  - if the dispute derivation is successful, it returns:
    1. the set of all assumptions supporting the arguments by the proponent(referred to as the *defence set*),
    2. the set of all assumptions in the support of arguments by the opponent and chosen by the proponent to be counter-attacked (referred to as the *culprits*),
    3. in the case of the proposal of ([8]), the *dialectical tree* of arguments by proponent and opponent.

With these features in mind several procedure and algorithms have been devised that determine whether a claim is "acceptable/winning" based on several of the semantics mentioned. These include a wide range of tools from flowcharts to logic programming systems, some of which are discussed in the following sections.

2.6 Proxdd

Proxdd (refer to [2]) is a system developed by Dr. Robert Craven that implements dispute derivations to analyse ABA frameworks. Given a constructed framework it can algorithmically analyse and return a set of winning arguments in accordance to the semantics specified. Proxdd currently supports admissible and grounded semantics. It is implemented using the Prolog logic programming language (for more details see [2]). The underlying computational mechanism used for Proxdd is the SXDD algorithm (defined in [8]).

Example 3. Example of input (from [2])

```
myAsm(a).
myAsm(b).
myAsm(c).
myAsm(d).
myAsm(e).
myAsm(f).
contrary(a, q).
contrary(b, f).
contrary(c, u).
contrary(d, v).
contrary(e, v).
contrary(f, v).
myRule(p, [a,u]).
myRule(q, [b,r]).
myRule(q, [c,s]).
myRule(q, [c,t]).
myRule(u, [a]).
myRule(s, []).
myRule(t, [d]).
myRule(t, [e]).
```

Example 4. Example of output (from [2])

```
DEF: [a]
CUL: [c]
ARG: [1:([a], [] -> p),6:([c], [] -> q),7:([a], [] -> u),
 8:([c], [d] -> q),9:([c], [e] -> q)]
ATT: [1-0,6-1,7-6,8-1,9-1]
```

2.7 Grapharg

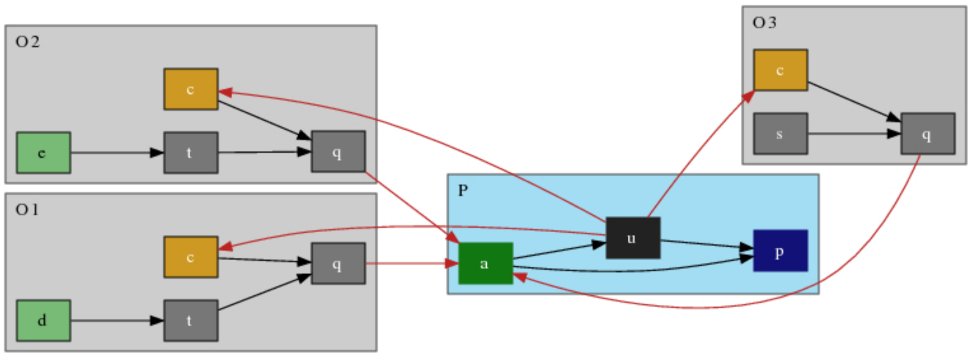
Grapharg (refer to [1]) is a system also developed by Dr. Robert Craven. Similarly to Proxdd it also provides dispute derivation for ABA. Furthermore, it is also implemented in Prolog, however the underlying algorithm is different. Unlike Proxdd it uses a graph-based dispute derivation algorithm that, according to the research's findings, optimises the computation of the dispute derivation (refer to [3]).

In both Proxdd and Grapharg the system takes in a framework in the form shown in example 3 and after the dispute derivation it produces an output similar to example 5. Additionally, by using graphviz, the output can be displayed graphically as in example 6.

Example 5. Example of output (from [1])

```
DEFENCE: [a]
CULPRITS: [c]
PROP JUSTIFICATIONS: [(a,*),(p,1),(u,5)]
OPP JUSTIFICATIONS: [[]-[(c,*),(d,*),(q,4),(t,7)]-[q],
[]-[(c,*),(e,*),(q,4),(t,8)]-[q],[]-[(c,*),
(q,3),(s,6)]-[q]]
ATTACKS: [(q,a),(u,c)]
GRAPH: []
```

Example 6. Example of current visualisation using graphviz (from [1])



However these systems are not user-friendly enough to promote ABA as an easily accessible tool. Both systems require setting up and both are operated using Prolog which many users might not be comfortable with. This means that the systems are not readily accessible to users and the interface provided could be more useful. This project aims at creating a web-application that will enable users to easily and instantly interact with the systems in an online and more user-friendly environment.

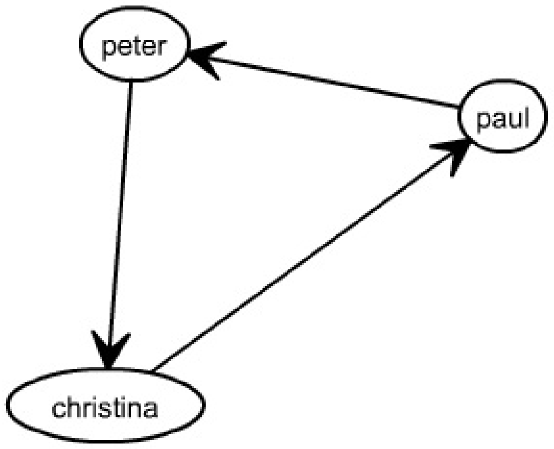
2.8 ASPARTIX

Perhaps the system that best resembles the desirable outcome of the project in terms of user experience and design is the ASPARTIX system (see [6]). ASPARTIX offers a web-application implementation that enables a user to enter a series of arguments and attack relationships between the arguments (see example 7), which is then processed and displayed diagrammatically to the user (see example 8). Our implementation should aim at providing a similar experience; allowing the user to implement their framework and displaying the results diagrammatically. However, ASPARTIX does not implement ABA instead it focuses more on Abstract Argumentation. Nonetheless, elements of the system such as its implementation as a web-application, the user's ability to define a framework and the existence of a canvas on which the visualisation is displayed will have to be integrated in this project's web-application as well.

Example 7. Example of input in ASPARTIX (from [10])

```
arg(peter).
arg(paul).
arg(christina).
att(christina, paul).
att(peter, christina).
att(paul, peter).
```

Example 8. Example of visualisation of ASPARTIX (from [10])



2.9 Implementing Ideal semantics dispute derivations

Both Proxdd and Grapharg currently do not support analysing ABA according to ideal semantics. Nonetheless, the computational mechanisms do exist (see [5]). By extending the engines with this additional computational mechanism, we will allow our web-application to provide analysis under the ideal semantics with minimal additional development effort.

Essentially this form of dispute derivation is an adapted version of that of the admissible semantics. Formally it is defined as follows (according to [5]):

Definition 4. Let $\langle \mathcal{L}, \mathcal{R}, \mathcal{A}, \bar{\ } \rangle$ be an assumption based framework. Given a selection function, an *IB-dispute derivation of an ideal support* A for a sentence α is a finite sequence of tuples

$\langle \mathcal{P}_0, \mathcal{O}_0, A_0, \mathcal{C}_0, \mathcal{F}_0 \rangle, \ldots, \langle \mathcal{P}_i, \mathcal{O}_i, A_i, \mathcal{C}_i, \mathcal{F}_i \rangle, \ldots, \langle \mathcal{P}_n, \mathcal{O}_n, A_n, \mathcal{C}_n, \mathcal{F}_n \rangle$
where

$\mathcal{P}_0 = \{\alpha\}$ $\quad$ $\mathcal{A}_0 = \mathcal{A} \cap \mathcal{P}_0$ $\quad$ $\mathcal{O}_0 = \mathcal{C}_0 = \mathcal{F}_0 = \{\}$
$\mathcal{P}_n = \mathcal{O}_n = \mathcal{F}_n = \{\}$ $\quad$ $A = A_n$

and for every $0 \leq i < n$, only one σ in $\mathcal{P}_i$ or one S in $\mathcal{O}_i$ or one S in $\mathcal{F}_i$ is selected, and:

1. If $\sigma \in \mathcal{P}_i$ is selected then

   (a) if σ is an assumption then

   $\mathcal{P}_{i+1} = \mathcal{P}_i - \{\sigma\}$ $\quad$ $A_{i+1} = A_i$ $\quad$ $\mathcal{C}_{i+1} = \mathcal{C}_i$
   $\mathcal{O}_{i+1} = \mathcal{O}_i \cup \{\{\bar{\sigma}\}\}$ $\quad$ $\mathcal{F}_{i+1} = \mathcal{F}_i$

   (b) if σ is not an assumption, then there exists some inference rule $\sigma \leftarrow R \in \mathcal{R}$ such that $\mathcal{C}_i \cap R = \{\}$ and

   $\mathcal{P}_{i+1} = \mathcal{P}_i - \{\sigma\} \cup (R - A_i)$ $\quad$ $A_{i+1} = A_i \cup (A \cap R)$ $\quad$ $\mathcal{C}_{i+1} = \mathcal{C}_i$
   $\mathcal{O}_{i+1} = \mathcal{O}_i$ $\quad$ $\mathcal{F}_{i+1} = \mathcal{F}_i$

2. If S is selected in $\mathcal{O}_i$ then σ is selected in S_u and

(a) if σ is an assumption, then

i. either σ is ignored, i.e.

$$\mathcal{O}_{i+1} = \mathcal{O}_i - \{S\} \cup \{m(\sigma, S)\} \quad \mathcal{P}_{i+1} = \mathcal{P}_i \quad A_{i+1} = A_i$$
$$\mathcal{C}_{i+1} = \mathcal{C}_i \quad \mathcal{F}_{i+1} = \mathcal{F}_i$$

ii. or $\sigma \notin A_i$ and $\sigma \in \mathcal{C}_i$ and

$$\mathcal{O}_{i+1} = \mathcal{O}_i - \{S\} \quad \mathcal{P}_{i+1} = \mathcal{P}_i \quad A_{i+1} = A_i$$
$$\mathcal{C}_{i+1} = \mathcal{C}_i \quad \mathcal{F}_{i+1} = \mathcal{F}_i \cup \{u(S)\}$$

iii. or $\sigma \notin A_i$ and $\sigma \notin \mathcal{C}_i$ and

A. if $\bar{\sigma}$ is not an assumption, then

$$\mathcal{O}_{i+1} = \mathcal{O}_i - \{S\} \quad \mathcal{P}_{i+1} = \mathcal{P}_i \cup \{\bar{\sigma}\} \quad A_{i+1} = A_i$$
$$\mathcal{C}_{i+1} = \mathcal{C}_i \cup \{\sigma\} \quad \mathcal{F}_{i+1} = \mathcal{F}_i \cup \{u(S)\}$$

B. if $\bar{\sigma}$ is an assumption, then

$$\mathcal{O}_{i+1} = \mathcal{O}_i - \{S\} \quad \mathcal{P}_{i+1} = \mathcal{P}_i \quad A_{i+1} = A_i \cup \{\bar{\sigma}\}$$
$$\mathcal{C}_{i+1} = \mathcal{C}_i \cup \{\sigma\} \quad \mathcal{F}_{i+1} = \mathcal{F}_i \cup \{u(S)\}$$

(b) if σ is not an assumption, then

$$\mathcal{P}_{i+1} = \mathcal{P}_i \quad A_{i+1} = A_i \quad \mathcal{C}_{i+1} = \mathcal{C}_i$$

$$\mathcal{F}_{i+1} = \mathcal{F}_i \cup \{S - \{\sigma\} \cup R | \sigma \leftarrow R \in \mathcal{R} \text{ and } R \cap \mathcal{C}_i \neq \{\}\}$$
$$\mathcal{O}_{i+1} = \mathcal{O}_i - \{S\} \cup \{S - \{\sigma\} \cup R | \sigma \leftarrow R \in \mathcal{R} \text{ and } R \cap \mathcal{C}_i = \{\}\}$$

3. If S is selected in $\mathcal{F}_i$ then *Fail(S)* and

$$\mathcal{O}_{i+1} = \mathcal{O}_i \quad \mathcal{P}_{i+1} = \mathcal{P}_i \quad A_{i+1} = A_i$$
$$\mathcal{C}_{i+1} = \mathcal{C}_i \quad \mathcal{F}_{i+1} = \mathcal{F}_i - \{S\}$$

Fail(S) is computed as follows:

Definition 5. Let $\langle \mathcal{L}, \mathcal{R}, \mathcal{A}, \overline{\ } \rangle$ be an assumption based framework. Given a selection function, a *Fail-dispute derivation* of a multiset of sentences S is a sequence $\mathcal{D}_0, \ldots, \mathcal{D}_n$ such that each $\mathcal{D}_i$ is a set of quadruples of the form $\langle \mathcal{P}, \mathcal{O}, A, \mathcal{C} \rangle$ where

$\mathcal{D}_0 = \{\langle S, \{\}, \mathcal{A} \cup S, \{\}\rangle\}$ $\mathcal{D}_n = \{\}$

and, for every $0 \leq i < n$, quadruple $Q = \langle \mathcal{P}, \mathcal{O}, A, \mathcal{C}\rangle$ is selected in $\mathcal{D}_i$ then either $\mathcal{P} \neq \{\}$ or $\mathcal{O} \neq \{\}$, and

1. If an element O from $\mathcal{O}$ is selected, then
   (a) if $O = \{\}$ then $\mathcal{D}_{i+1} = \mathcal{D}_i - \{Q\}$;
   (b) if $O \neq \{\}$ then let $\sigma \in O$ be the selected sentence in O:
      i. if σ is not an assumption then $\mathcal{D}_{i+1} = \mathcal{D}_i - \{Q\} \cup \{Q'\}$ where Q' is obtained from Q as in step (2.ii) of AB-dispute derivation;
      ii. if σ is an assumption then there are two cases:
      Case 1: $\sigma \notin A$. Then $\mathcal{D}_{i+1} = \mathcal{D}_i - \{Q\} \cup \{Q_0, Q_1\}$ where Q_0 is obtained from Q as in step (2.i.a) and Q_1 is obtained from Q as in steps (2.i.b) or (2.i.c) (as applicable) of AB-dispute derivation;
      Case 2: $\sigma \in A$. Then $\mathcal{D}_{i+1} = \mathcal{D}_i - \{Q\} \cup \{Q_0\}$ where Q_0 is obtained from Q as in step (2.i.a) of AB-dispute derivation;
2. If a $\sigma \in \mathcal{P}$ is selected, then
   (a) if σ is an assumption then $\mathcal{D}_{i+1} = \mathcal{D}_i - \{Q\} \cup \{Q'\}$ where Q' is obtained from Q as in step (1.i) of AB-dispute derivation;
   (b) if σ is not an assumption then $\mathcal{D}_{i+1} = \mathcal{D}_i - \{Q\} \cup \{Q'\}$ where there is a rule $\sigma \leftarrow R$ such that Q' is obtained from Q as in step (1.ii) of AB-dispute derivation.

The procedure defined above is explicitly explained in [5] and is used as the basis of the ideal semantics implementation. The web-application will now be able to analyse according to more semantics.

2.10 Decision Making with ABA

One of the most promising uses of argumentation is the potential of assisting decision making. This relies on being able to implement argumentation in a context that will allow it to compute the suitability of decisions. What makes argumentation exceptionally useful in the decision making context is that it not only proposes a decision, but it also provides the argumentation-based justification of it. This can be useful in real-life scenarios as it allows the user to understand the reasoning behind the decision taken. Research (see

[11]) has proposed the use of Decision frameworks and Decision Functions that are mapped to ABA in order to compute the required decision.

Definition 6. A decision framework *is a tuple* $\langle D, A, G, T_{DA}, T_{GA} \rangle$, *consisting of:*

- a set of decisions $D = \{d_1, \ldots, d_n\}, n > 0$,
- a set of attributes $A = \{\alpha_1, \ldots, \alpha_m\}, m > 0$,
- a set of goals $G = \{g_1, \ldots, g_l\}, l > 0$, and
- two tables: T_{DA}, of size $(n \times m)$, and T_{GA}, of size $(l \times m)$, such that
  - for every $T_{DA}[i, j]^2, 1 \leq n, 1 \leq j \leq m, T_{DA}[i, j]$ is either 1, representing that a decision d_i has attributes a_j, or 0, otherwise;
  - for every $T_{GA}[i, j], 1 \leq i \leq l, 1 \leq j \leq m, T_{GA}[i, j]$ is either 1, representing that goal g_i is satisfied by attribute a_j, or 0, otherwise.

In order to use ABA for decision the decision frameworks and decision functions need to be adapted to ABA. Computational Procedures have been identified for computing strongly dominant decisions, dominant decisions and weakly dominant decisions.

Definition 7. Given a decision framework $df = \langle D, A, G, T_{DA}, T_{GA} \rangle$, in which $|D| = n, |A| = m$ and $|G| = l$, the *strongly dominant ABA framework* corresponding to $\langle D, A, G, T_{DA}, T_{GA} \rangle$ is $df_s = \langle \mathcal{L}, \mathcal{R}, \mathcal{A}, \mathcal{C} \rangle$, where

- $\mathcal{R}$ is such that : for all $k = 1, \ldots, n; j = 1, \ldots, m$ and $i = 1, \ldots, l$:
  - if $T_{DA}[k, i] = 1$ then $d_k a_i \leftarrow$;
  - if $T_{GA}[j, i] = 1$ then $g_j a_i \leftarrow$;
  - $d_k g_j \leftarrow d_k a_i, g_j a_i$;
- $\mathcal{A}$ is such that: d_k, for $k = 1, \ldots, n; N d_k g_j$, for $k = 1, \ldots, n$ and $j = 1, \ldots, m$;
- $\mathcal{C}$ is such that: $\mathcal{C}(d_k) = \{N d_k g_1, \ldots, N d_k g_n\}$, for $k = 1, \ldots, n$;

$\mathcal{C}(N d_k g_j) = \{d_k g_j\}$, for $k = 1, \ldots, n$ and $j = 1, \ldots, m$.

Definition 8. Given $df = \langle D, A, G, T_{DA}, T_{GA} \rangle, |D| = n$, and $|A| = m$, let the corresponding strongly dominant ABA framework be $df_s = \langle \mathcal{L}, \mathcal{R}, \mathcal{A}, \mathcal{C} \rangle$, then the *dominant ABA framework corresponding to* df is $df_D = \langle \mathcal{L}, \mathcal{R}_\mathcal{D}, \mathcal{A}_\mathcal{D}, \mathcal{C}_\mathcal{D} \rangle$, where:

- $\mathcal{R}_\mathcal{D} = \mathcal{R} \cup N g_j^{\bar{k}} \leftarrow N d_1 g_j, \ldots, N d_{k-1} g_j, N d_{k+1} g_j, \ldots, N d_N g_j\}$ for $k = 1, \ldots, n$ and $j = 1, \ldots, m$;

- $\mathcal{A}_{\mathcal{D}} = \mathcal{A}$;
- $\mathcal{C}_{\mathcal{D}}$ is $\mathcal{C}$ with $\mathcal{C}(Nd_kg_j) = \{d_kg_j\}$ replace by $\mathcal{C}(Nd_kg_j) = \{d_kg_j, Ng_j^{\bar{k}}\}$, for $k = 1, \ldots, n$ and $j = 1, \ldots, m$.

Definition 9. Given $df = \langle D, A, G, T_{DA}, T_{GA}\rangle, |D| = n$ and $|A| = m$, the *weakly dominant ABA framework corresponding to* df is $df_W = \langle \mathcal{L}, \mathcal{R}, \mathcal{A}, \mathcal{C}\rangle$, where

- $\mathcal{R}$ is such that: for all $k = 1, \ldots, n; j = 1, \ldots, m$ and $i = 1, \ldots, l$:
  - if $T_{DA}[k, i] = 1$ then $d_ka_i \leftarrow$;
  - if $T_{GA}[j, i] = 1$ then $g_ja_i \leftarrow$;
  - $d_kg_j \leftarrow d_ka_i, g_ja_i$;
    for all $r, k = 1, \ldots, n, r \neq k$; and $j = 1, \ldots, m$:
  - $Sd_rd_k \leftarrow d_rg_j, Nd_kg_j, NSd_kd_r$;
  - $\bar{S}d_kd_r \leftarrow d_kg_j, Nd_rg_j$;
- $\mathcal{A}$ is such that: d_k, for $k = 1, \ldots, n$;
  NSd_kd_r, for $r, k = 1, \ldots, n, r \neq k$;
  Nd_kg_j, for $k = 1, \ldots, n$ and $j = 1, \ldots, m$;
- $\mathcal{C}$ is such that: $\mathcal{C}(d_k) = \{Sd_1d_k, \ldots, Sd_{k-1}d_k, Sd_{k+1}d_k, \ldots, Sd_nd_k\}$, for $k = 1, \ldots, n$;
  $\mathcal{C}(NSd_kd_r) = \{\bar{S}d_kd_r\}$, for $r, k = 1, \ldots, n, r \neq k$;
  $\mathcal{C}(Nd_kg_j) = \{d_kg_j\}$, for $k = 1, \ldots, n$ and $j = 1, \ldots, m$.

By implementing the above mapping and the computational mechanisms provided we can enable our web-application to come closer to being decision-making tool.

2.11 Mapping AA to ABA

AA can be mapped to ABA and thus AA frameworks can be computed using ABA. Specifically the mapping is as follows (according to [7]):

Definition 10. Each AA framework $\mathcal{F} = (Arg, attacks)$ can be mapped onto a *corresponding ABA framework* $ABA(F) = \langle \mathcal{L}, \mathcal{R}, \mathcal{A}, \overline{} \rangle$ with

- $\mathcal{A} = Arg$;
- for any $\alpha \in \mathcal{A}, \bar{\alpha} = c(\alpha)$, with
  - $c(a) \notin \mathcal{A}$ and,
  - for $\alpha, \beta \in \mathcal{A}$, if $\alpha \neq \beta$ then $c(\alpha) \neq c(\beta)$;
- $\mathcal{R} = \{c(\alpha) \leftarrow \beta | (\beta, \alpha) \in attacks\}$;

- $\mathcal{L} = \mathcal{A} \cup \{c(\alpha) | \alpha \in \mathcal{A}\}$.

By implementing the mapping within the system, we can now allow it to take as an input an AA framework and evaluate it using the underlying ABA engines. This will make the web-application more flexible as it will be able to compute in accordance to different forms of argumentation.

Chapter 3

System Overview.

3.1 Providing a web application based interface for derivation engines.

The project's main outcome is creating an extendible web application that provides an interface to various derivation engines for ABA. The application should be user friendly and allow for future integration with other engines. This extensibility can enable the application to establish itself as a common gateway to various derivation engines for ABA. As more engines are added the Web Application can be established as a one-stop-shop for all things ABA.

3.1.1 Making ABA more accessible.

By providing access to the derivation engines through a web application we can increase the availability of ABA derivation engines to the public. Anyone with access to the internet will be able to communicate with these engines and obtain derivation for frameworks they require. This accessibility will allow the increase in popularity of ABA frameworks and can provide the basis for further development to take place with ABA at its centre.

3.1.2 Common gateway to various engines.

Rather than just providing an online GUI interface for the derivation engines, the system is structured so as to provide an extendible middle—ware solution. This implementation allows the system to be extended both in the front—end and the back—end. Further applications can be built on top of the existing application allowing for either different styles of argumentation to be evaluated (as described in [TODO]) or applications that implement ABA in real—life scenarios.

Thus, it is important that the system is abstracted in such a way that allows such extendibility. It is also important that key elements of the systems flow can easily be adapted to any further implementation. This can be achieved by modularising the functionalities. The modules themselves are explored further in the following section [TODO]. However, in this section we discuss the flow of the system and how we accomplished the desired functionality. The specifics of the system in terms of implementation and design are discussed more elaborately in section [TODO].

3.2 Flow of the System.

Just like any other system this web application can be considered as a series of interlocking components that are combined to perform specific tasks. This implies there is a flow of information between these components when carrying out a task. In the case of our system this flow is mostly linear and can be considered as a pipeline to which we feed information and get a desired output.

When considering the system at the highest level of abstraction it can be defined as in figure [TODO]. The system receives an input by the user through its GUI and then handles all preprocessing steps in order to make the input compatible with the derivation engine. Once the derivation is complete the output is a again processed into a desirable output form which is provided by the GUI to the user. An overview of the individual steps and their purpose is provided in section [TODO].

3.2.1 Flow overview and Diagram.

The main components of the system are analysed in detail in terms of their implementation, design and evaluation in the respective sections. However to provide a more detailed overview of how the web application handles information flow we use Figure [TODO]. There we can see how the system handles the input and the configuration parameters. We can also see how it would be able to handle additional layers being built on top of it.

[TODO] - INSERT DIAGRAM OF FLOW.

3.2.2 Overview of functionalities.

Having considered the flow of data withing the application, we can now look at the individual components of the system and their purpose and functionality. For details of the implementation references to the relevant sections that follow are provided.

User Input.

The user is expected to be able to provide three key pieces of information.

- *Framework* - Users should be able to define an ABA framework using a suitable input mechanism.
- *Configuration* - Users should be able to configure the application by choosing which derivation engine and which semantics to use.
- *Target* - Users should be able to define the target for which they are seeking a derivation.

Therefore, a suitable method is required so the user can accomplish the above. As specified before the web application aims at providing a portal for users to numerous derivation engines. These engines can be implemented using different technologies, programming languages and might expect different input formats. By abstracting the input mechanism from the derivation engine, we can now allows user to use the same input mechanism to specify the input without having to concern themselves with the specifics of the derivation engine's implementation. To achieve this a simple input language has been defined and implemented, allowing the users to define their frameworks in the same way, irrespective of the underlying derivation engine chosen. The formal definition of this language and further elaboration on the implementation is discussed in section [TODO].

Parser.

Having established that the input mechanism will be command based a parser is required in order to recognise a command, check if it is valid and carry out the necessary instruction. The purpose of the parser is to check the validity of the input provided and provide the user with the required feedback if the input is invalid. If the input is valid then the parser forwards the input to the grounder so the respective grounder framework is derived. The implementation of the parser is discussed in section [TODO].

Grounder.

ABA frameworks can be defined based on an underlying domain. If this is the case then rules and assumptions might exist over certain parts of the domain, but not over others. The input mechanism accommodates this by allowing the users to define a domain over which an assumption exists. The means by which this is done are explained in section [TODO].

However, in our case both Proxdd and Grapharg do not accommodate ungroudned frameworks. Therefore, a mechanism must be provided that allows the generation of the respective grounded framework before it is passed to the derivation engine. Once again by abstracting the grounding process we can ensure that it is irrelevant whether the underlying derivation engine supports ungrounded frameworks as we can guarantee that any framework provided would first have been grounded over the specified domain.

By implementing a grounder we provide greater flexibility to the users and allow this to be done without having to change the implementation of the derivation engines. The details concerning the design and implementation of the grounder as established in section [TODO].

Input Generation.

This is the last step of the preprocessing carried out on the user input before it is provided to the derivation engine. By this stage we have not acquired a valid and grounded framework and have parsed the configuration parameters provided by the user. This implies that we are now aware of both the engine we require for the derivation and the semantics. Therefore, we are now able to translate the grounded framework we have in the required format expected by the targeted derivation engine.

This is the first bridge between our system and the underlying derivation engines. Generators have to be implemented based on the requirements of the expected input for each derivation engine. Based on the configuration settings provided by the user the correct generator is used to provide the input for the derivation engine in the expected format. Such generators should be easy to implement since the process they carry out is simply generating a string in the correct format. Implementation is discussed further in section [TODO].

Derivation.

The derivation process is carried out by the derivation engines and their implementation. This aspect of the web application can be considered as a black box. Ideally, the inner working of each derivation engine should not affect us much. Our web application is concerned with setting up the derivation engine as required by specifying the semantics to be used, providing the framework as an input and specifying a target. Once this is done the application then runs the derivation engine and awaits for the result.

Therefore once again we abstract the inner workings of the derivation engines from the rest of the applications. This allows us easily switch between the derivation engines depending on what the user has specified. Chosen implementation by which the web application interacts with the derivation engines is discussed in section [TODO].

It should be noted, however, that the Proxdd derivation engine has also been extended in order to provide the user with the ability to find derivations based on Ideal Semantics. The implementation of these semantics are discussed in section [TODO].

Output Generation.

Similarly to the Input Generation this step acts as a bridge between the derivation engine used and the web application. Depending on the derivation engine used the output provided might be in a different format. An output generator must be implemented that will allow the interpretation of this output and its conversion to the format required by the web application. Implementing such generators would again be straight forward. In interpreting the result in a single output format for the web application we abstract any further processing carried out from the specifications of the derivation engine. Once the result has been converted to the desired format the application can then process it as it requires, without having to concern itself about which derivation engine was used. Further discussion of the implementation of such generators is provided in section [TODO].

Visualisation.

Lastly, having analysed the framework and derived a solution for our target the web application then has to meaningfully transfer the result to the user. At this step the web application has interpreted the result and it is now able to provide the result to the user through it GUI. Visualisation of the result can be extended to include any format that the users might find useful. In our case the visualisation chosen is that of a derivation tree. The implementation of the visualisation of the result is discussed in section [TODO].

3.3 System Design choices.

There are certain peculiarities of the requirements of this system that require certain design choices to be made. These can be seen as part of the flow of data in the system and have to do with our desire to establish a flexible and extendible solution. The web application put substantial effort in abstracting the format of the user input and the output from what is expected by the derivation engine. Had we been building a simple web application laying on top of a single engine we could have built the application specifically in accordance to the implementation of the derivation engine, thus avoiding many of the preprocessing and post—processing steps, especially those involved with interpreting and converting the inputs and outputs to a format recognisable by the engine. However, this would make the system rigid and hard to accommodate any other engines or applications on top of it.

3.3.1 Plug—In and Plug—Out.

In order to be able to allow for the functionality not to be reliant on the underlying derivation engine the web application has to accommodate a Plug—In and Plug—Out mentality for some of the desired components. This implies that depending on the configuration chosen by the user the application must adapt and plug into the required components effortlessly. Additionally, this implies that new components should be added to this application with similar ease. Once the necessary adjustments are done the system will be able to plug the new components in and out.

3.3.2 Need for Generator functions.

Integral to this idea of plug—in and plug—out approach is the ability to isolate the derivation engines and consider them as black boxes. In order to do so we require two generator functions for each engine that act as wrappers to the engines. They allow the conversion of our input and output to and from the formats the derivation engine expects. Depending on which derivation engine has been chosen during configuration the correct generator functions are chosen. These functions, one could argue, could hinder the performance of the application as it involves what could potentially be unnecessary format conversion. However, this is an inevitably trade—off in order to achieve the flexibility we require to provide a true middle—ware solution that can be extended on both sides easily.

Chapter 4

Creating an Input Language.

A user of the application must be able to define an ABA framework from which he wants to derive a solution for a claim. Therefore a mechanism must exist that allows the user to do so. However, the web application has been designed with extendibility in mind and the input provided should be in a form that makes it useful irrespective of the underlying derivation engine.

4.1 Creating a Context Free Grammar for the Input.

When deciding on the input method there were several parameters that had to be considered. Some of the most important ones were:

- It must be easy for the user to understand and use.
- Should be able to accommodate all the derivation engines implemented now and in the future.
- Should allow for the easy input of both large and small frameworks.

Having considered these factors, the input method chosen for the web application is a simple text input comprised of predetermined commands that can be interpreted by a parser to input the framework to the desired derivation engine.

4.2 The need for a new input.

Perhaps the most important reason for defining formally a new input language to create frameworks is that by abstracting the input language from the derivation engines we can then use a single language for all the potential derivation engines. Alternatively, the user would have to be aware of the input mechanism for each derivation engine. Instead we chose to abstract the user input from the engines themselves and then create the corresponding required input for the specified engine. Essentially the user remains blissfully ignorant of the specifics of each derivation engine and of the further tiers of the systems. This reduces their ability to interfere with the engines directly and avoids confusion between different input languages.

Furthermore the input language suggested and implemented was designed to be as simple and straight forward as possible. This reflects the fact defining an ABA framework simply requires the definition of assumptions, rules and contraries. Therefore, the language is made up of two statements:

- asm(a,b). used to define an assumption and its contrary.
- b(X) <- [a(X)]. used to define a rule.

Additionally, the language we specified allows for the definition of a domain over each of the elements of the framework is valid. By combining this with the grounder described in section [TODO], we can define domain specific framework. This feature of our language also allows us to reduce the amount of lines of input the user has to enter in order to define a specific framework.

4.2.1 Formal definition of Input.

In order to be able to check the input for validity a formal definition must be followed. The Context Free Grammar for our input language is defined below in section [TODO].

```
Context Free Grammar for Argumentation Web-Application.
==========================================================


S
=====
StatList

P
=====
StatList -> <EoF>
StatList -> Stat Domain '.' StatList
Stat -> 'asm(' Atom ',' Atom ')'
Stat -> Atom '<-' '[' Terms ']'
Terms -> ''
Terms -> Atom Atoms
Atoms -> ''
Atoms -> ',' Atom Atoms
Strings -> ''
Strings -> ',' String Strings
Atom -> AtomName '(' String Strings ')'
Domain -> ''
Domain -> '{' Elem Elements '}'
Elements -> ''
Elements -> Elem Elements
Elem -> VarName '=' Value Values ';'
Values -> ''
Values -> ',' Value Values
Value -> [a-z0-9]+
VarName -> [A-Z]?[A-Za-z0-9]*
AtomName -> [a-z0-9]+
String -> [A-Za-z0-9]+

t
=====
{'.','asm(',',',')','<-','','{','}','<EoF>','=',';','[A-Za-z0-9]+'}

nt
=====
{StatList, Stat, Terms, Strings, String, Domain, Elements, Elem,
Atom, AtomName,Atoms,Values,Value,VarName,AtomName}
```

4.3 Parsing in the input.

Having formally defined the language we now proceeded with creating an interpreter to parse the user input and take the necessary action. This involves a simple 3 step process (as in figured [TODO]) by which the input is parsed, checked if it is valid and then the corresponding input for the targeted derivation engine is generated.

4.3.1 Building a simple parser for the language.

As noted the input language specified has just two distinct commands that the user can use to specify elements of a framework. This is due to the simplicity of specifying ABA frameworks. The two commands available are:

- *asm(a,b).* - used to define an assumption and its contrary.
- *b(X) <- [a(X)].* - used to define a rule.

The input can be considered as a series of statements separated by the terminal character ".". Each statement specifies either an assumption with its contrary or a rule. Using these two commands interchangeably the user can specify their argumentation framework. By keeping the language simple a simple parser can be created that checks whether each input statement is in one of the two forms.

Once a statement is extracted it is mapped to one of the two cases using the unique identifier tokens "asm(" or "<-", which are used to identify whether the user is specifying a rule or an assumption. The tokens were chosen to be easy to remember and reuse. The use of the "<-" operator in the rule definition is exceptionally memorable for users as it is the most common way in which rules are specified in the literature and it is a also a very commonly used operator in logic overall.

Due to the simple and strict nature of the language the parser created is simple in its implementation. It carries out the the following three tasks:

1. Separates the statements by the terminal character ".".

2. Each statement is then identified as either an assumption declaration, rule declaration or an invalid statement.

3. If the statement is valid, it is then separated in its individual parts according to the terminal symbols and the corresponding object (a rule or an assumption/contrary) is created.

In addition to these two commands the language also offers the possibility for the user to define a domain over which certain assumptions and rules are valid. this is done using the tokens defined in [TODO]. The existing engines of Proxdd and Grapharg do not take into account ungrounded assumptions and rules that contained unassigned variables. However, when defining an ABA framework it might be the case that the same assumption holds over various members of a domain.

The language supports the definition of a domain over which an assumption is valid. The functionality of grounding these assumptions over the domain is handled as part of a pre-processing step before the input for the derivation engines is generated. This is explained thoroughly in section [TODO].

If the parser manages to parse the whole of the input successfully, then by the end a web of assumption contrary and rule objects is created, that corresponds to the ABA framework defined.

4.3.2 Verifying the correctness of the input.

One of the key purposes of the parser is not only to interpret the input, but also to verify whether it is valid. Due to the limited amount of statements that are considered valid and the simple linear structure of an input (list of statements) we can simply compare each statement against a mask created using a regular expression. Specifically the regular expressions are defined in [TODO]. For simplicity and clarity we have isolated the regular expressions that checks terms and domains as these are repeated in the individual regex of the commands.

```
Term
====
[a-z]?[A-Za-z0-9]+\([A-Z][A-Za-z0-9]*(,[A-Z][A-Za-z0-9]*)*\)

Domain
======
(\{(([A-Z]+=[A-Za-z0-9])+(,[A-Za-z0-9])*;)+\})?

Assumption
==========
asm\(Term,Term\) Domain

Fact
====
Term<-\[\] Domain

Rule
====
Term<-\[Term(,Term)*\]
```

If a statement does not fit in these regular expressions then it is an invalid statement and the input in its entirety is considered as invalid. This strict approach to the input ensures that the user has not made any human error that would render it invalid.

4.3.3 Forcing the existence of a contrary.

ABA frameworks themselves have certain rules by which they must adhere to. A common example, that is prone to human error when defining an ABA framework, is that a contrary must be defined for every assumption declared. Our suggested input language is constructed so as to force the user to declare the contrary upon defining a new assumption. This is done by incorporating the definition of a contrary in the declaration of an assumption, as shown in example [TODO].

```
asm(a(X),b(X)){X=1,2,3;}.
asm(b(X),c(X)){X=1,2,3;}.
c(X)<-[d(X)].
```

In example [TODO] we see an example of a simple framework. We can also see that when defining an assumption using the *asm(Term,Term)* command we are also forced to define a contrary to the assumption (the second term). This contrary can either be an assumption itself (as is the case with b(X)) or it can be the head of a rule (as is the case with c(X)).

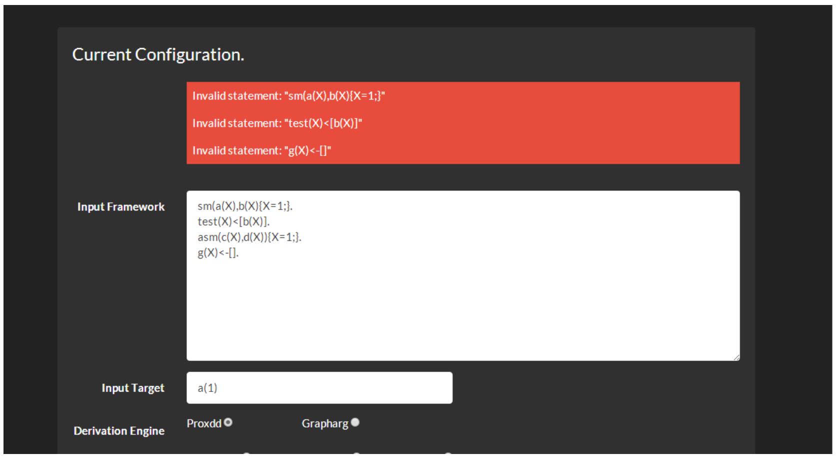
By forcing the user to define the contrary in the same step as declaring the assumption we ensure that there will always exist a contrary for an assumption. It should be noted that the current implementation restricts the user to defining only one contrary for each assumption. If the need arises in the future then the second term in the assumption command could be redesigned as a list that could specify more than one contraries for a single assumption.

4.4 Error checking and feedback for user.

The formally defined language we have established allows for the easy and clear declaration of a framework from a user, using the simple commands provided. We have already seen how the parser verifies that valid commands have been provided, however mechanisms must exist that help the user rectify mistakes they have done and aid them in establishing the validity of their input. The web application does so by providing feedback to the users about errors through useful error messages.

4.4.1 Detecting an Invalid Statement.

Firstly, we provide feedback when invalid statements are detected by the regular expressions explained in section [TODO]. When the user submits a framework input that includes commands that have not been matched to any of the regular expressions, then the web application recognises this and instead of moving on to grounding an invalid framework, it returns to the user feedback that allows them to correct their mistakes. This feedback is in the form of a list of errors notifying the user of the existence of the invalid commands, as shown in example [TODO].



Current Configuration.

Invalid statement: "sm(a(X),b(X){X=1;}"

Invalid statement: "test(X)<[b(X)]"

Invalid statement: "g(X)<-[]"

Input Framework

```
sm(a(X),b(X){X=1;}.
test(X)<[b(X)].
asm(c(X),d(X)){X=1;}.
g(X)<-[].
```

Input Target: a(1)

Derivation Engine: Proxdd ● Grapharg ●

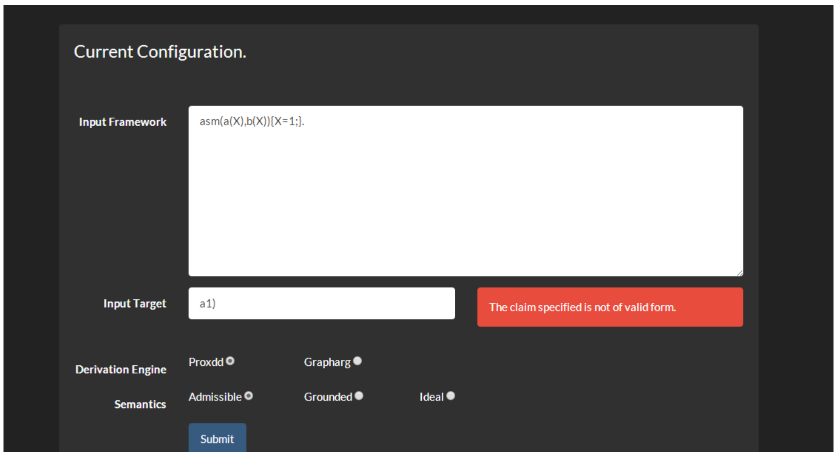
Figure 4.1: Invalid Statement notifications for the user.

4.4.2 Validate form of user defined target.

Additionally, before processing the input the application also verifies that the user defined target is of valid form. This is done by verifying that the target matches a provided regular expressions. The regular expression is defined in [TODO].

```
Target
======
[a-z]?[A-Za-z0-9]+\([a-z0-9][A-Za-z0-9]*
                              (,[a-z0-9][A-Za-z0-9]*)*\)
```

If the target specified does not match, then it is not of a valid form and the user should correct it. The user is notified of such error by the display of an alert message as shown in [TODO].


Current Configuration.

Input Framework: asm(a(X),b(X)){X=1;}.

Input Target: a1)

The claim specified is not of valid form.

Derivation Engine: [✓ Proxdd] Grapharg

Semantics: [✓ Admissible] Grounded Ideal

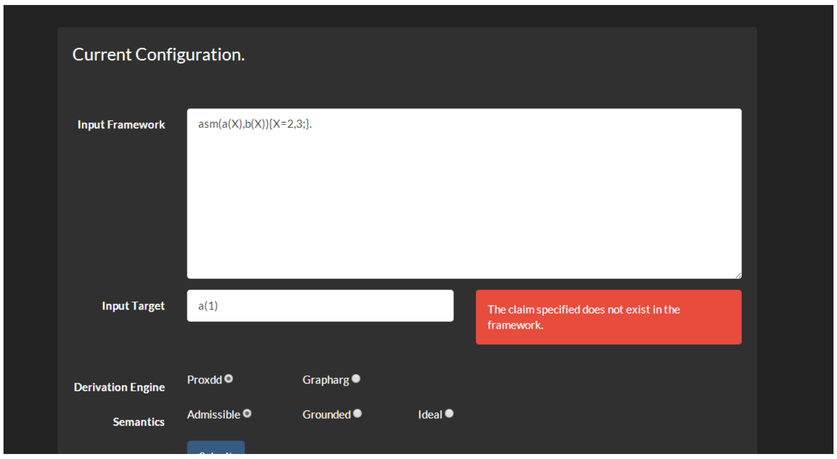
Figure 4.2: Input Target is not in a valid format.

4.4.3 Validate the existence of a target in the grounded framework.

Taking it a step further our web application's interpreter not only checks if the input of both the target and the framework are syntactically correct, but also provides feedback on whether the target defined is reasonable. Specifically, once the grounded framework has been established the web application then checks whether the target defined by the user exists in the grounded framework.

The implementation searches among the grounded framework for an assumption statement or a rule statement that defines the user provided target. If no such rule exists then the target cannot possibly be reachable withing the current framework. Therefore, a derivation would not be possible.

The user is notified of this case through an alert notification generated and displayed as shown in example [TODO].


Current Configuration.

Input Framework
```
asm(a(X),b(X)){X=2,3;}.
```

Input Target: a(1)

The claim specified does not exist in the framework.

Derivation Engine: [✓ Proxdd] Grapharg

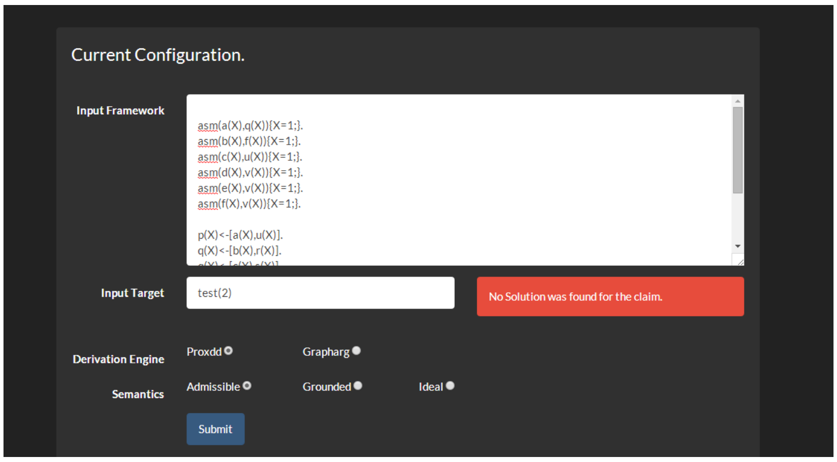
Semantics: [✓ Admissible] Grounded Ideal

Figure 4.3: Target specified does not exist in framework.

4.4.4 Check if derivation for target was found.

Lastly, the system also runs a post-derivation check. This allows us to verify that a derivation has been found for the target once the framework has been processed by the derivation engine. Despite ensuring the validity of the framework and the existence of the target in the grounded framework, it might still be the case that there is no valid derivation for the target under the semantics specified.

This is a perfectly reasonable outcome. Failure to find a solution under specific semantics does not imply that a mistake is made, but rather that no solution is possible to the problem specified. Therefore, it is useful for the user to be informed of this case, as is done in example [TODO].


Current Configuration.

Input Framework

```
asm(a(X),q(X)){X=1;}.
asm(b(X),f(X)){X=1;}.
asm(c(X),u(X)){X=1;}.
asm(d(X),v(X)){X=1;}.
asm(e(X),v(X)){X=1;}.
asm(f(X),v(X)){X=1;}.

p(X)<-[a(X),u(X)].
q(X)<-[b(X),r(X)].
```

Input Target: test(2)

No Solution was found for the claim.

Figure 4.4: Derivation for target was not found.

4.5 Example of an input.

To illustrate a valid input corresponding to a valid ABA framework the following example is provided [TODO].

Example 9. Consider a ABA framework as follows:

R - includes the following rules:

z←a
z←b
y←a
x←d
v←c

A $= \{$a,b,c,d$\}$ and $\bar{a} = z, \bar{b} = y, \bar{c} = x, \bar{d} = v$.

The user can input the framework above into the web application using the formally defined input language. This can be done be providing the representative list of commands that represent this framework. The input required is provided in section [TODO]. Note that the input provided is in propositional logic and represents an already grounded framework. This is done for readability and the handling of ungrounded frameworks and variables are elaborated on in secton [TODO].

[TODO] CHECK THAT THE GROUNDED UNGROUNDED THING IS VALID AT END!

```
asm(a,z).
asm(b,y).
asm(c,x).
asm(d,v).

z <- [a].
z <- [b].
y <- [a].
x <- [d].
v <- [c].
```

The input can be provided to the web application through the user interface and the "Input Framework" section. This is discussed further in section [TODO], however for illustrative purposes please refer to [TODO] on how this would be achieved.

[TODO] - Image of inputting framework in app.

The user has successfully inserted their framework in the desired format and has additionally declared a valid target. By clicking the "Submit" button the input configuration provided is submitted and, provided no error have been found, the web application starts the required procedures to find a valid derivation of the target.

In the case where users desire to specify a domain over which the framework is valid, they can you our language's support of such a case and adapt their commands accordingly. For example consider the framework defined in [TODO], but this time we also want to define the domain over which it is valid (X=1,2). The input to the web application would now be as in [TODO], were we can see the domain being defined in the format {X=1,2;} as specified by our formal language.

```
asm(a(X),z(X)){X=1,2;}.
asm(b(X),y(X)){X=1,2;}.
asm(c(X),x(X)){X=1,2;}.
asm(d(X),v(X)){X=1,2;}.

z(X) <- [a(X)].
z(X) <- [b(X)].
```

```
y(X) <- [a(X)].
x(X) <- [d(X)].
v(X) <- [c(X)].
```

Chapter 5

Grounding a framework over its domain.

As explained in previous sections the current implementation of the derivation engines of both Grapharg and Proxdd do not support the inputting of frameworks that include non-grounded elements. Consider example [TODO].

```
asm(likes(X,Y),dislikes(X,Y)).
```

This example illustrates the definition of an assumption “likes” which takes to parameters X and Y. This could be used to define a relationship by which we are assuming that “john likes cheese”, this is represented by the assumption “likes(john,cheese)”. In addition to this we also define the contrary to be “dislikes(X,Y)” which can be interpreted similarly to “likes”. However, in our definition X and Y are variables and therefore our definition is not grounded. The variables X and Y can take any instances in a domain to represent the existence of the required relationships. By incorporating this feature we can then use the general ungrounded definition to define specific relationships occurring in our domain, such as everyone who “likes cheese” or even everyone who “likes ham” or any other ingredient.

The use of assumptions and rules specific to a certain domain is very important when it comes to decision making as often enough assumptions and rules change depending on the parameters of a situation. Unbounded variables to assumptions and rules cannot be handled and therefore we are forced with creating frameworks that are either valid towards just one specific domain or provide a general overview of what the expected outcome would be.

5.1 The need of specifying a domain for a framework.

Often enough the assumptions and rules of an ABA framework are valid over a specific domain rather being valid globally. However, defining the domain over which elements of the domain are valid allow us to derive a specific solution for a more specific scenario. Often enough, the parameters of a situation might affect which assumptions and rules are valid. Consider the example [TODO]

[TODO] - Find Example, Write it and Walk through it.

Therefore, a mechanism must exist that allows the user to specify over which domain the assumptions or rules hold. One such possibility is for the users to directly specify the parameters of an assumption or rule in the declaration sentence when inputting the framework. This would be a cumbersome task especially when it comes to exceptionally large frameworks.

Consider, for example an assumption that is valid for 20 different people. When inputting the framework the user would have to declare a new assumption for every individual. This would imply that the user will have to declare the same assumption, but with different parameters, 20 times.

An alternative is to provide the user with the ability to define the framework over which an element is valid and then build a grounder that would bound all unbound variables according to the domain specified.

5.2 Implementing a grounder.

The input language of the application allows the user to define a domain over which assumptions and rules are valid as specified in section [TODO]. However, for the domain to come into effect the elements declared in the input should be grounded over the provided domain. The grounder was implemented in the Server module using C# and is part of the pre-processing of the input (as shown in figure [TODO]) before the corresponding input to the derivation engine is generated. This implies that the input to the derivation engine does not include any unbounded variables, which is a requirement by the current derivation engines Proxdd and Grapharg.

The grounder used in our implementation has been developed from the ground up and is based on the grounder algorithms used by DLV (Add reference [TODO]). The algorithms themselves, along with details of the implementation are provided in section [TODO].

With the existence of a grounder our web application can now handle more real world decision problems that have been formulated in an ABA framework. It can also evaluate the validity of a claim given certain parameters and can provide an analysis of domain specific problems.

5.2.1 Description of the algorithm.

The algorithm implemented is built based on the grounder algorithms used for DLV (reference [TODO]). The objective of the grounder is to take the user specified framework and ground the individual components over the domain they are valid. This is carried out by the algorithm described in this section. The algorithm can be summarised in the following simplified steps:

1. The elements of the framework are placed in the EDB and IDB. If an element is an assumption or a fact (a rule without a body) then they are grounded based on the domain they are specified and added to the EDB. All other rules that are depended on other predicates are added to the IDB.
2. A dependency graph of the program represented by the IDB is formulated and split into subprograms, thus creating a modular dependency graph made up of Strongly Connected Components.
3. An ordering of the modules of the dependency graph is derived and implemented.
4. The modules are processed in the order defined and the grounded rules within each module are instantiated and added to the grounded program.

As specified the first step of our grounding process is to identify facts that definitely hold in our framework, ground them over the domain specified and add them to the EDB. This includes two special cases:

- *asm(a(X),b(X)){X=1,2,3; }.* - once assumptions and contraries are defined then they must exist in the ABA framework as they are not reliant on the existence of other predicates.
- *b(X)<- {X=1,2,3; }.* - rules without a body are equivalent to *b(X)<-true* and therefore again are not reliant on the existence of other predicates.

Both of these cases can simply be grounded by calculating all possible combinations of the variables as defined in the domain by the user and then instantiating these predicates with these combinations and adding them to the EDB. The rest of the rules are added to the IDB for further processing and grounding.

Once we have isolated the initial IDB we can now pre-process it and start working towards grounding the various extra rules. This pre-processing allows us to avoid instantiating unnecessary rules that are not achievable since their predicates are not valid under the specified domain. The first step of the pre-processing is generating a dependency graph that replicates the interdependencies between the various rules. This is pretty straight forward to implement and was done by creating a data structure that includes the nodes and the edges between them as specified in [TODO].

Having established the dependency graph we now proceed by partitioning the graph into sub-programs. This is done by identifying strongly connected components (SCC) as defined in definition [TODO]. The dependency graphed is analysed using Tarjan's strongly connected component (see [TODO]) algorithm and the modules of our graph are identified and used to construct a new Component graph representing the program.

Definition 11. A strongly connected component is defined as a partition of a graph such that it is a maximal subset of vertices, such that every vertex is reachable by every other vertex.

With the modular dependency graph now constructed we must derive an ordering between the modules. The modules are placed into sorted list with the order being defined as in [TODO]. This enables us to first instantiate the rules on which rules from the following modules depend on. Having done so we can minimise the amount of unnecessary rules that would be created as explained in [TODO].

Definition 12. An admissible ordering between components is one such that: If A and B are components of our modularised graph G, then A precedes B if there exists a path from component A to component B.

```
algorithm tarjan is
  input: graph G = (V, E)
  output: set of strongly connected components
      (sets of vertices)

  index := 0
  S := empty
  for each v in V do
    if (v.index is undefined) then
      strongconnect(v)
    end if
  end for

  function strongconnect(v)
    // Set the depth index for v to the smallest unused index
    v.index := index
    v.lowlink := index
    index := index + 1
    S.push(v)

    // Consider successors of v
    for each (v, w) in E do
      if (w.index is undefined) then
        // Successor w has not yet been visited; recurse on it
        strongconnect(w)
        v.lowlink  := min(v.lowlink, w.lowlink)
      else if (w is in S) then
        // Successor w is in stack S and hence in the
        // current SCC
        v.lowlink  := min(v.lowlink, w.index)
      end if
    end for

    // If v is a root node, pop the stack and generate an SCC
    if (v.lowlink = v.index) then
      start a new strongly connected component
      repeat
        w := S.pop()
        add w to current strongly connected component
      until (w = v)
      output the current strongly connected component
    end if
  end function
```

Having completed the pre-processing stage we can now focus on instantiating the rules that represent our framework over the defined domain. This is carried out by the algorithms used for DLV (reference [TODO]) which are described in [TODO]. These instantiate the rules of one module at a time and work incrementally until all of the modules of the ordered list are processed.

```
Procedure Instantiate(P:Program; CG:ComponentGraph;
                var GP:GroundProgram)

        var S:SetOfAtoms, (C1,....,Cn): List of nodes of CG;
        S = EDB(P);
        GP := null;
        /* admissible component sequence */
        (C1,....Cn) := OrderedNodes(CG);
        for i = 1...n do InstantiateModule(P,Ci,S,GP);

Procedure InstantiateMethod(P: Program; C: SetOfPreficates;
                var S: SetOfAtoms; var GP: GroundProgram)

        var NS:SetOfAtoms, dS:SetOfAtoms;
        NS := null;
        dS := null;
        for each r in Exit(C,P) do
                InstantiateRule(r,S,dS,NS,GP);
        do
                dS := NS;
                NS := null;
                for each r in Recursive(C,P) do
                        InstantiateRule(r,S,dS,NS,GP);
                S := union(S,dS);
        while NS != null
```

```
Algorithm Instantiate
Input R:Rule,I:Set of instances of predicates;
Output S:Set of Total Substitutions;
var L:Literal, B:List of Atoms, theta:Substitution,
        MatchFound: Boolean;

Begin
        theta=null;
        /* Returns ordered list of body literals */
        B:=BodyToList(R);
        L:=L1;
        S:=null;
        while L != null
                Match(L,theta,MatchFound);
                if MatchFound
                        if(L not Last) then
                                L:= NextLiteral(L);
                        else
                  /* theta is substitution for variables */
                  /* of R */
                                S:= union(S,theta);
                                L := PreviousLiteral(L);
                         /* Look for other substitution */
                                MatchFound:= false;
                         /* Bounded variables up to */
                         /* previous literal */
                                theta:=PreviousVars(L);
                else
                        L:=PreviousLiteral(L);
                        theta:=PreviousVars(L);
        output S;
end;
```

For every rule we instantiate it according to the matching algorithm as defined in [TODO]. This algorithm goes through the predicates of a rule and finds valid grounded substitutes of the predicate already existing on the EDB. If a match is found then that substitution of the rule is added to the grounded program and the EDB.

```
Procedure Match(L:Literal, var theta:Substitution,
                var MatchFound:Boolean)
begin
        /* First try on new literal */
        if MatchFound then
                FirstMatch(L,theta,MatchFound);
        /* last match failed */
        /* try other match on previous literal */
        else
                NextMatch(L,theta,MatchFound);
end;

Procedure FirstMatch(L:Literal, var theta:Substitution,
                var MatchFound:Boolean)
        /* find first tuple of values matching theta*/
        /* Update theta and if Match found set */
        /* MatchFound to true else false */

Procedure NextMatch(L:Literal, var theta:Substitution,
                var MatchFound:Boolean)
        /* Similar to FirstMatch by finds next match. */
```

Once all the rules of all the modules have been processed we are now provided with the final grounded framework.

5.2.2 Advantages of implementation.

An easier to implement and more primitive grounder could simply compute all possible values for each variable in our framework and then build a dictionary of these values. This dictionary would then be used to create the grounded rules using all possible combinations of values for the unbounded variables. Nonetheless, this could create an explosion to the size of the framework that might be unnecessary. Consider the following example, although the ungrounded framework is just 2 commands we can see this is multiplied manifold once groudned.

```
Ungrounded framework
====================
asm(a(X),b(X)){X=1,2,3,4,5,6,7,8,9;}.
c(X)<-[a(X)].

Grounded framework
==================
asm(a(1),b(1)). asm(a(2),b(2)).
asm(a(3),b(3)). asm(a(4),b(4)).
asm(a(5),b(5)). asm(a(6),b(6)).
asm(a(7),b(7)). asm(a(8),b(8)).
asm(a(9),b(9)).

c(1)<-[a(1)]. c(2)<-[a(2)].
c(3)<-[a(3)]. c(4)<-[a(4)].
c(5)<-[a(5)]. c(6)<-[a(6)].
c(7)<-[a(7)]. c(8)<-[a(8)].
c(9)<-[a(9)].
```

As the number of commands increases and the size of the domain becomes larger the explosion of commands can only become worse.

By implementing a smarter grounded we can avoid the generation of rules that are not achievable. If a rule is made up of predicates that are not valid over the domain specified then the rule is not constructed in the final grounded framework. This allows us to restrict our framework only to rules that could be valid and useful. This also provides a performance boost to the derivation engines as the framework provided for analysis is smaller.

5.2.3 Disadvantages of implementation.

Grounders are an extensive research area especially when it comes to creating highly optimised algorithms. There are several techniques that can be used to improve the performance of a grounder and make it more efficient such as [TODO] reference to Gringo and [TODO] Reference to BJ instantiate for DLV. Our implementation does not focus on making the most of these optimisation techniques. This was deemed reasonable as since the input will be defined by the user will hardly ever be of a size that would provide a considerable performance gain to justify the extra development effort that would be required to implement these optimisation techniques.

Additionally, the current grounder is restricted in when it comes to the input it accepts. Although it serves the formal input language defined it would require additional development to be used in other cases or if the formal language is extended. The most notable limitation of our grounder is its inability to accommodate negative predicates, as in exampe [TODO]. However, in the ABA frameworks we are analysing negative predicates are not often encountered in the body of a rule.

```
asm(a(X),b(X)){X=1,2,3;}.
c(X,Y)<-[a(X),not a(Y)].
```

In the above example we can see the use of a negative predicate (using the not operator) which acts as the negations of the predicate. With the current implementation the grounder does not handle negative predicates in this manner. To do so the derivation of the ordering of the modules must change to accommodate such cases. However in ABA these are very isolated instances.

5.3 Example of expected output.

To demonstrate the use of our grounder we will use the example of a simple family tree. Consider the following family tree which goes back three generations [TODO].

[TODO] - Add simple family tree.

Our aim is to establish is to demonstrate the ability of the grounder to ground framework by finding valid relationships using global relationships such as "parent" and "grandparent" over a specific domain (this family). Essentially we need to establish the following two relationships that exist in our family tree.

[TODO] - add frame.

parent(X,Y) - this implies that X is a parent of Y. grandparent(X,Y) - this implies that X is a grandparent of Y.

The framework can be converted to a series of commands that allow for the definition of the family we are trying to represent in our framework. These commands are included in section.

```
asm(parent(X,Y),notparent(X,Y))
                {X=andy,beth;Y=fiona,eleanor;}
asm(parent(X,Y),notparent(X,Y))
                {X=cal,doris;Y=gary,helen;}
asm(parent(X,Y),notparent(X,Y))
                {X=fiona,gary;Y=ian,jake,ken;}

grandparent(X,Z) <- [parent(X,Y),parent(Y,Z)].
```

The objective of the grounder here is to use the knowledge we have defined concerning the "parent" relationships and ground the framework. In our example this will become visible as the grounder will derive the "grandparent" relationship that are viable based on the domain we have specified. Therefore, by grounding the framework over the domain using our grounder we can get the output shown in [TODO].

```
asm(parent(andy,fiona),notparent(andy,fiona)).
asm(parent(beth,fiona),notparent(beth,fiona)).
asm(parent(andy,eleanor),notparent(andy,eleanor)).
asm(parent(beth,eleanor),notparent(beth,eleanor)).
asm(parent(cal,gary),notparent(cal,gary)).
asm(parent(doris,gary),notparent(doris,gary)).
asm(parent(cal,helen),notparent(cal,helen)).
asm(parent(doris,helen),notparent(doris,helen)).
asm(parent(fiona,ian),notparent(fiona,ian)).
asm(parent(gary,ian),notparent(gary,ian)).
asm(parent(fiona,jake),notparent(fiona,jake)).
asm(parent(gary,jake),notparent(gary,jake)).
asm(parent(fiona,ken),notparent(fiona,ken)).
asm(parent(gary,ken),notparent(gary,ken)).

grandparent(doris,ken)<-[parent(doris,gary),parent(gary,ken)].
grandparent(doris,jake)<-[parent(doris,gary),parent(gary,jake)].
grandparent(doris,ian)<-[parent(doris,gary),parent(gary,ian)].
grandparent(cal,ken)<-[parent(cal,gary),parent(gary,ken)].
grandparent(cal,jake)<-[parent(cal,gary),parent(gary,jake)].
grandparent(cal,ian)<-[parent(cal,gary),parent(gary,ian)].
grandparent(beth,ken)<-[parent(beth,fiona),parent(fiona,ken)].
grandparent(beth,jake)<-[parent(beth,fiona),parent(fiona,jake)].
grandparent(beth,ian)<-[parent(beth,fiona),parent(fiona,ian)].
grandparent(andy,ken)<-[parent(andy,fiona),parent(fiona,ken)].
grandparent(andy,jake)<-[parent(andy,fiona),parent(fiona,jake)].
grandparent(andy,ian)<-[parent(andy,fiona),parent(fiona,ian)].
```

The power of being able to specify a domain and essentially introducing first-order logic principles to ABA, is that the same framework can now be used to represent other families or can be extended to accommodate the growth of this family, by simply changing the variable values, rather than having to define new rules. The "grandparent" relationship is defined only once, but since it is a global relationship it is grounded over all viable instances, as can be seen in [TODO].

This is just one example of a run of the grounder for further examples please refer to the evaluation section [TODO] that explains how the grounder as evaluated. Additionally, in section [TODO], we explain how the grounded framework is provided to the user from the web application for review purposes.

Chapter 6

Implementing Ideal Semantics dispute derivations.

When analysing an ABA framework semantics play an important roles. As explained in section [TODO] Background, when finding a derivation for a claim the semantics by which we do so must be specified. The current derivation engines Proxdd and Grapharg support derivation based on Grounded and Admissible semantics (as defined in [TODO]). As part of our implementation I looked into extending the Proxdd engine to support derivations based on ideal semantics. For an overview of ideal semantics refer to section [TODO].

6.1 Extending Proxdd to include ideal semantics.

Ideal Semantics dispute derivations are heavily based on the admissible semantics based dispute derivations. Proxdd already implements the such a derivation using the sxdd dispute derivation algorithm as described in [TODO]. This algorithm can be adapted to provide a derivation based on ideal semantics by extending it as in the algorithm described at [TODO]. There a are two adaptations to be made to the ab-derivation algorithm so that it can compute ideal semantics based dispute derivations:

1. Extend the algorithm to compute (P,O,D,C,Attacks,Arguments,F) tuples instead of the current (P,O,D,C,Attacks,Arguments) tuples and update F according to the algorithm in section [TODO].

2. Implement the *Fail(S)* check as explained in section [TODO]. This

check is then implemented in an extra step in the algorithm as described in section [TODO].

6.2 Implementing ideal semantics.

When implementing the ideal semantics the process was broken out in three steps:

1. Append the “F” list to the tuples and update it correctly.
2. Implement the *Fail(S)* check procedure.
3. Bring everything together in the final derivation engine.

Our ability to modularise the implementation process was largely dependent on the fact that the implementation is an extension of the ab-dispute derivation algorithm already implemented. This implied that each of the three steps could be implemented and tested individually before connecting the parts in our final derivation engine. The sections that follow ([TODO]) elaborate on the changes that were required, the implementation process and difficulties faced when implementing.

Updating the F set.

The first step was also the step that was less invasive to the current implementation of the derivation engine. This involved appending the “F” set to the tuples used by the sxdd algorithm as described in definition [TODO]. Additionally the updating of the “F” set was not invasive to the rest of the already implemented sxdd algorithm. This can be seen in the algorithm in section [TODO]. Therefore, “F” could be included and updated throughout Proxdd’s sxdd algorithm without interfering with ab-dispute derivations or gb-dispute derivations. The “F” set would be updated at any derivation but its content would be irrelevant unless the user specified that ideal semantics were desirable.

Updating the “F” set in accordance with the algorithm in most cases is straight forward. Consider the case where it is the Proponent’s turn. In this case the “F” set remains unchanged. As in step [TODO] of example [TODO]. The updating process becomes more complicated when it is the Opponent’s turn. On most cases (as seen in the algorithms definition [TODO]) the “F” set is updated by adding the sentences that are unmarked. Marking of sentences is already handled by the existing sxxdd algorithm and we can therefore use the current implementation to add add just the corret sentences.

The most complicated updating of the “F” set takes place at step [TODO]. An update at this step is illustrated by the example [TODO].

[TODO] - Add example that updates the F set accordingly.

Example 10. Consider the following assumption based framework:

R - includes the following rules:

z←a
z←b
y←a
x←d
v←c

A = {a,b,c,d} and $\bar{a} = z, \bar{b} = y, \bar{c} = x, \bar{d} = v$.

Step	P	O	A	C	F
1	{z}	{}	{}	{}	{}
2	{b}	{}	{b}	{}	{}
3	{}	{{x}}	{b}	{}	{}
4	{z}	{}	{b}	{a}	{{a}}
5	{}	{}	{b}	{a}	{{a}}
6	{}	{}	{b}	{a}	{}

Table 6.1: Example of IB-derivation of example [TODO]

Having appended the “F” set we can then test that it is updated correctly, in accordance to the algorithm, without having to implement any of the next steps. This can be done using the ab-dispute derivation semantics and checking the value of the “F” set at each step. The “F” set would simply be ignored and the derivation should still return a valid derivation.

It should be noted that, according to the algorithm in [TODO], the “F” set is also update when it is F’s turn and a set of sentences is selected from “F”. However, this case was implemented in the third step as it would not come into effect until we need to become invasive with the current implementation.

Carrying out the ***Fail(S)*** check.

The purpose of including an “F” set that is being updated during the derivation process, is so that we can run the *Fail(S)* check over the set of sentences included in “F”. *Fail(S)* is referred to by [TODO] reference as the Fail-dispute derivation of a multiset of sentences. The *Fail(S)* check was implemented in accordance to the algorithm defined in [TODO] and in accordance to [TODO] reference.

[TODO] Add paragraph about what exactly it checks, ask Toni at meeting.

Similarly the *Fail(S)* check plays a Proponent/Opponent games similarly to how the dispute derivation games are played and similarly produces tuples of the structure (P,O,D,C). In fact the new tuple is created using steps from the algorithm in [TODO] as defined in the fail-dispute derivation algorithm in [TODO].

Having implemented the various cases and steps the ideal semantics algorithm was tested by running each step with dummy input and checking whether the output would reflect the expected output, as illustrated by example [TODO]. This ensured the validity of the tuples generated. Having established the validity of these steps we then incorporated them together in a recursive dispute derivation game (see algorithm in [TODO]). This enabled us to check *Fail(S)* as a whole by again inputting a dummy case and checking whether the derivation process was carried out and completed. In this case, due to the semantics being implemented in Prolog, the *Fail(S)* check was expected to either fail or return true if it succeeded, depending on the input we provided. Nonetheless, to ensure the validity of the derivation the process was checked at each step to ensure that the tuples were being generated as expected.

Example 11. The example below is based on the frame work in example [TODO]. It displays the fail-dispute derivation of {a} step-by-step.

$\mathbf{D}_0$ = {({a},{},{a},{})}
$\mathbf{D}_1$ = {({},{{z}},{a},{})}
$\mathbf{D}_2$ = {({},{{a},{b}},{a},{})}
$\mathbf{D}_3$ = {({},{{},{b}},{a},{})}
$\mathbf{D}_4$ = {}

In the above the example we can see that the fail-dispute derivation terminates successfully as $\mathbf{D}_4 = \{\}$. Therefore, for {a} a fail-dispute derivation is possible. Provided that the check succeeds in all the other elements of the “F”- set, then the derivation is valid under ideal-semantics.

The current implementation has potential to be optimised further to allow for faster Fail-dispute derivation checks. This would be especially useful in larger scale frameworks. Although the functionality is accurate there is still room for improvement in some areas. One such area can be the selection process by which the algorithm chooses which sentence or set of sentences to check at each turn. For the ab-derivation in Proxdd there is a a selection process that implements an ordering on these sentences before it chooses. Such an optimisation has not been implemented in the current implementation. Currently the Fail-dispute derivation chooses sentences simply by picking off the head of the list each time.

Adding the Ideal Semantics to Proxdd.

The final step involved seamlessly incorporating these new features in the current implementation of Proxdd. The derivation engine uses flags such as "set_ab." and "set_gb." to allow the user to specify which semantics to be used. Naturally, a similar "set_ib." flag was implemented to allow the user to specify the use of ideal semantics.

The major change of the ib-dispute derivation is the inclusion of a "new player" in the dispute derivation game, the set "F". Similarly, to admissible dispute derivation the game is played until both P and O of the currently processed tuple are empty. This would imply the termination of the derivation process having successfully found a derivation solution for the claim provided under the semantics for admissibility. However, in ideal semantics we are looking for a specific subset of the admissible solutions which is defined by whether the *Fail(S)* check is satisfied for all sets in 'F" as defined in [TODO].

Therefore, the first design choice we need to make is when do we attempt to check whether the sets in "F" satisfy the Fail-dispute derivation check. Our solution to this problem is to let the ab-derivation run to completion (i.e both P and O are empty) and then, instead of terminating, identifying that it is F's turn to run the fail-dispute derivation check. Therefore, when incorporating the ideal semantics into Proxdd the "choose_turn" function was adapted as shown in definition [TODO].

```
choose_turn([], [], _, fail) :-
 !.
choose_turn([], _, _, opponent) :-
 !.
choose_turn(_, [], _, proponent) :-
```

```
 !.
choose_turn(P, O, _, Player) :-
 option(strategy(turn_choice), Strategy),
 turn_choice(Strategy, P, O, Player).
```

The above code segment shows how we re-implemented the “choose_turn” function to accommodate running ideal-based semantics. We implemented an extra case that allows for the selection of F’s turn.

By implementing F’s turn we now have an implementation of the algorithm that finds an admissible derivation and then checks whether this solution is also in accordance to the ideal semantics, as specified in algorithm [TODO].

Additionally to the “choose_turn” function the “derivation” function had to be adapted as well. Specifically, the base case had to be adapted as shown in [TODO].

```
derivation([[],[],D,C,At,Ar,[]], _, _,[D,C,At,Ar,[]]) :-
 !.
derivation(T, N, L, Result) :-
 derivation_step(T, L, T1, L1),
 poss_show_step(N, T1),
 N1 is N + 1,
 derivation(T1, N1, L1, Result).
```

The extension of the base case now ensures that the F set is also empty before the derivation terminates. This ensures that our derivation under Ideal semantics has been completed successfully.

Following this we also had to change the “derivation_step” function to run the necessary function when it is F’s turn. The updated function can be seen below:

```
derivation_step([P,O,D,C,At,Ar,F], L, T1, L1) :-
 choose_turn(P, O, F, Turn),
 (
  Turn = proponent
  -> proponent_step([P,O,D,C,At,Ar,F], L, T1, L1)
  ;
        (
         Turn = opponent
         -> opponent_step([P,O,D,C,At,Ar,F], L, T1, L1)
         ; fail_step([P,O,D,C,At,Ar,F], L, T1, L1)
         )
 ).
```

The derivation function has been adapted as shown in the code segment above in order to incorporate the *Fail(S)* check. Having adapted the function to choose when it is F's turn, we now adapted the "derivation" function to catch the case when it is F's turn and direct it to the correct step that has to be carried out.

The current implementation of Proxdd terminates the derivation successfully if both P and O are empty. However as noted already in ideal semantics this is not enough. The derivation base case has been adapted so as to terminate successfully when the "F" set is empty as well. This implies that each set of sentences in the "F" set has successfully passed the *Fail(S)* check and has been removed from "F" in accordance to the algorithm defined in [TODO].

An empty "F" set at the end implies that the solution found is a valid derivation under ideal semantics. If the "F" set is not empty then the base case of the derivation function is never reached, Prolog fails the derivation and attempts a different solution. Thus, enforcing dispute derivations under ideal semantics.

6.3 Example of ideal derivation.

Chapter 7

Building a web application interfacing the derivation engines.

The following section focuses on providing an overview of the solution implemented. It also explores various design and implementation choices made. Lastly, an evaluation of the current implementation and its alternatives is explored.

7.1 Overview of the Solution.

The web application's architecture is based on a 3-tier system that includes three distinct parts that make up the application. These parts of the application communicate between each other to provide the user with the final desired outcome. The system is made up from the Client, Server and Derivation Engines. The Server acts as a coordinator between the Client and the Derivation Engines. Users interact with the client side of the application which then communicates with the Server. The Server then process the request from the user and submits it to the eligible Derivation Engine. Once the derivation is complete the Server process the solution and sends it back to the client as illustrated in [TODO].

7.1.1 Client

The Client part of the application is the user's gateway to the argumentation derivation engines. It provides the user with a GUI providing both the means required to submit an argumentation framework and the tools required to view the derivation tree received. It also allows the user to specify which engine and what semantics are to be used.

It was built using tools such ASP.Net and Javascript. ASP.Net, along with HTML was used to create the web pages the user can interact with. Javascript and the D3.js library where used to provide a useful visualisation of the outcome to the user in the form of a derivation tree.

7.1.2 Server

The Server part acts as a coordinator between the Client and the Derivation Engines. Within the Server the input from the Client is processed and prepared to be given to the necessary derivation engine to be analysed. The Server part carries out the following functionality:

- Analyses the parameters from the user and selects the correct derivation engine and semantics.
- Parses the user's input, processes it and generates the relative input for the derivation engine.
- Grounds the framework provided by the user over the domain specified.
- Parses and analyses the output from the derivation engine and builds a JSON string that is passed to the client to build the derivation tree visualisation.

7.1.3 Derivation Engines

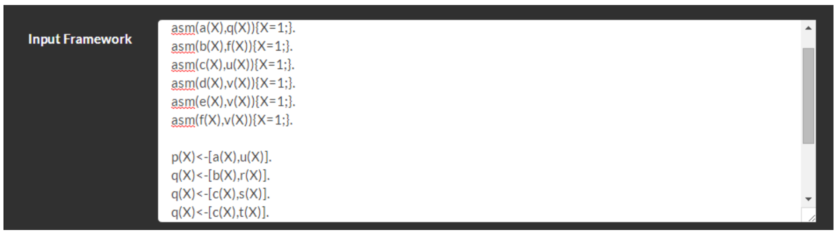
The Derivation Engines used for the current implementation include Proxdd and Grapharg which are both implemented in Prolog. The correct derivation engine is initialised with the correct parameters and provided a suitable input it outputs a correct derivation. The derivation engines can be seen as separate entities from the rest of application which implies a plug-in plug-out possibility. By providing a module within the server that interprets and formulates the input and output to the engine, the application should be able to handle the introduction of any new derivation engine it its back end.

7.2 User Interface with Application.

The application, like most web applications, carries out most of the intensive processes in its back end. Therefore, the user interface module carries out little of the functionality of the application, but instead focuses on allowing the user to interact with the engines. The GUI controlled by the User part of the application carries out two tasks. It allows users to input their framework and parameters and it constructs the derivation tree of the solution.

7.2.1 Setting up the framework.

Since the application derives the solution of a user defined argumentation framework, then the user should have the means to input said framework. The GUI provides a text box in which the user can define their ABA framework using a simple language that allows them to define assumptions, contraries and rules over a specified domain. A simple example of such an input can be seen at [TODO]:



Input Framework

```
asm(a(X),q(X)){X=1;}.
asm(b(X),f(X)){X=1;}.
asm(c(X),u(X)){X=1;}.
asm(d(X),v(X)){X=1;}.
asm(e(X),v(X)){X=1;}.
asm(f(X),v(X)){X=1;}.

p(X)<-[a(X),u(X)].
q(X)<-[b(X),r(X)].
q(X)<-[c(X),s(X)].
q(X)<-[c(X),t(X)].
```

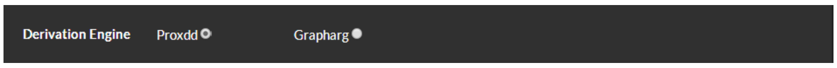
Figure 7.1: Defining framework and targeted conclusion.

The formal context free grammar of the input language along with how it is parsed is explained in detail in section [TODO].

7.2.2 Configuration of the engines.

Additionally the users should be able to choose the configuration of the derivation engine to be used. This includes two main decisions, the engine to be used and the semantics to be used.

The users can choose the engine required by checking the check-box as in example [TODO].



Derivation Engine: Proxdd ○ Grapharg [✓ ●]

Figure 7.2: Choosing derivation engine.

The user can then choose the semantics required by selecting the check-box corresponding to the required semantics, as in example [TODO].

[COULD INCLUDE JUST ONE IMAGE WITH ANNOTATION]

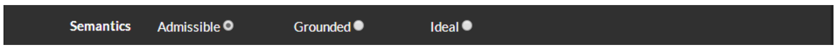


Figure 7.3: Choosing semantics for derivation.

Lastly, the user must specify the claim for which a derivation is required as in example [TODO].

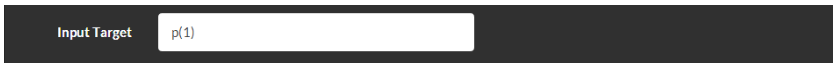
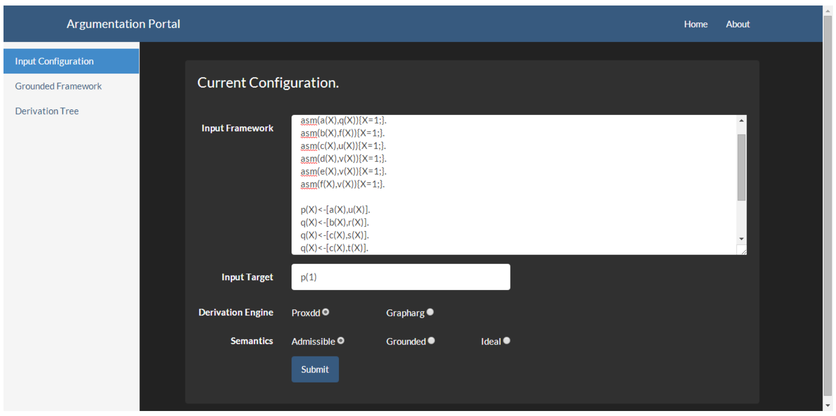

Input Target: p(1)

Figure 7.4: Declaring a targeted conclusion.

An example of the input form completed in its entirety can be seen at example [TODO]



Input Framework

```
asm(a(X),q(X)){X=1;}.
asm(b(X),f(X)){X=1;}.
asm(c(X),u(X)){X=1;}.
asm(d(X),v(X)){X=1;}.
asm(e(X),v(X)){X=1;}.
asm(f(X),v(X)){X=1;}.

p(X)<-[a(X),u(X)].
q(X)<-[b(X),r(X)].
q(X)<-[c(X),s(X)].
q(X)<-[c(X),t(X)].
```

Input Target: p(1)

Derivation Engine: [✓ Proxdd] Grapharg

Semantics: [✓ Admissible] Grounded Ideal

Figure 7.5: Example of completed input configuration form.

7.2.3 Visualisation the derivation tree.

Once the input provided has been analysed by the derivation engines then the solution found is provided to the user as a derivation tree. Once the Server side finishes processing the solution it provides a JSON string that is interpreted client-side by javascript and the D3.js visualisation library. The JSON string has a structure as illustrated in example [TODO]. This structure specifies the characteristics of each node in the tree and the edges between the nodes.

```
{
        "nodes":
        [
        {"id":"s0_0","name":"p(1)","shape":3,"group":0},
        {"id":"s0_1","name":"a(1)","shape":0,"group":1},
        ]
        ,"links":
        [
        {"source":1,"target":0,"value":1,"group":0},
        {"source":3,"target":2,"value":1,"group":0},
        {"source":2,"target":1,"value":1,"group":1},
        ]
}
```

On the client side the JSON is processed and the derivation tree it represents is illustrated in the canvas area of the web page as shown in example [TODO]. The tree is annotated and colour coded for clarity and it also allows the user to zoom in and out.

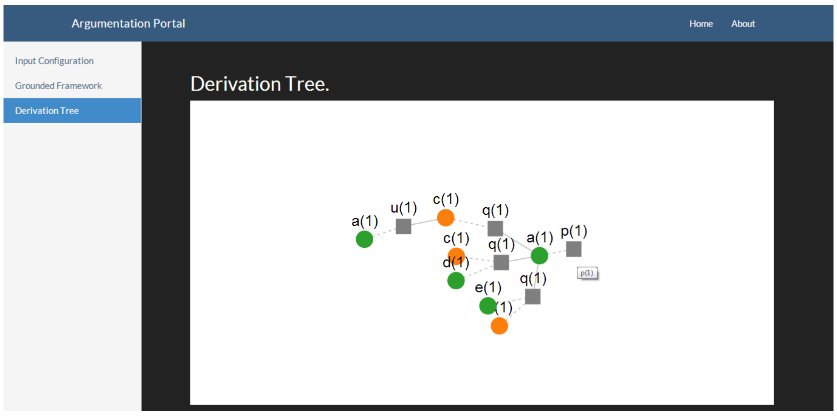


Figure 7.6: Visualisation of output as derivation tree.

7.3 Server - Processing the Input.

Once the user specifies the framework it then has to be passed to the server to be processed. Within the server several steps take place concerning the processing of the framework. These allow the interpretation of the input and the generation of the corresponding input for the engine. These steps include:

- Parsing the Input and seperating the various elements (as described in [TODO]).

- Checking the validity of the input (as described in [TODO]).
- Grounding the input over a certain domaind (as desribed in [TODO]).
- Generating the derivation engines input (as described in [TODO]).

Due to the modularity of the web application it is possible that new derivation engines could be added to the back-end by implementing alternative to these steps. For example, if a new engine is added than by switching to a different generator we can now use the parsed and grounded input to create an input viable for the new derivation engine.

7.4 Server - Interfacing with the Prolog Engine.

Both Proxdd and Grapharg are implemented in the Prolog programming language which is commonly used in the field of AI. However, it has not been developed with web application development in mind which would hinder any attempts to extend the web application in the future if the whole of the server implementation was carried out in Prolog.

7.4.1 Need for an interface between C# and Prolog Engines.

The following decision choices were all possible:

- Re-implementing both Proxdd and Grapharg in C#
- Using an external library that provides an interface for C# with Prolog.
- Implementing the back-end of the web application in Prolog.

Nonetheless, for reasons explained in more detail in section [TODO] the use of an external library seemed to be the most reasonable choice. The interface allows for seamless integration of Prolog and C#. Additionally, it enables us to focus most of the development work in a programming framework that is more popular and extendible than Prolog.

7.4.2 Using the SwiPIC.dll library as an interface.

The SwiPIC.dll library was chosen as it a provide a simple interface from C# to the SWI-Prolog. It allows for the initialisation of an Instance of the SWI-Prolog Engine and we can then build Prolog queries that can be run. The code snippet at [TODO] provide an example of how the Prolog Engine is initialised and how a query is run:

```
if (!PlEngine.IsInitialized)
{
    String[] param = { "-q", "-f", runProlog };
    PlEngine.Initialize(param);

    using (PlQuery q = new PlQuery("loads(" + termList + "),
                    sxdd(" + claim.Text + ",X,Y)"))
    {
            int idx = 0;
        foreach (PlQueryVariables v in q.SolutionVariables)
        {
                solution.Add(v["Y"].ToString());
        }
    }
    PlEngine.PlCleanup();
}
```

By using an external library such as this we can easily interact with the derivation engines implemented in Prolog by submitting the query we have constructed based on the user's input and extracting the solution returned by the engine. The interface also enables us to implement the bulk of the application in C# and then use external libraries such as this to interact with any other derivation engines in the future, while keeping the bulk of the implementation common.

7.5 Choosing the current implementation.

There are often enough more than one correct ways of designing and implementing a system. Related decisions are often taken based on the priorities of the project and other limitations. In this subsection I look into justifying some of the design decisions taken and potential disadvantages. Additionally we will look into one of the proposed alternatives for the derivation engines.

7.5.1 Advantages of current implementation.

The main advantage of the 3-tier system is that we provide modularity to the system allowing us to separate various significant elements of the system. In our implementation most of the processing other than the derivation is carried out in the Server tier by the C# code. This makes it highly extendible as a simple switch can be implemented in the Server tier that enables the user to choose from an array of engines. The implementation, being modular, does not require extensive changes in the code to accommodate new derivation engines. As long as an interpreter is created to generate the required JSON string for the client side and the user input is translated to the correct corresponding input for the derivation engine, then the rest of the application should remain unchanged.

In addition to this the modular implementation allows the incorporation of further extensions to the application as part of the pre-processing process in the Server module. As an example, consider that there is a direct mapping that can be implemented between Abstract Argumentation and Assumption Based Argumentation. This mapping can be implemented in a module that is set to run as part of the pre-processing done in the Server module. This can be added to the process in a seamless manner, by which it can be enabled or disabled when required without any additional code changes.

Lastly, the choice to interact with the Prolog engine through an interface rather than the alternatives suggested was done for the purposes of usability and extensibility. Details of this approach are discussed in the last subsection [TODO].

7.5.2 Disadvantages of current implementation.

One of the disadvantages of the current implementation is its reliance on rather exotic external libraries to interface and run the Prolog derivation engines. There is the potential that if the web application is further developed in the future an updated version of this library might be required. However, as the library is not one of the core libraries in C# such an update might take long to be released. Nonetheless, this was taken into account when deciding whether to used this library or not. The application deliberately tries to limit the use of this library to just initialising the engine, submitting a query and extracting the solution. The specifics of the query, for example, are built without the use of this library. Additionally, when deploying the application the environment needs to be configured to account for these libraries.

Another disadvantage of the current implementation is that there is currently a significant amount of processing required to generate the data communicated between the tiers in the required format. When a user inputs a framework this must then be translated in the required form for the input of the derivation engine. Once the derivation engine complete the solution then has to be interpreted and the corresponding JSON string must be generated. This takes processing time. An alternative would be to adapt the derivation engines to directly produce the JSON string rather than an alternative output. However, it would be unrealistic to expect derivation engines to conform to these standards. Instead, the current implementation allows for better integration of new engines rather than processing power.

7.5.3 Proposed alternative (Prolog Web Server).

There is one exceptionally interesting alternative implementation that was considered during the design process. SWI-Prolog which is the flavour of Prolog used in our implementation also provides the ability for it to run as a Web Server on its own. The proposed alternative is to establish a web server based on SWI-Prolog that would also include the derivation engines. This web server can then be configured to listen on certain ports for requests that it would be able to handle and reply to. Our web application would then be able to send over the required parameters of the derivation in the form of an Http Request to the Prolog Server. The Prolog Server would then carry out the derivation process and respond with the solution. A major potential advantage of this implementation would be that the Prolog Server could be used by further applications in the future. Requests can be made by various applications and these would be handled accordingly by the Prolog Server.

Nonetheless, there are issues that had to be considered when rejecting this alternative. Configuring the web server and the firewalls between it and our web application would be a very cumbersome task, which could be avoided. Also by creating a completely distinct web server to handle the derivation we are relying our web application on two servers. This creates two potential hazard points for our application. If the Prolog Web Server happens to stop working or face a fault then the web application will not work properly. By integrating Prolog through C# we reduce this potential weakness point. Lastly, the idea of implementing the whole Server side of the application on a Prolog Web Server was rejected as Prolog is not a well supported web development framework such as C# and ASP.Net. This would provide problems when having to extend the application to include new engines which might be built in other programming languages and frameworks such as C++, C, Pythonm etc.

Chapter 8

Evaluation of the System.

The web application was tested and evaluated to ensure the quality of the final system. This involves using qualitative and quantitative methods to ensure the system is performing as expected. Evaluation focused on the following three key areas:

- Usability
- Robustness
- Performance

In the sections that follow we discuss how the above were tested for various levels of the system.

8.1 Evaluating GUI Usability.

One of the key requirements for the web application was to allow user to easily be able to interact with it. This involved designing a system that would be simple and clear to use. It should present the users with all the required information, abstracting them from the complexities and unnecessary details of what goes behind. Our aim was to create an application that looks and feels good, but also makes the definition of a derivation problem as clear and simple as possible.

8.1.1 User testing and feedback.

To ensure that the user experience was satisfactory we relied on user feedback to guide any design changes implemented. As the project progressed test users were given access to the graphical user interface in order to provide feedback about the design itself and additional features that might be useful. Test users included were from a wide array of backgrounds that included both people with experience in ABA frameworks, but also users that were new to ABA. This was accomplished by relying on a group that included colleagues, friends, family and of course the supervisors of the project.

By allow the testing with users with a diverse understanding of ABA it was easier to gain feedback both on matters that technically adept users might consider important and on the general perception of the application by the public. The feedback gained was used to drive the design changes throughout the development process.

8.1.2 Initial Concept.

The project always aimed at providing a simple and understandable user interface. This implied an easy way to input a framework, configure the application and understand the output. Even though the web application is minimal and has a simple user interface, the end—product has had significant changes from the initial concept.

Initially the user interface aimed at providing the user with all the available interactions on a single page. This aimed at allowing the users to view the result, edit the framework and change the configuration all on the same page. However, this lead to an over—cluttered and confusing interface as realised from user feedback.

Additionally, various methods were considered for the input mechanism that would be implemented in order to allow the users to specify frameworks. In an attempt to make the process easier one of the ideas considered was to be able to declare rules and assumptions one—by—one using a respective for for each that asked the user for the required information (head of the rule, predicates, variable, etc). This would make it easier for users with little or no programming experience to use the application. However, user feedback showed that such an approach would be very cumbersome in terms of time. For small examples for educational purposes such an approach could potentially work, but the web application was designed in order to handle any sort of framework. As realistic ABA frameworks tend to be constructed from numerous assumptions and rules, an input mechanism had to be chosen that allowed users to easily define frameworks of any size.

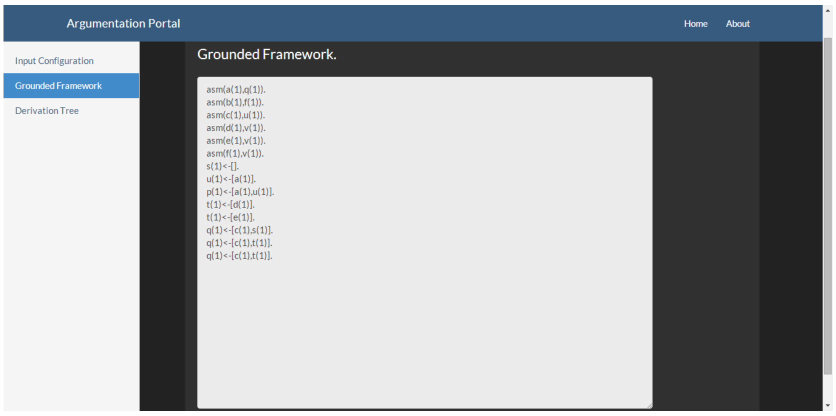
Lastly, based on the feedback received from these test users additional features were added to the web applications. These are features that make the web application easier to use or provide the more technical users with exposure to the inner—working of the web application (as is the case with the grounded described in section [TODO]).

8.1.3 Changes to GUI.

Building on the feedback from out test users we proceeded to evaluate possible alternatives and implementations that would benefit them. Some of the most major contributions from the user feedback that were implemented are discussed below.

Include ability to see grounded framework.

For the more technical users access to the grounded framework can be a desirable feature. The user can look over the grounded framework and review it. This allows users to find other suitable targets they can get a derivation for and can also evaluate whether certain rules or assumptions exist as under the domain they specified.



```
asm(a(1),q(1)).
asm(b(1),f(1)).
asm(c(1),u(1)).
asm(d(1),v(1)).
asm(e(1),v(1)).
asm(f(1),v(1)).
s(1)<-[].
u(1)<-[a(1)].
p(1)<-[a(1),u(1)].
t(1)<-[d(1)].
t(1)<-[e(1)].
q(1)<-[c(1),s(1)].
q(1)<-[c(1),t(1)].
q(1)<-[c(1),t(1)].
```

Figure 8.1: Example of the panel displaying the grounded framework.

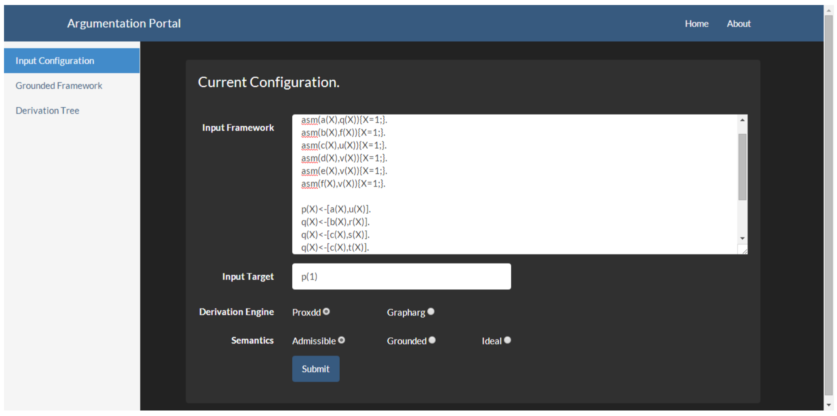
As shown in figure [TODO] the output is in a format almost identical to the format the user used ton define the framework in the first place. Additionally, when grounding a framework over an extensive domain it is expected that there might be a significant increase in the number of rule and assumption definitions. Therefore, the window provided is both scrollable and resizeable. Furthermore it is also read—only to avoid any tampering with the grounded framework.

Tabulated sections.

Following from the feedback gathered that described the initial interface as over—cluttered and confusion, we separated the sections in easily reachable and distinct panels that each serves a specific purpose. The section are:

- Input Configuraion.
- Grounded Framework.
- Derivation Tree.

All the tabs exist in the same session and switching between tabs is handled client—side by selecting the appropriate section from the navigation panel on the left as shown in image [TODO]. This allows for easy and simple navigation between the various aspects of the web application. It also provides the users with ability to change the input framework and the configurations on the fly.



Input Framework

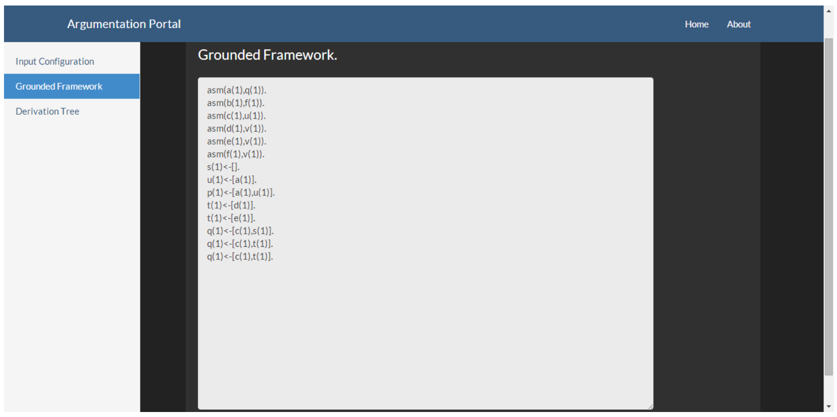
```
asm(a(X),q(X)){X=1;}.
asm(b(X),f(X)){X=1;}.
asm(c(X),u(X)){X=1;}.
asm(d(X),v(X)){X=1;}.
asm(e(X),v(X)){X=1;}.
asm(f(X),v(X)){X=1;}.

p(X)<-[a(X),u(X)].
q(X)<-[b(X),r(X)].
q(X)<-[c(X),s(X)].
q(X)<-[c(X),t(X)].
```

Input Target: p(1)

Derivation Engine: [✓ Proxdd] Grapharg

Semantics: [✓ Admissible] Grounded Ideal

Figure 8.2: The input configuration tab.



```
asm(a(1),q(1)).
asm(b(1),f(1)).
asm(c(1),u(1)).
asm(d(1),v(1)).
asm(e(1),v(1)).
asm(f(1),v(1)).
s(1)<-[].
u(1)<-[a(1)].
p(1)<-[a(1),u(1)].
t(1)<-[d(1)].
t(1)<-[e(1)].
q(1)<-[c(1),s(1)].
q(1)<-[c(1),t(1)].
q(1)<-[c(1),t(1)].
```

Figure 8.3: The grounded framework tab.

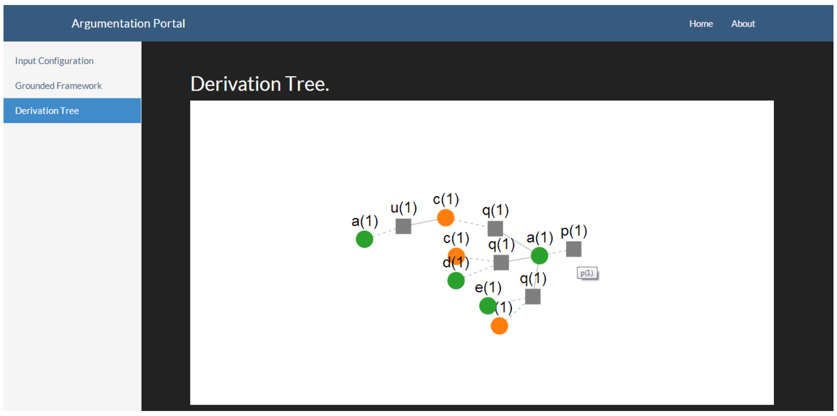


Figure 8.4: The derivation tree tab.

Bootstrap based GUI.

Lastly, the general design and colour scheme used for the web application was deemed uncreative and dull. Therefore, to ensure a better user experience and to instill faith in the system's capabilities the user interface was re—designed to a more polished—up look. This was done using a popular tool that allows for the creation of good looking web application and websites called Bootstrap. Although the underlying functionalities of the web application are exactly the same, the final looked received better feedback from users, who now seemed more confident using the systems. The difference can be seen in figures [TODO] and [TODO].

[TODO] - Add image of before.

[TODO] - Add image of after.

8.2 Evaluating the Grounder.

Ensuring the validity of the grounder is important in ensuring the validity of the whole system. As a preprocessing step, if this is not carried out correctly then we risk passing to the derivation engine wrong or even invalid input. Therefore, we need to ensure the robustness of our grounding mechanism with both large and small frameworks. Specifically during the evaluation process for the grounder we seeked to verify the following:

- Correct Grounding of frameworks over their domain.
- Robustness when dealing with large scale frameworks.
- Satisfying performance for our grounder.

8.2.1 Checking validity through examples.

Initially we tested the grounder by running the examples we used to test the derivation engines in general, adapted however to include domains. This means that the example frameworks now have to be grounded first before they get provided to the derivation engine. By isolating the grounded we were able to provide the input for which we knew the expected output and then compare this to the actual output received.

There are three cases which we needed to check how they were grounded. These represent the three valid forms of the commands we can provide based on the language we have defined. These are:

- *asm(a(X),b(X))* - This defines an assumption that exists on the domain specified by the user.
- *a(X)<-[]* - This defines a fact. A fact is handled by the grounder similarly to assumptions.
- *a(X)<-[b(X)]* - This defines a rule which can exist if the predicates in the detail exist in the global domain of our framework.

The examples we used had to accommodate and test all of these cases. Consider example [TODO] which is one of the examples used.

```
b(X,A)<-[a(X,A)].
asm(a(X,Y),d(Y)){X=1,2;Y=2,3;}.
c(X,Y,Z)<-[]{X=1,3;Y=2,3;Z=Michael,John;}.
```

```
asm(a(1,2),d(2)).
asm(a(2,2),d(2)).
asm(a(1,3),d(3)).
asm(a(2,3),d(3)).
c(1,2,Michael)<-[].
c(3,2,Michael)<-[].
c(1,3,Michael)<-[].
c(3,3,Michael)<-[].
c(1,2,John)<-[].
c(3,2,John)<-[].
c(1,3,John)<-[].
c(3,3,John)<-[].
b(2,3)<-[a(2,3)].
b(1,3)<-[a(1,3)].
```

```
b(2,2)<-[a(2,2)].
b(1,2)<-[a(1,2)].
```

This way we ensured that the grounding was carried out successfully.

8.2.2 Random Framework generator.

Later on in our evaluation of the grounder we explored the possibility of test running the grounder using large random frameworks. This would enable us to both check the validity of the grounder, but also to test its performance against frameworks of a larger scale.

How the random generator works.

However, creating such frameworks by hand is cumbersome and prone to errors. Therefore, we devised an algorithm for pseudo—random framework generations which is described in [TODO].

[TODO] - Pseudo code for random framework generation.

```
Procedure genRandFramework()
        int[] domainA
        int[] domainB
        Dictionary assumCont
        Queue assumQ
        Queue ruleQ
        List framework
        List frameworkAssums
        List frameworkRules
        List startSymbols = getStartList()
        List availableSymbols = startSymbols

        ruleQ.Enqueue("a")

        while( (assumQ and ruleQ not empty)
                        and availableSymbols not empty)
                if(assumQ not empty)
                        assum = createAsm(assumQ.Dequeue,
                                startSymbols,assumCont)
                        frameworkAssums.Add(assum)
                else
                        rule = createRule(ruleQ.Dequeue,
                                availableSymbols,assumQ,ruleQ)
                        frameworkRules.Add(rule)

        foreach assumption in frameworkAssums
                framework.Add(assumption)

        foreach rule in frameworkRules
                framework.Add(rule)
end

Procedure createAsm(id,startSymbols,assumCont)
        /* Pick Random Contrary from startSymbols */
        /* Randomly choose between possible domains */
        /* Add assumption and contrary pair to Dictionary */
        /* Construct assumption and return */
end

Procedure createRules(id,availableSymbols,assumQ,ruleQ)
        /* Randomly choose number of terms in rule's */
        /* tail (between 1-4) */
        /* Randomly pick terms's Ids from availableSymbols */
        /* and remove from availableSymbols */
        /* Randomly add each of the tail terms to either */
        /* assumQ or ruleQ */
        /* Construct Rule and return */
end
```

Grounding large scale frameworks.

Having implemented this algorithm we were now able to create random frameworks and check whether they were grounded correctly. Additionally we used the generation of large frameworks to test the performance.

Example [TODO] shows an example of the ungrounded framework and example [TODO] show the respective grounded framework. As the large scale frameworks would be hard to provide as part of this report, we take a smaller framework to be indicative of the process carried out with the larger frameworks as well. In these examples we consider a framework made up of approximately 10-20 commands. For a larger scale examples used during testing please refer to Appendix [TODO]

```
asm(c(X),u(X)){X=1,2,3,4,5,6,7;}.
asm(e(X),u(X)){X=1,2,3,4,5,6,7;}.
asm(f(X),p(X)){X=1,2,3,4,5,6,7;}.
asm(y(X),r(X)){X=5,6,7,8,9;}.
a(X)<-[h(X),u(X)].
h(X)<-[c(X)].
u(X)<-[q(X)].
q(X)<-[w(X)].
w(X)<-[e(X),s(X)].
s(X)<-[j(X)].
j(X)<-[r(X),l(X)].
r(X)<-[k(X)].
l(X)<-[f(X),v(X)].
k(X)<-[t(X)].
v(X)<-[n(X),b(X)].
t(X)<-[d(X),x(X),m(X)].
n(X)<-[p(X)].
b(X)<-[i(X),o(X)].
d(X)<-[y(X),g(X)].
x(X)<-[z(X)].
```

```
asm(c(1),u(1)).        asm(c(2),u(2)).
asm(c(3),u(3)).        asm(c(4),u(4)).
asm(c(5),u(5)).        asm(c(6),u(6)).
asm(c(7),u(7)).        asm(e(1),u(1)).
asm(e(2),u(2)).        asm(e(3),u(3)).
asm(e(4),u(4)).        asm(e(5),u(5)).
asm(e(6),u(6)).        asm(e(7),u(7)).
asm(f(1),p(1)).        asm(f(2),p(2)).
asm(f(3),p(3)).        asm(f(4),p(4)).
asm(f(5),p(5)).        asm(f(6),p(6)).
asm(f(7),p(7)).        asm(y(5),r(5)).
asm(y(6),r(6)).        asm(y(7),r(7)).
asm(y(8),r(8)).        asm(y(9),r(9)).

h(7)<-[c(7)].
h(6)<-[c(6)].
h(5)<-[c(5)].
h(4)<-[c(4)].
```

```
h(3)<-[c(3)].
h(2)<-[c(2)].
h(1)<-[c(1)].
```

8.2.3 Performance of Grounder.

[TODO] - Performance analysis.

As explained in section [TODO] there are optimisation mechanism that can be implemented to improve the performance

8.3 Evaluating the ideal semantics implementation.

During the implementation of the project I also got the opportunity to work with the derivation engines directly. The aim was to extend the Proxdd derivation engine to be able to accommodate derivations based on ideal semantics as described in section [TODO].

In order to be able to add the extended derivation engine to the web application we first had to ensure that the required functionality was carried out properly. In order to do so the semantics had to be tested with several example of which we had an expected outcome.

Due to the nature of definition of the ideal semantics derivation (as described in section [TODO]) we were able to implement it in two stages. This also made the testing process easier. The evaluation of the validity of our ideal semantics derivation was broken down in the following steps:

1. Ensure the correct updating of the F set.
2. Test the cases for the Fail(S) derivations.
3. Test the complete implementation with examples.

8.3.1 Testing in Swi—Prolog.

The design of the web application is such by which changes in the derivation engines can be carried out without affecting the rest of the application. Therefore, our implementation of the ideal semantics could easily be tested directly in Swi-Prolog through its console to ensure its validity and only add it to the existing project one we were sure it was safe.

This approach provides us with various advantages:

- Ability to debug using Swi-Prolog debugging tools.
- Ability to test the predicates that make up the derivation individually.
- Ability to easily test using predefined examples defined in files.

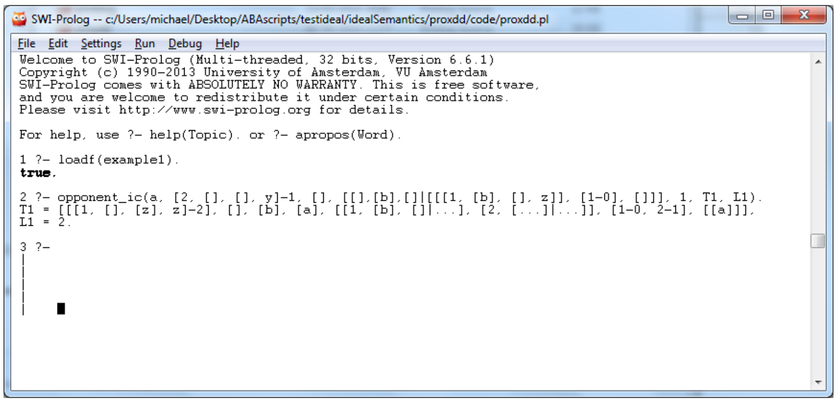
Since the application is unaffected by the required changes to implement the ideal semantics, we were able to carry out the testing using the Swi-Prolog console. We the simply integrated the derivation engine including the ideal semantics in the existing web application.

Testing updating of F set.

The first step in implementing the ideal semantics is establishing and updating correctly the F-Set as described in section [TODO]. This requires extending the existing admissible derivation, but does not require changes that will alter the functionality of the admissible based derivations. Therefore, we can implement the F set as an extension of the admissible based derivation and check that it is updated correctly during an admissible derivation.

As defined in algorithm [TODO] the F set is updated depending on the turn and case we are facing at that part of the derivation, just like the other sets in the tuple are. Additionally, the algorithm itself is turn based. At every turn the tuple is updated accordingly and the whole process is repeated again until certain conditions apply. Therefore, we decided to check that at every possible case in every possible turn the updating of the F-set is done correctly.

This is done by using the Swi-Prolog console to test the updating of the tuple at every possible case of the derivation algorithm individually. This is done by providing a tuple as an input for which we are aware of the expected output. Constructing such examples was simple as the logic used to update the set was primitive. Therefore, constructing such test tuples and finding their outputs was easily done by hand. Consider example in figure [TODO].



SWI-Prolog -- c:/Users/michael/Desktop/ABAscripts/testideal/idealSemantics/proxdd/code/proxdd.pl

```
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 6.6.1)
Copyright (c) 1990-2013 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- loadf(example1).
true.

2 ?- opponent_ic(a, [2, [], [], y]-1, [], [[],[b],[]|[[[1, [b], [], z]], [1-0], []]], 1, T1, L1).
T1 = [[[1, [], [z], z]-2], [], [b], [a], [[1, [b], []|...], [2, [...]|...]], [1-0, 2-1], [[a]]],
L1 = 2.

3 ?-
```

Figure 8.5: A verification run of opponent step case 2.i.c.1 according to algorithm [TODO].

In the above test we tested the case of the opponent, case 2.i.c.1 according to the algorithm provided in [TODO]. In example [TODO] below we can see the admissible derivation for the whole of the example [TODO]. The test we provided above as an example refers specifically to the updating of the F-set between steps 3 and 4. As it can be seen the last set in the tuple is identical to the expected value of F according to our expected derivation. In our example T1 refers to the tuple [P,O,D,C,Att,Arg,F] after the updating is carried out for the opponent step. Therefore the F set for T1 corresponds to "[[a]]" at the end of tuple which is in accordance to the expected output based on our derivation table in [TODO]. The other cases for both the opponent and the proponent, as proposed in the algorithm [TODO] were tested similarly.

By testing all of the checks individually we can ensure that the updating is carried out successfully at each case. This also makes it easier to detect and correct bugs relating to the updating. Once the updating has been verified then the rest should still be valid because it has already been implemented as part of the admissible derivations. The whole implementation so far was lastly tested by running full examples under admissible semantics and observing and verifying the updating of the F set during said derivations, as in example [TODO].

Example 12. Consider the following assumption based framework:

R - includes the following rules:

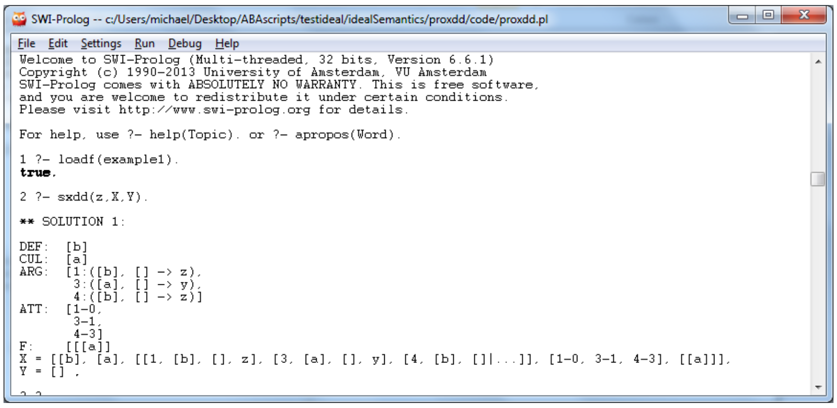
z←a

z←b
y←a
x←d
v←c

A = {a,b,c,d} and $\bar{a} = z, \bar{b} = y, \bar{c} = x, \bar{d} = v$.

Step	P	O	A	C	F
1	{z}	{}	{}	{}	{}
2	{b}	{}	{b}	{}	{}
3	{}	{{x}}	{b}	{}	{}
4	{z}	{}	{b}	{a}	{{a}}
5	{}	{}	{b}	{a}	{{a}}

Table 8.1: Example of IB-derivation of example [TODO]


SWI-Prolog -- c:/Users/michael/Desktop/ABAscripts/testideal/idealSemantics/proxdd/code/proxdd.pl
File Edit Settings Run Debug Help

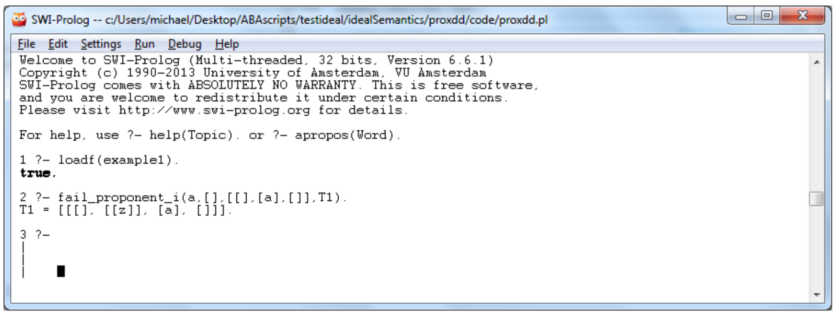
```
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 6.6.1)
Copyright (c) 1990-2013 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- loadf(example1).
true.

2 ?- sxdd(z,X,Y).

** SOLUTION 1:

DEF:  [b]
CUL:  [a]
ARG:  [1:([b], [] -> z),
       3:([a], [] -> y),
       4:([b], [] -> z)]
ATT:  [1-0,
       3-1,
       4-3]
F:    [[[a]]
X = [[b], [a], [[1, [b], [], z], [3, [a], [], y], [4, [b], []|...]], [1-0, 3-1, 4-3], [[a]]],
Y = [] .
```

Figure 8.6: A verification run of admissible semantics on example [TODO]. Note that the F-set has an end value as expected from the derivation table [TODO]

Testing Fail(S) derivations.

The approach described in the section above was similarly used to test the implementation of the Fail(S) derivations used in the ideal based derivations. The algorithm described in section [TODO] that carries out the Fail(S) derivation, involves the generation of tuples from other tuples in a turn based manner.

By implementing the algorithm as a series of predicates rather than just one large predicate we were able to test the validity of each predicate individually. This again implied providing a predefined tuple as an input and comparing the output to the expected output we derived by hand, as shown in example [TODO]


SWI-Prolog -- c:/Users/michael/Desktop/ABAscripts/testideal/idealSemantics/proxdd/code/proxdd.pl
File Edit Settings Run Debug Help

```
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 6.6.1)
Copyright (c) 1990-2013 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- loadf(example1).
true.

2 ?- fail_proponent_i(a,[],[[],[a],[]],T1).
T1 = [[[], [[z]], [a], []]].

3 ?-
```

Figure 8.7: A verification run of proponent case i for *Fail(S)* derivation.

In the above example we consider the case were the initial set of tuples provided is

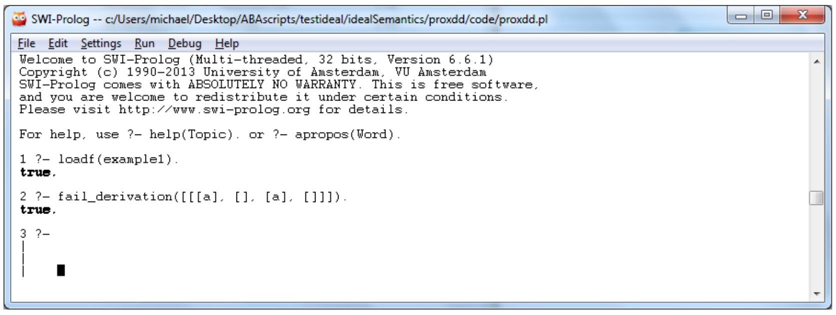
{({a},{},{a},{})}

This verification run was used to test proponent case i which can be seen in algorithm [TODO]. By running the step by hand we realise that the expected output is

{({},{{z}},{a},{})}

Our verification run verifies this by returning a set of tuples in T1 that corresponds to the expected output. Similar tests were carried out with the required input tuples on all the other cases as well. It should be noted that the test used in this example corresponds to the transition from set $\mathbf{D}_0$ to set $\mathbf{D}_1$ in example [TODO] below.

Similarly before we integrate the fail derivation in the system we ensured that as a whole it worked correctly by trying out examples we could work through the derivation by hand. We then provided the starting tuple to the fail derivation predicate and verified that the derivation was carried out correctly, as in example [TODO], where we run *Fail(S)* derivation successfully with an initial input set corresponding to {({a},{},{a},{})}.


SWI-Prolog -- c:/Users/michael/Desktop/ABAscripts/testideal/idealSemantics/proxdd/code/proxdd.pl
File Edit Settings Run Debug Help

```
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 6.6.1)
Copyright (c) 1990-2013 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- loadf(example1).
true.

2 ?- fail_derivation([[[a], [], [a], []]]).
true.

3 ?-
```

Figure 8.8: A verification run of *Fail(S)* derivation.

Example 13. The example below is based on the frame work in example [TODO]. It displays the fail-dispute derivation of {a} step-by-step as it was expected to be carried out if the algorithm was implemented successfully.

$\mathbf{D}_0$ = {({a},{},{a},{})}
$\mathbf{D}_1$ = {({},{{z}},{a},{})}
$\mathbf{D}_2$ = {({},{{a},{b}},{a},{})}
$\mathbf{D}_3$ = {({},{{},{b}},{a},{})}
$\mathbf{D}_4$ = {}

Fail(S) derivations by definition either succeed or fail, which makes it hard to understand whether the derivation was carried out correctly or not. For our example [TODO] we can see that the *Fail(S)* derivation was completed successfully as expected since $\mathbf{D}_4 = \{\}$. In order to verify this we run the predicate carrying out the derivation in the Swi-Prolog console while having set the *guitracer* mode on and having set relevant spy points. This allowed us to verify that at each turn of the derivation the expected tuples were being created and that we did not just end up with the desired outcome by luck.

Testing the ideal based derivations as a whole.

Having ensured the validity of the individual parts that allow us to run derivations under ideal semantics and having integrated them in the current derivation engine, we now had to ensure that the ideal based derivations worked as expected from start to finish. In order to verify this we used frameworks that were designed to verify that the derivation responds as it should. These could be considered as test cases that ensured that the ideal based derivations were correct.

This examples were designed with the theory behind ideal semantics in mind and ensured to test various cases for which ideal derivations should or should not exist.

[TODO] - BIG CHUNK. ADD Toni examples and explain the case of each.

Chapter 9

Conclusions

The web application has been developed to an extend that allows it to perform according to the requirements specified at the beginning of the project. It provides a portal for users to experiment with ABA frameworks using an array of derivation engines. However, due to the nature of the project it was built with possibility of further extending it in the future. It is important to look into the potential this system generates, how it can be extended and what are its contributions to the world of ABA and Computing in general.

9.1 What comes next?

Extendibility and flexibility have always been at the heart of this project, throughout both the design and implementation stages. The potential of the current web application lies in the ability to use the functionality as a middleware on top of which further application can be developed. Additionally, further derivation engines can be adapted so as to be included in the back end of the project. This can allow for comparison to be made between engines, but also can enable the inclusion of derivation engines with specific purposes.

9.1.1 Building on top of the Web—Application.

Due to the modular approach followed when implementing the web application and due to the definition of input commands that can be used to interact with the system, extra applications can be built on top using our web application as middle-ware. The system can be considered as in Figure [TODO], which shows how the system allows a black—box approach to the derivation engines. An input can be provided, it will be processed and the relevant output will be given to the user. The specifics have already been elaborated on during the report, however what should be mentioned is the systems ability to accommodate extensions on both the input mechanisms and the output mechanism.

Reconsidering figure [TODO] we can see that extensions can be built on top of the application both on the input and the output end. These extensions can enable alternative inputs or alternative methods of interpreting the output. As long as the extensions are built so as to use the input language provided by the application and understand the format of the output, such extensions can be integrated seamlessly.

As an example consider Fan and Tonis suggestion [TODO] ADD REFERENCE, by which an Abstract Argumentation framework (AA) can be directly mapped to an ABA framework. A possible input extension would allow the users to input an AA framework which would then be mapped to the respective ABA framework. This can then be automatically formed in the input language of our application and be processed for a derivation. This would extend our application to allow derivations for AA framework to be carried out as well. Similarly other extensions can be implemented that could provide additional functionality, especially when it comes to building applications around real—life problems.

9.1.2 Real World Application.

The strength of ABA lies in its ability to not only provide a true or false argument, but also provide a valid derivation of why that was the outcome. This can be exceptionally useful in decision making problems or providing advisory tools. Applications of argumentation in general can be found in the area of medicine and law, where they provide tools that assist in establishing the validity of claim. For example, provided the correct ABA framework an application can be used to derive whether a diagnosis provided by a doctor is possible according to the framework.

Applications can be built on top of the existing one that are designed specifically for such a real—world scenario. Essentially our web application can be used as a back—end for a more specific application. For ABA to be truly appreciated by the public such tangible, real—life implementations will have to be built. Tools can be built on top of the existing web application in such a way that allows for a useful service to be provided, while abstracting the user from the specialised knowledge of ABA frameworks.

As a suggestion it is worth mentioning the possibility that an advice—providing tool that could possibly work based on ABA could be a tool that provides financial feedback to Small and Medium Enterprises (SMEs) or to people's personal finance. Rules and assumptions can be derived from financial statements that could provide a better understanding of what the users could do to improve their financial situation. This is closely tied to the study of Managerial Accounting were certain calculations and variances act as indicators of a company's strengths and weaknesses. Such an application could take in the relevant financial statements and then return a report concerning suggestions for improvement based on several derivations handled in the back—end.

9.2 Concluding remarks.

In conclusion the web application can be considered as more than just a derivation tool. In fact it can be considered as a platform on which other tools can be implemented with further development. It can also be used as a portal for accessing and comparing various derivation engines. We provided a unified input mechanism that is abstracted by the specifics of the derivation engines, allowing the users to ignore the specifics of what goes on behind. We also extended the existing derivation engines by providing a grounder for the frameworks specified. By handling this as a pre—processing stage we can ensure that the derivation engine will always receive a grounded framework, which it is able to handle it. Additionally we further extended the derivation engines by implementing ideal based semantics derivations, thus extending the engines functionality further. All of this has been wrapped into an easy to use web application that users can access and experiment with ABA framework.

In its current implementation the system would best serve as an educational tool for introducing the users to ABA frameworks. The simple input required and the clean visualisation provided enables users to easily understand the derivation of the target they specified. The ability to do so for 3 different semantics allows them to learn about the differences between the different semantics. However having built the system to provide a middle—ware solution for other applications and engines the project has great potential for further development. Additional engines can be added and further applications can be built both on top of the input and the output of our solution. This way, eventually, the web application can grow into a one–stop—shop for all things ABA.

Bibliography

[1] Dr. Robert Craven. Grapharg. *http://www.doc.ic.ac.uk/∼rac101/proarg/grapharg.html.*

[2] Dr. Robert Craven. Proxdd. *http://www.doc.ic.ac.uk/∼rac101/proarg/proxdd.html.*

[3] Robert Craven, Francesca Toni, and Matthew Williams. Graph-based disput derivations in assumption-based argumentation. In *TAFA 2013.*

[4] Phan Min Dung, Robert Kowalski, and Francesca Toni. Assumption-based argumentation. In *Argumentation in AI.*

[5] P.M. Dung, P. Mancarella, and F. Toni. Computing ideal sceptical argumentation. *Artificial Intelligence*, 171(1015):657 – 660, 2007. ¡ce:title¿Argumentation in Artificial Intelligence¡/ce:title¿.

[6] Uwe Egly, Sarah Alice Gaggl, and Stefan Woltran. Aspartix: Implementing argumentation frameworks using answer-set programming. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, volume 5366 of *Lecture Notes in Computer Science*, pages 734–738. Springer, 2008.

[7] Francesca Toni. Reasoning on the web with assumption-based argumentation. In *8th Reasoning Web Summer School volume 7487 of Lecture Notes in Computer Science.*

[8] Francesca Toni. A generalised framework for dispute derivations in assumption-based argumentation. *Artificial Intelligence*, 195(0):1 – 43, 2013.

[9] Francesca Toni. A tutorial on assumption-based argumentation. *Argument & Computation*, 2013.

[10] TU Wien. Aspartix. *http://www.dbai.tuwien.ac.at/proj/argumentation/systempage/.*

[11] Fan Xiuyi and Toni Francesca. Decision making with assumption-based argumentation. In *TAFA 2013*.