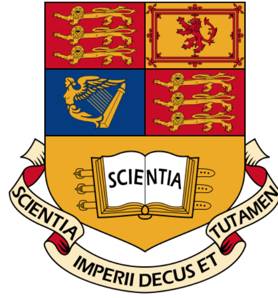

Imperial College London

Department of Computing

A Toolkit for Exploring Argumentation Logic

Author:
Giorgos Flourentzos

Supervisors:
Dr. Krysia Broda
Dr. Francesca Toni
Second Marker:
Claudia Schulz

June 16, 2014

Abstract

Argumentation Logic is a new concept that tries to bridge the gap between Natural Deduction in Propositional Logic and Argumentation Theory by establishing a correspondence between the two. Ultimately, it aims to provide grounds for reasoning in paraconsistent environments.
In order to help explore this concept, a collection of procedures and a graphical user interface have been implemented that allow the creation of natural deduction proofs and their visualization as arguments and vice-versa. This is accomplished through the use of a mapping that has been devised that describes the connection between natural deduction proofs and arguments.

Acknowledgments

First and foremost I would like to wholeheartedly thank my supervisors Dr. Krysia Broda and Dr. Francesca Toni for their continuous and invaluable advice, feedback and support throughout the course of this project, as well as for all the intriguing conversations during our meetings.

Many thanks go to Claudia Schulz, my second marker, whose feedback and advise greatly improved the quality of my report.

I would personally like to thank Professor Antonis C. Kakas for his valuable feedback and discussions regarding Argumentation Logic, without which I would not have been able to accomplish as much as I did.

Special thanks go to Lauren Bennett, who greatly helped me achieve and maintain a healthy balance between my personal and work life and to keep up the fight no matter what, teaching me many important life lessons in the process.

Finally I would like to thank my family and friends for their undying love and support, as well as MeatLiquor and Patty & Bun for the great nights out with friends!

Contents

Contents 5

1 Introduction 9
1.1 Contributions 14
1.2 Structure of Remainder of Report 14

2 Background 15
2.1 Argumentation Theory 15
2.1.1 What is Argumentation Theory 15
2.1.2 Attacking Arguments 15
2.1.3 Types of Arguments 16
2.1.4 Argumentation Example 16
2.1.5 Relevance 16
2.1.6 Abstract Argumentation Framework 17
2.1.7 Visualization of Abstract Argumentation Framework 17
2.2 Natural Deduction 17
2.2.1 Rules for Propositional Logic 18
2.2.2 Example of Natural Deduction Proof 18
2.2.3 Automated Theorem Proving and Proof Search 18
2.2.4 Example of a Proof Search Implementation 19
2.3 Argumentation Logic 20
2.3.1 Introduction 20
2.3.2 Argumentation Logic Framework 20
2.3.3 Acceptability Semantics 23
2.3.4 Reductio ad Absurdum, Genuine Absurdity Property and Acceptability Semantics 24
2.3.5 Disjunction and Implication Connectives 28
2.3.6 Paraconsistency 28
2.4 Existing Visual Argumentation Tools 29

3 Solution Overview 31
3.1 Exploring Argumentation Logic 31
3.1.1 Step 1: Basic Natural Deduction Proof System 31
3.1.2 Step 2: Improving the Proof System 31
3.1.3 Step 3: Genuine Absurdity Property 31
3.1.4 Step 4: Argumentation Logic Visualization 32
3.1.5 Step 5: Converting Natural Deduction Proofs to Argumentation Logic Proofs 32
3.1.6 Step 6: Re-Introduction of Disjunction and Implication Connectives 32
3.1.7 Step 7: Paraconsistency 32

3.1.8 Step 1+: Proof Builder . . . 32
3.1.9 Step 3+: Extending the Genuine Absurdity Property . . . 33
3.1.10 Step 4+: Extracting proofs from arguments . . . 33
3.1.11 Step 4++: Argument Builder . . . 33
3.2 Solution Architecture . . . 33
3.2.1 Core . . . 33
3.2.2 Server . . . 34
3.2.3 Client . . . 34
3.3 Justification of Solution Architecture . . . 34
3.3.1 Advantages of Chosen Architecture . . . 35
3.3.2 Disadvantages of Chosen Architecture . . . 35
3.4 Functional Overview . . . 36

4 Theorem Proving System **38**
4.1 Ruleset Used . . . 38
4.2 Propositional Logic Format . . . 40
4.3 High-Level Description of Implementation . . . 40
4.4 Output Format . . . 41
4.5 Remarks . . . 42

5 Checking for Genuine Absurdity Property **44**
5.1 Short Description of Algorithm . . . 44
5.2 Details of Implementation . . . 45
5.2.1 Checking for RAND Proof . . . 45
5.2.2 Checking for Restricted Formulas . . . 46
5.2.3 Ensuring the Lack of Substitution . . . 46
5.2.4 Checking for Genuine Absurdity Property . . . 47
5.3 Example Walkthrough . . . 49
5.4 Remarks and Limitations . . . 52

6 Extending the Genuine Absurdity Property **53**
6.1 Arriving at the Definition . . . 53
6.2 Definition of Extension . . . 57
6.3 Correctness of Extension . . . 57
6.3.1 Case 1: Not Referencing Sibling Derivations . . . 57
6.3.2 Case 2: Referencing Sibling Derivations . . . 57
6.3.3 Effects of More Specific Context of Implicitly Copied Siblings . . . 57
6.3.4 Assumption of Proof Sketch . . . 58
6.4 Details of Implementation . . . 58
6.5 Remarks . . . 59

7 Visualization of Genuine Absurdity Property Proofs **60**
7.1 Assumptions Made by Algorithm . . . 60
7.2 Description of Algorithm . . . 60
7.3 Observations and Remarks . . . 62
7.4 Details of Implementation . . . 63
7.5 Example Walkthough . . . 66
7.6 Visualization Examples . . . 67

8 Extracting Proofs from Arguments 71
- 8.1 Assumptions Made by Algorithm . . . 71
- 8.2 Description of the Algorithm . . . 71
- 8.3 Observations, Remarks and Future Work . . . 73
- 8.4 Details of Implementation . . . 74
- 8.5 Example Walkthrough . . . 77

9 Server Module 80
- 9.1 Implementation Details . . . 80
- 9.2 Remarks . . . 83
- 9.3 Alternatives . . . 84

10 Client Module 85
- 10.1 Features . . . 85
- 10.2 Usage . . . 85
  - 10.2.1 Clipboard . . . 87
  - 10.2.2 Workbench . . . 88
  - 10.2.3 Options . . . 92
  - 10.2.4 Logic Syntax and Natural Deduction in Client Module . . . 93
- 10.3 Alternatives . . . 94

11 Proof Builder 96
- 11.1 Motivation for Client Side Implementation . . . 96
- 11.2 Features . . . 96
- 11.3 Usage . . . 97

12 Argument Builder 100
- 12.1 Features . . . 100
- 12.2 Usage . . . 101
- 12.3 Generating Arguments Automatically . . . 104

13 Evaluation 105
- 13.1 Theorem Prover . . . 105
  - 13.1.1 Correctness . . . 105
  - 13.1.2 Performance . . . 106
  - 13.1.3 Pruning . . . 112
  - 13.1.4 Optimization . . . 113
  - 13.1.5 Change of Focus . . . 113
  - 13.1.6 Summary . . . 113
- 13.2 Other Algorithms and Procedures . . . 114
- 13.3 Server . . . 115
- 13.4 Client and User Interface . . . 116
- 13.5 Overall Software Engineering Evaluation . . . 116
- 13.6 Project Aims, Objectives and Contributions . . . 116

14 Conclusions and Future Work 118

List of Figures 120

Listings 123

Bibliography **124**

Chapter 1

Introduction

Formal logic is a standard method of giving validity to arguments. However, formal logic trivializes in the presence of inconsistencies, thus becoming inflexible when reasoning about inconsistent theories. Additionally, the Reductio ad Absurdum rule is used freely; under this rule the derived inconsistency can be reached without necessarily using the hypothesis assumed at the beginning of the rule's application. In argumentation, this translates to reaching a conclusion with an argument that has nothing to do with the topic of the conversation.

Argumentation Logic is a new argumentation framework proposed by Professor Antonis Kakas (University of Cyprus, Cyprus), Professor Paolo Mancarella (Universitá di Pisa, Italy) and Dr Francesca Toni (Imperial College London, United Kingdom). It is a framework based on propositional logic that allows reasoning closer to the way humans do. Argumentation Logic does not trivialize in the presence of inconsistencies.

Argumentation Logic is largely based on the already established natural deduction system. The major difference however lies in the fact that it restricts the use of the Reductio ad Absurdum (known as "not introduction") rule in a way that the assumed hypothesis must be critical to the application of that rule. Argumentation Logic can be seen from an argumentative point of view, as a debate between a proponent who puts forth arguments and defends them against an opponent, who in turn tried to attack those arguments. The arguments are sets of propositional sentences. The argument of the proponent is successful and a conclusion is therefore drawn if it can be successfully defended against the attacks made by the opponent. Argumentation Logic is a recent addition to the field of logic and remains partly unexplored.

The idea is to implement a simple and flexible GUI that allows for the construction of valid propositional Argumentation Logic natural deduction proofs. In addition, the implemented software should be able to visualize the proofs as exchanges of arguments between proponent and opponent. This is an attempt to enable further research and study of Argumentation Logic as an established method for reasoning about potentially inconsistent environments in a human-like way, with potential applications in artificial intelligence.

The project was originally split into seven steps which build on top of each other.

- Step 1: Basic Natural Deduction Proof System (chapter 4)

  The first step requires a natural deduction proof system that can find the steps required to reach a goal, given the theory and the goal that must be met. The aim of this step is to provide the basis on which to build the Argumentation Logic framework.

Consider this example: Given theory $T = \{\neg(a \wedge \neg b \wedge \neg c), \neg(a \wedge b), \neg(a \wedge c \wedge \neg d), \neg(d \wedge \neg b)\}$ and goal $\neg a$, the theorem prover could give the proof shown in Figure 1.1.

```
0   !((a&!b)&!c)   given
1   !(a&b)   given
2   !((a&c)&!d)   given
3   !(d&!b)   given
  4   a   hypothesis
    5   b   hypothesis
    6   a&b   &I(4,5)
    7   falsity   _I(1,6)
  8   !b   !I(5,7)
    9   c   hypothesis
      10   d   hypothesis
      11   d&!b   &I(10,8)
      12   falsity   _I(3,11)
    13   !d   !I(10,12)
    14   a&c   &I(4,9)
    15   (a&c)&!d   &I(14,13)
    16   falsity   _I(2,15)
  17   !c   !I(9,16)
  18   a&!b   &I(4,8)
  19   (a&!b)&!c   &I(18,17)
  20   falsity   _I(0,19)
21   !a   !I(4,20)
```

Figure 1.1: One of the generated proofs of the theorem prover with theory $T = \{\neg(a \wedge \neg b \wedge \neg c), \neg(a \wedge b), \neg(a \wedge c \wedge \neg d), \neg(d \wedge \neg b)\}$ and goal $\neg a$. "!" represents negation and "&" represents conjunction

- Step 2: Improving the Proof System

  The aim of this stage is to customize the natural deduction proof system in order to facilitate its usage in the context of Argumentation Logic and its exploration.

- Step 3: Genuine Absurdity Property (chapter 5)

  The third step involves the processing of produced natural deduction proofs in order to check the presence of the Genuine Absurdity Property. This property is closely tied to the identification of natural deduction proofs that are compatible with (that is, supported by) Argumentation Logic. Compatible proofs can be visualized as arguments between two debaters as in the following step.

  For example, the proof given in the example for step 1 does indeed follow this property for reasons that will be explained in chapter 5 and the GUI will award the proof a ribbon to indicate success after the check as shown in Figure 1.2.

- Step 4: Argumentation Logic Visualization (chapter 7)

  The fourth step requires the construction of a GUI (and a collection of background procedures) that allows the visualization of Argumentation Logic proofs as sets of arguments.

  Given the example in step 3, the proof can now be visualized as a futile attempt of the proponent to propose and defend the opposite of the outermost conclusion $\neg a$, a, successfully attacked by the opponent as shown in Figure 1.3.

  A mapping between natural deduction proofs and argumentation theory is discussed in chapter 7.

- Step 5: Converting Natural Deduction Proofs to Argumentation Logic Proofs

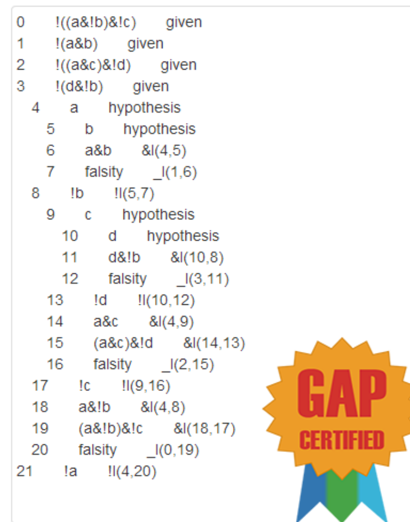

Figure 1.2: The proof in Figure 1.1, now awarded the Genuine Absurdity Property ribbon to indicate that it follows the property. "!" represents negation and "&" represents conjunction

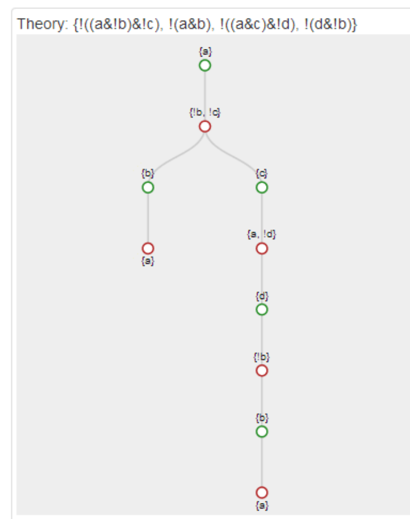

Figure 1.3: The proof in Figure 1.2, now now vizualised as an argument. Green nodes represent defenses by the proponent, and red nodes represent attacks by the opponent.

The fifth step revolves around the conversion of natural deduction proofs that are unsupported by Argumentation Logic (proofs that do not follow the Genuine Absurdity Property) to compatible ones. The aim of this step is to allow the possibility of virtually any natural deduction proof to be visualized from an argumentative view.

- Step 6: Re-Introduction of Disjunction and Implication Connectives

  The sixth step involves the introduction of the disjunction and implication connectives. The use of disjunction and implication remain partly subject for future work. The aim of this step is to explore further this area.

- Step 7: Paraconsistency

  The seventh and final step ventures into how Argumentation Logic can allow for reasoning within an inconsistent environment. The aim of this step is to probe the notion of para-consistency of Argumentation Logic.

The final three steps were unfortunately not explored. In addition to the original steps, several others were planned and implemented during the project. The reason was to provide a better and more integrated environment with a back-and-forth flow between propositional logic and argumentation.

- Step 1+: Proof Builder (chapter 11)

  The creation of a natural deduction proof builder was deemed necessary as users can directly input natural deduction proofs that they may have in mind into the system.

  A sneak peak of the proof builder is shown in Figure 1.4.

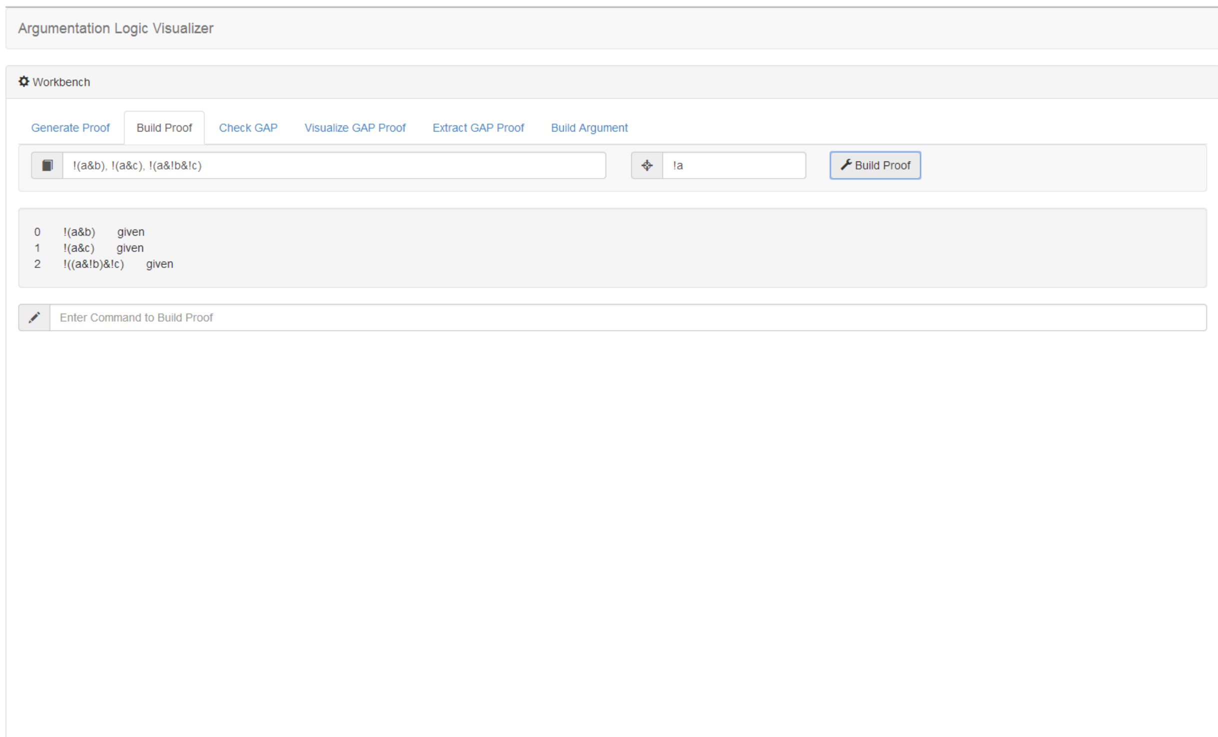

Figure 1.4: The proof builder initialized to build a proof with theory $T = \{\neg(a \wedge b), \neg(a \wedge c), \neg(a \wedge \neg b \wedge \neg c)\}$ and goal $\neg a$.

- Step 3+: Extending the Genuine Absurdity Property (chapter 6)

  This step enables the Genuine Absurdity Property to be extended so that it covers natural deduction proofs that make use of the substitution shortcut.

  For example note the use of the shortcut on line 11 of the proof in Figure 1.2. This uses the substitution shortcut of natural deduction to use the sibling conclusion $\neg b$ of line 8 of the proof, which is the conclusion of the sub-derivation on lines 5 to 7. Normally, this proof is not taken into account even though it seems natural. The extension addresses this specifically.

- Step 4+: Extracting proofs from arguments (chapter 8)

  This step allows a more circular data flow by enabling the transition from arguments back to natural deduction, that complements the original step 4.

  For example, given the argument in Figure 1.3, this algorithm can be used to extract the proof in Figure 1.2.

- Step 4++: Argument Builder (chapter 12)

  Allows for the construction of arguments that follow the semantics of Argumentation Logic directly. This enables a different data entry point as the original steps only allowed proofs to be made and arguments only to be created by converting proofs.

  A sneak peak of the argument builder is shown in Figure 1.5.

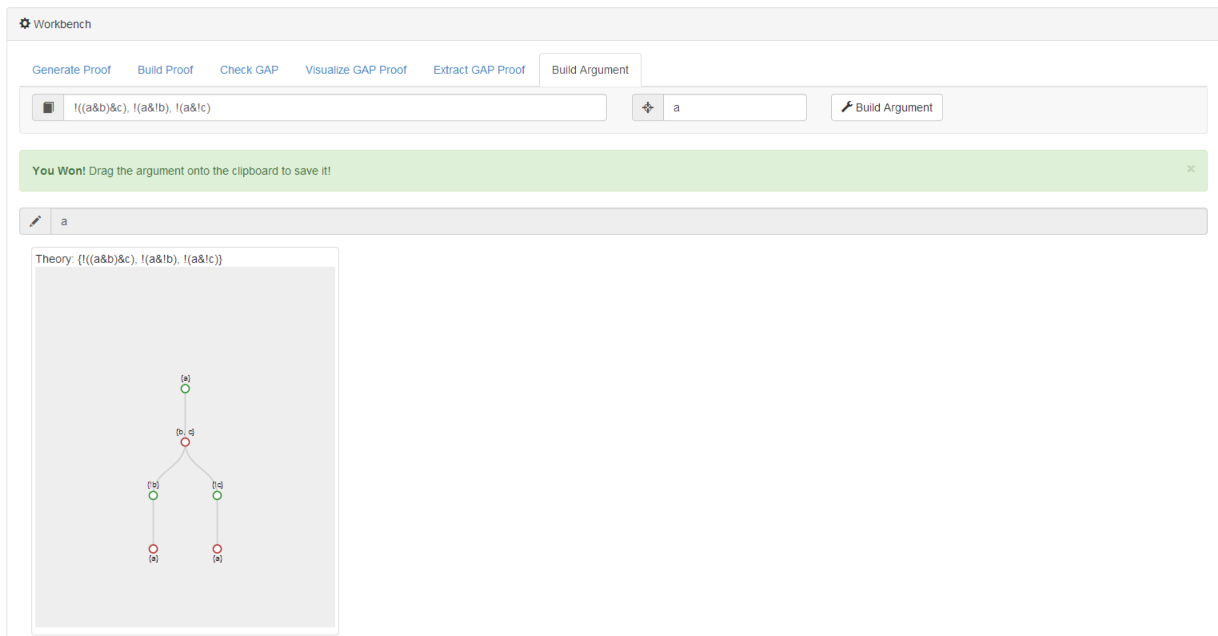

Figure 1.5: The argument builder initialized with a complete argument about a, using theory $\neg(a \wedge b \wedge c), \neg(a \wedge \neg b), \neg(a \wedge \neg c)$.

All of the steps are explained in greater detail in section 3.1.

In conclusion, this project explores the premise of Argumentation Logic, a recent framework based on natural deduction of propositional logic that allows proofs to be visualized as an exchange of arguments. To aid the understanding of Argumentation Logic, a software was created that enables the construction and visualization of proofs that adhere to this logic. At the time of writing, there is no published work in this area, and the nature of this project is investigative.

1.1 Contributions

The project contributes in several ways:

- A theorem prover was built to facilitate the exploration of Argumentation Logic. It can prove a goal given a theory, or fail when it cannot be proven (steps 1 and 2)
- A proof builder was created that facilitates the generation and input of proofs inside the system (step 1+)
- A check was implemented that reflects the exact definition of the Genuine Absurdity Property that can be used to evaluate proofs (step 3)
- A drawback of the Genuine Absurdity Property was identified and remedied by extending the definition to cover more proofs of natural deduction (step 3+)
- An algorithm for drawing proofs that follow the (extended) Genuine Absurdity Property was devised and implemented (step 4)
- An algorithm corresponding to the one above was devised and implemented, with the ability to take an argument and its surrounding theory and extract a proof that will yield the same argument if run through the algorithm in step 4 (step 4+)
- An argument builder feature was created in order to allow for the direct creation of arguments, according to the semantics of Argumentation Logic (step 4++)
- All of the above are wrapped in a user-friendly and intuitive GUI that provides basic usability features such as persistent storage of generated assets, exporting and importing of those assets and visual feedback to the user during constructions or general use.

1.2 Structure of Remainder of Report

The background chapter, chapter 2 covers argumentation theory briefly, as well as natural deduction. It introduces the rules used in natural deduction throughout the paper, as well as its format and style. Argumentation Logic is then introduced, along with explanations of the relevant concepts used in this project.

The project is split into a different modules, as explained by chapter 3, namely, the core module that includes all of the procedures that generate or manipulate proofs and arguments, the server module that imports the core module and serves requests issued by the client module and finally the client module that acts as a wrap-around for the core module, providing a comfortable and useful working environment for the user.

The core implementation as well as the findings of this project are discussed from chapter 4 to chapter 8. An overview of the different processes and features implemented in the solution can be found in section 3.4, along with the data flow between them. The server implementation is discussed in more detail in chapter 9. The client is discussed in chapter 10. The evaluation of this project can be found in chapter 13, and finally, a conclusion is drawn based on the evaluation in chapter 14.

Chapter 2

Background

2.1 Argumentation Theory

Recommended Reading: (Simari and Rahwan, 2009)

2.1.1 What is Argumentation Theory

Exactly what defines an argument varies between different sources in argumentation theory. Douglas Walton defines an argument as being made of three parts: a conclusion, the premises based on which the conclusion is derived and an inference, which links the premises to the conclusion. Arguments are sets of propositions of some format and they tend to attack or defend other arguments in a conversation. They can be used in order to choose a course of action, decide for or against a decision, or find common ground between two (or more) parties. There are several packages that draw arguments as chains of attacks and defenses which will be discussed briefly in section 2.4.

Argumentation differs from the traditional approaches of inference based on deductive logic. The difference lies in that traditional approaches (such as propositional calculus) prove that the conclusion sought after does indeed derive from the given premises. The conclusion and theory are both known in advance, and a single inference is made to link the two. This is called a monological approach. On the contrary, argumentation involves a process that looks more like a dialogue (hence it is a dialogical approach) which tries to look at the pros and cons of an argument. The process involves analyzing the arguments set forth for and against the initial argument, and finding strengths and weaknesses. The final outcome is then based on the strongest argument.

2.1.2 Attacking Arguments

There are different ways to attack an argument. Asking a critical question that raises doubt about a previous argument leads to that argument being refuted unless the other party can respond with a satisfactory answer. Questioning an argument's premises or inference is another way of attack. Putting forward a counter-argument (an argument that reaches the opposite conclusion of the first argument) or arguing that the premises are irrelevant to the conclusion (this problem, introduced by the Reductio ad Absurdum rule concerns Argumentation Logic) are also valid attacks. Different views exist about what constitutes an attack and what not; for example, Krabbe suggests his own seven ways to react to an argument (Krabbe, 2007).

2.1.3 Types of Arguments

Generally, arguments belong to three different categories, based on how the inference that links the conclusion to the evidence was made: deductive, inductive and defeasible. Defeasible arguments are different from inductive in that they cannot be anticipated statistically. For example, "Adults can drive. I am an adult. Therefore, I can drive." is an example of deductive reasoning, but in a defeasible environment it could be the case that I still cannot drive because of a broken leg. Argumentation Logic, as it is based on natural deduction, involves arguments of the deductive kind.

2.1.4 Argumentation Example

An example of an argument between two parties is given below:

1. Student: Higher grades mean higher employability. Decreasing the volume of the curriculum will increase student performance and allow them to get higher marks. Therefore, we should decrease the volume of taught material.
2. Director of Studies: How do you know that decreasing the volume of the material taught will improve students' marks?
3. Student: Students will have more time to digest the curriculum and revise for the exams. In that way, they will achieve higher marks in the exams and overall grades.
4. Director of Studies: Weakening the curriculum will make your degree less desirable at the same time, thus reducing your employability.
5. Student: How do you know that our degree will become less important?
6. Director of Studies: Reports we have gathered from the industry indicate that students from this university are of high demand because of their vast knowledge of material not covered in most other universities.

The student sets the premise by stating that higher grades imply higher employability and smaller curriculum implies higher grades. His conclusion is that the taught material should be decreased. The director of studies attacks the student's argument by challenging the second premise. The student tries to defend himself by providing a concrete argument as to how a smaller curriculum can lead to better grades. The director of studies cannot attack that argument, and thus poses a different (yet relevant) argument: reducing the material offered will make the degree less attractive. The student then attacks this argument by questioning it (the same way the director of studies questioned the student's initial argument). In the end, the director supplies facts that support his argument, leading to a counter-example that suggests that cutting down the curriculum will actually result in lower employability. The debate ends, as the student can no longer support his argument.

2.1.5 Relevance

As briefly mentioned before, irrelevance is one type of fallacy concerning argumentation theory. Under this fallacy, a debater can put forth an argument with premises that are of no relevance to the conversation at hand, perhaps stray away into a different matter and reach a conclusion that otherwise could not be met. Alternatively, this issue could cause the conversation to lead to nowhere. This will be discussed further when explaining how Argumentation Logic restricts the use of the Reductio ad Absurdum rule in order to establish a form of relevance (as seen in section 2.3.4).

2.1.6 Abstract Argumentation Framework

An argumentation framework is a framework that is established in order to allow for formal study of argumentation theory. Perhaps the most basic argumentation framework is the widely known abstract argumentation framework (Dung, 1995). This is a very simple framework that is defined by a set containing the arguments set forth by a party and the attack relation, a binary relation that defines which argument in the set attacks which.

In formal notation, an abstract argumentation framework is usually given by the tuple

$$\langle Args, Att \rangle$$

where:

- Args is the set of all possible arguments that are relevant to the subject modeled by the argumentation framework
- Att is the binary relation $Args \times Args$ where a $(a, b) \in Att$ means that argument a attacks argument b.

For example consider the small abstract argumentation framework that models the possibility of going hiking if it is raining. Assume for this example that going hiking is not possible if it is raining, unless a raincoat is at hand. Additionally, not going hiking on a beautiful sunny day is out of the question, unless of course it happens to be very muddy (from the rain the night before).

Our arguments can be c for having a raincoat, r to indicate that it is raining, h for deciding to go on a hike, h' for deciding against hiking, s for being sunny and finally m for being muddy. Thus $Args = \{c, r, h, h', s, m\}$.

In this framework, argument r attacks argument h (rain does not allow hiking), and argument c attacks argument r (having a raincoat guards against the rain). Also, s attacks h' (as it would be a shame to not go hiking on a sunny day) and m attacks s (since a lot of mud will incur a lot of hand-washing later on). Arguably, h attacks h' and vice-versa, as the two arguments are mutually exclusive. Thus $Att = \{(r, h), (c, r), (s, h'), (m, s), (h, h'), (h', h)\}$.

The abstract argumentation framework resulting from this example will then be of the form $\langle Args, Att \rangle$.

There are very many extensions to Dung's abstract argumentation framework for different reasons, such as aiming to cover limitations of this framework or tailoring it to better fit particular domains. Examples of extensions of this framework are the abstract bipolar argumentation framework (Modril and Caminada, 2008), assumption-based argumentation (Toni, 2013), logic-based argumentation frameworks (Besnard and Hunter, 2001) and value-based argumentation frameworks (Bench-Capon, 2002).

It will be seen later on in section 2.3.2 that Argumentation Logic establishes its own framework by building directly on top of an abstract argumentation framework.

2.1.7 Visualization of Abstract Argumentation Framework

An abstract argumentation framework can be seen from a graphical point of view. The arguments $Args$ form nodes in a graph, and the attack relation Att forms the edges. The above example, with $\langle Args, Att \rangle$ (where $Args = \{c, r, h, h', s, m\}$ and $Att = \{(r, h), (c, r), (s, h'), (m, s), (h, h'), (h', h$ can be visualized in Figure 2.1.

2.2 Natural Deduction

Recommended Reading: (Barker-Plummer, Barwise, and Etchemendy, 2011, pp. 17-225)

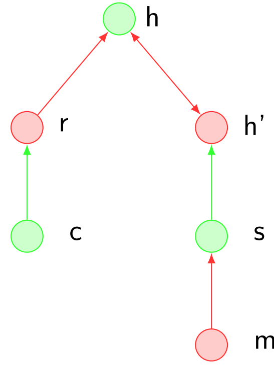

Figure 2.1: Visualization of abstract argumentation example in subsection 2.1.6

2.2.1 Rules for Propositional Logic

The rules for propositional logic used throughout this paper are as follows:

$\wedge I : \frac{\phi,\psi}{\phi\wedge\psi}$	$\wedge E : \frac{\phi\wedge\psi}{\psi}$	$\wedge E : \frac{\phi\wedge\psi}{\phi}$	$\vee I : \frac{\psi}{\phi\wedge\psi}$	$\vee I : \frac{\phi}{\phi\wedge\psi}$	$\vee E : \frac{\phi\vee\psi,[\phi...\chi],[\psi...\chi]}{\chi}$
$\rightarrow I : \frac{[\phi...\psi]}{\phi\rightarrow\psi}$	$\rightarrow E : \frac{\phi,\phi\rightarrow\psi}{\psi}$	$\neg I : \frac{[\phi...\bot]}{\neg\phi}$	$\neg E : \frac{\neg\neg\phi}{\phi}$	$\bot I : \frac{\phi,\neg\phi}{\bot}$	$\bot E : \frac{\bot}{\phi}$

Note: the notation $[\phi...\psi]$ means a derivation of ψ with hypothesis ϕ.

2.2.2 Example of Natural Deduction Proof

An example of a natural deduction proof can be shown below. The format of natural deduction proofs will follow the format of this example:

Assume theory $T = \{\alpha \rightarrow \beta \rightarrow \neg\gamma, \neg\gamma \wedge \beta\}$ and prove $\neg\alpha$.

1	$\alpha \rightarrow \beta \rightarrow \gamma$	given
2	$\neg\gamma \wedge \beta$	given
3	α	hypothesis
4	$\beta \rightarrow \neg\gamma$	$\rightarrow E(1,3)$
5	β	$\wedge E(2)$
6	γ	$\rightarrow E(1,5)$
7	$\neg\gamma$	$\wedge E(2)$
8	$\bot$	$\bot I(6,7)$
9	$\neg\alpha$	$\neg I(3,8)$

A box is used to contain the hypotheses, derivations inside which cannot be used outside. Each derivation is numbered on the left, and reasons (i.e. rules used) for each derivation are given on the right, following the rules defined in the previous section. Theory is indicated as "given", and assumptions (hypotheses) are indicated as "hypothesis" at the beginning of a sub-derivation (inner box). Sub-derivations can also be indicated as $[\phi...\bot]$ in the text, where ϕ is the hypothesis.

2.2.3 Automated Theorem Proving and Proof Search

Despite what the name implies, proof theory in general does not work with natural deduction calculi as the expressiveness of natural deduction comes at a price in terms of its unrestricted search space (Sieg and Scheines, 1992, pp. 140-141). Hilbert systems and natural deduction are said to be inappropriate for automated theorem proving since they both exhibit the modus ponens rule (Fitting, 1995, p. 95). Natural deduction calculi that are indeed used in theorem proving or proof search have special properties and restrictions

in their search space in order to provide a good strategic method of achieving their goals (Sieg and Scheines, 1992, pp. 140-141).

2.2.4 Example of a Proof Search Implementation

The Carnegie Mellon Proof Tutor[1] is an example of an automated proof search program that can formulate proofs in natural deduction by employing a wide range of natural deduction proofs. A summary of its ruleset (with a slight adjustment to the notation) is given below:

Using the notation in Sieg and Scheine's article (Sieg and Scheines, 1992), $\alpha; \beta ? G$ is a triplet describing the following:

- available assumptions as the set of formulas α
- set of formulas obtained by using the $\wedge E$ and $\rightarrow E$ rules on the available assumptions β
- current goal G

The following conventions are used in the article: the concatenation of sequences α and β is indicated by their juxtaposition; if ϕ is a formula and α is a sequence, then the extension of α by ϕ is given by α, ϕ; lastly, a formula ϕ that is a member of the sequence α can be denoted by $\phi \in \alpha$.

The rules used by this proof search program are split into three categories, namely the elimination rules (↓-rules), the introduction rules (↑-rules) and the negation rules.

The elimination rules are listed here:

- $r \downarrow \wedge_i : \alpha, \beta ? G, \phi_1 \wedge \phi_2 \in \alpha\beta, \phi_i \notin \alpha\beta \Rightarrow \alpha; \beta, \phi_i ? G, i = 1\, or\, 2$

  (if there is a conjunction in the set $\alpha\beta$, add its components if not already present)

- $r \downarrow \vee : \alpha, \beta ? G, \phi_1 \vee \phi_2 \in \alpha\beta, \phi_1 \notin \alpha\beta, \phi_2 \notin \alpha\beta \Rightarrow \alpha, \phi_1 ? G\, AND\, \alpha, \phi_2 ? G$

  (if there is a disjunction in the set $\alpha\beta$, and neither of the components have been proven, do a case by case analysis on the goal)

- $r \downarrow\rightarrow \alpha, \beta ? G, \phi_1 \rightarrow \phi_2 \in \alpha\beta, \phi_1 \in \alpha\beta, \phi_2 \notin \alpha\beta \Rightarrow \alpha; \beta, \phi_2 ? G$

  (if there is an implication in the set $\alpha\beta$ and the first component has been proven, add the second component to the set β if not already present)

Note that the restricted versions of the elimination rules are to be used in the proof search implementation (hence the "r" in front of the rules' names) so that redundancy is avoided. The restricted rules only add elements if they are absent from the set $\alpha\beta$. Otherwise, the system could hang by adding infinite copies of the same formulas by repeatedly applying the unrestricted rules. The restrictions in the above rules are the $\phi_i \notin \alpha\beta$ conditions not present in the unrestricted versions.

Consider for example the query $\alpha, \beta ? G$ where $\alpha = \{\gamma \wedge \delta\}, \beta = \{\gamma\}$ and $G = \delta$. If the restriction $\gamma \notin \{\gamma \wedge \delta\}$ was not present, then the first elimination rule would be applicable all the time, and the program could keep adding γ to the set β forever, making no real progress.

The introduction rules are listed here:

- $\uparrow \wedge : \alpha; \beta ? \phi_1 \wedge \phi_2 \Rightarrow \alpha; \beta ? \phi_1\, AND\, \alpha; \beta ? \phi_2$

  (if the goal is a conjunction, prove its constituent parts)

[1]http://www.cs.cmu.edu/ cogito/cptproject.html

- $\uparrow \vee : \alpha; \beta ? \phi_1 \vee \phi_2 \Rightarrow \alpha; \beta ? \phi_1 \, OR \, \alpha; \beta ? \phi_2$

  (if the goal is a disjunction, prove either of its constituent parts)

- $\uparrow\rightarrow: \alpha; \beta ? \phi_1 \rightarrow \phi_2 \Rightarrow \alpha, \phi_1; ?\phi 2$

  (if the goal is an implication, assume the first part and prove the second)

The negation rules are listed here:

- $\bot_c : \alpha; \beta ? \phi, \phi \neq \bot \Rightarrow \alpha, \neg\phi; ?\bot$

  (proof by contradiction: if the goal is a positive formula other than contradiction, prove that its negation leads to a contradiction)

- $\bot_i : \alpha; \beta ? \neg\phi \Rightarrow \alpha, \phi; ?\bot$

  (if the goal is a negated formula, prove that the positive subformula leads to a contradiction)

- $\bot_F : \alpha; \beta ? \bot, \phi \in F \Rightarrow \alpha; \beta ? \phi \, AND \, \alpha; \beta ? \neg\phi$

  (if the goal is a contradiction, prove it by taking any subformula in the set $\alpha\beta$ and prove both that subformula and its negation)

The last rule mentions F; this is the finite class of formulas that consist of all subformulas of elements in the subsequence $\alpha\beta$. The theorem prover built for this project (chapter 4) will base some of its rules on the Carnegie Mellon Proof Tutor.

2.3 Argumentation Logic

Recommended Reading: (Kakas, Toni, and Mancarella, 2012)

2.3.1 Introduction

This section gives a brief introduction of the concepts behind Argumentation Logic, as found in the technical report. Section Exploring Argumentation Logic (section 3.1) shows how these concepts are used to build the visualization tool.

Argumentation Logic builds a bridge between argumentation theory and propositional logic. This duality is formed by combining notions from both argumentation theory and natural deduction. For consistent theories, Argumentation Logic is equivalent to propositional logic, but it also extends into a para-consistent logic for inconsistent theories. From the argumentation point of view, Argumentation Logic can be seen as arguments that are sets of propositional formulas that attack and defend against other arguments. From the propositional logic point of view, Argumentation Logic can be seen as a natural deduction system that restricts the use of the Reductio ad Absurdum rule in order to allow for relevant arguments to be used only. The rest of this section is devoted to explaining the concepts behind this new logic.

2.3.2 Argumentation Logic Framework

In order to establish the Argumentation Logic framework, the notions of "direct derivation" and "direct consistency" must first be defined:

Direct Derivation

A direct derivation for a sentence from a theory is a natural deduction derivation of that sentence from the given theory that does not contain any application of the Reductio ad Absurdum rule. If such a derivation exists, then we say that this sentence is directly derived (derived modulo RA) from the theory. For a sentence ϕ directly derived from theory T, we denote $T \vdash_{MRA} \phi$. For example, assume theory $T = \{\alpha \rightarrow \beta, \beta \rightarrow \delta\}$, and derive $\alpha \rightarrow \delta$:

1	$\alpha \rightarrow \beta$	given
2	$\beta \rightarrow \delta$	given
3	α	hypothesis
4	β	$\rightarrow E(1,3)$
5	δ	$\rightarrow E(2,4)$
9	$\alpha \rightarrow \delta$	$\neg I(3,5)$

This is a direct derivation as the Reductio ad Absurdum rule was not used.

As another example, assume theory $T = \{\alpha \rightarrow \bot\}$ and derive $\neg\alpha$:

1	$\alpha \rightarrow \bot$	given
3	α	hypothesis
4	$\bot$	$\rightarrow E(1,2)$
9	$\neg\alpha$	$\neg I(2,3)$

This is not a direct derivation as the Reductio ad Absurdum rule had to be used.

Classical and Direct Consistency/Inconsistency

The word "classical" is used to denote the original natural deduction entailment. The word "direct" uses the notion above. A theory is classically inconsistent if a contradiction can be derived from it in the "classical" sense. A theory is directly inconsistent if a contradiction can be derived through a direct derivation. A theory is classically or directly consistent if it is not classically or directly inconsistent, respectively.

In notation, a theory T is

- classically inconsistent if $T \vdash \bot$
- directly inconsistent if $T \vdash_{MRA} \bot$
- classically consistent if $T \nvdash \bot$
- directly consistent if $T \nvdash_{MRA} \bot$

In a sense, direct derivation capabilities form a subset of those of the classical derivation. Hence, if a theory is classically consistent then it is directly consistent too. A directly consistent theory can be classically inconsistent however, since classical derivation has one more rule for proving contradiction (namely, the Reductio ad Absurdum rule) that direct derivation does not have.

As an example, consider theory $T = \{\alpha \rightarrow \beta, \neg\alpha \rightarrow \gamma, \neg\beta \wedge \neg\gamma\}$ and prove contradiction:

1	$\alpha \to \beta$	given
2	$\neg\alpha \to \gamma$	given
3	$\neg\beta \wedge \neg\gamma$	given
4	α	hypothesis
5	β	$\to E(1,4)$
6	$\neg\beta$	$\wedge E(3)$
7	$\perp$	$\perp I(5,6)$
8	$\neg\alpha$	$\neg I(4,7)$
9	γ	$\to E(2,8)$
10	$\neg\gamma$	$\wedge E(3)$
11	$\perp$	$\perp I(9,10)$

This proof requires the use of the Reductio ad Absurdum rule, without which a contradiction cannot be derived. Thus, this theory is classically inconsistent, but directly consistent.

Argumentation Logic Framework Definition

The Argumentation Logic framework relies on Dung's abstract argumentation framework as defined in subsection 2.1.6. It involves a set of arguments (where each argument is a set of propositional sentences) and the attack relation between the arguments. Thus for a given theory T, the Argumentation Logic framework becomes

$$\langle Args^T, Att^T \rangle$$

where:

- $Args^T = \{T \cup \Sigma\}$ where Σ is a set of propositional formulas. Hence all arguments include the starting theory T and potentially more propositional formulas Σ
- $Att^T = \{(b,a) | a, b \in Args^T, a = T \cup \Delta, \Delta \neq \{\}, b = T \cup \Gamma, T \cup \Delta \cup \Gamma \vdash_{MRA} \perp\}$, that is, a set of pairs of arguments, the union of which provides ground for the direct derivation of a contradiction. In other words, Att^T contains all pairs of arguments that don't agree with each other!

Since the theory is fixed for the argumentation framework, any argument $a = T \cup \Sigma$ will be referred to only by Σ. The argument $T \cup \{\}$ will thus be referred to as the empty argument. Note that in the attack relation, the attacked argument cannot be empty and apart from this exception, all attacks are reflexive. As an example, consider $T = \{\alpha \to \beta, \alpha \to \gamma\}$. Here, $\{a\}$ attacks (and is attacked by) $\{\neg\beta\}$ or $\{\neg\gamma\}$. For a directly inconsistent theory, all arguments are hostile to each other since a contradiction can be derived from any possible pair of arguments (the empty argument can still not be attacked).

Defense Against an Attack

Using the Argumentation Logic framework described above, and taking any argument $a = T \cup \Delta$, an argument d can be described as a defense against a if any of the following is true:

- $d = T \cup \{\neg\phi\}$ or $d = T \cup \{\phi\}$ for some sentence $\phi \in \Delta$ or $\neg\phi \in \Delta$ respectively
- $d = T \cup \{\}$ and $a \vdash_{MRA} \perp$

What this means is that argument d can take an opposing view on one of the sentences in argument a (this can be interpreted as questioning one of the premises or the conclusion of an argument in argumentation theory) or if argument a is self-contradicting, then saying nothing (empty argument) still counts as a defense against that argument.

2.3.3 Acceptability Semantics

This section defines what it means for an argument to be acceptable in Argumentation Logic, as discussed in the technical report.

Acceptability of Arguments

Given an argumentation framework $\langle Args^T, Att^T \rangle$ as discussed in the previous section fixed for a consistent theory T, with $a, b \in Args^T$, then a is acceptable with respect to b, denoted by $ACC^T(a, b)$, if and only if either of the following conditions is met:

- $a \subseteq b$
- for all $c \in Args^T$ such that $(c, a) \in Att^T$ both of the following are true:
  - $c \nsubseteq a \cup b$
  - there is an argument $d \in Args^T$ which defends against c and $ACC^T(d, a \cup b)$

Intuitively, an argument is acceptable with respect to one other one if it is a subset of it (they share the same ideas), or all of its attacking arguments are not based on the same ideas (are not subsets of the two arguments whose acceptability is under examination) and they can be successfully blocked by other acceptable arguments.

Non-Acceptability of Arguments

Similarly to the acceptability of arguments, for argumentation framework $\langle Args^T, Att^T \rangle$ fixed for a consistent theory T, with $a, b \in Args^T$, then a is not acceptable with respect to b, denoted by $NACC^T(a, b)$, if and only if both of the following conditions are met:

- $a \nsubseteq b$
- there is an argument $c \in Args^T$ such that $(c, a) \in Att^T$ and either of the following is true:
  - $c \subseteq a \cup b$
  - for all arguments $d \in Args^T$ which defend against c it is true that $NACC^T(d, a \cup b)$

Intuitively, an argument is unacceptable with respect to another if they are different and there is an attacking argument that comes from the same ideas as the arguments under examination and it cannot be defended against (by an acceptable argument). Note that non-acceptability is the exact opposite of acceptability, and so it holds that $NACC^T(a, b) = \neg ACC^T(a, b)$.

Example of Non-Acceptability

Consider theory $T = \{\alpha \wedge \beta \to \bot, \neg\beta \wedge \gamma \to \bot, \neg\gamma \wedge \delta \to \bot\}$. $NACC^T(\{a\}, \{\})$ holds, because:

- $\{a\} \nsubseteq \{\}$, $\{\beta\}$ attacks $\{\alpha\}$ and $\{\neg\beta\}$ is the only defense against $\{\beta\}$, and so it suffices to show that $NACC^T(\{\neg\beta\}, \{\alpha\})$
- $\{\neg\beta\} \nsubseteq \{\alpha\}$, $\{\gamma\}$ attacks $\{\neg\beta\}$ and $\{\neg\gamma\}$ is the only defense against $\{\gamma\}$, and so it suffices to show that $NACC^T(\{\neg\gamma\}, \{\alpha \neg\beta\})$
- $\{\neg\gamma\} \nsubseteq \{\alpha, \neg\beta\}$, $\{\delta\}$ attacks $\{\neg\gamma\}$ and there is unfortunately no defense against it, thus $NACC^T(\{\neg\gamma\}, \{\alpha, \neg\beta\})$, $NACC^T(\{\neg\beta\}, \{\alpha\})$ and in turn $NACC^T(\{\alpha\}, \{\})$ all hold

2.3.4 Reductio ad Absurdum, Genuine Absurdity Property and Acceptability Semantics

This section introduces the Genuine Absurdity Property, and relates it to the use of the Reductio ad Absurdum rule and the acceptability semantics.

RAND Derivations

Reductio ad Absurdum derivations (RAND for short) are natural deduction derivations that are enclosed by a Reductio ad Absurdum rule application.

Thus a RAND derivation of a propositional formula $\neg\phi$ is a natural deduction derivation of $\neg\phi$ which starts with a hypothesis ϕ and reaches a contradiction, allowing for the Reductio ad Absurdum rule to be applied in order to deduce $\neg\phi$.

A sub-derivation (of a certain derivation) is a RAND sub-derivation (of that derivation) if the sub-derivation itself is a RAND derivation. Hence a tree can be formed with the RAND derivation as the root, its RAND sub-derivations as its immediate children, the RAND sub-sub-derivations as the next node level, and so on.

Consider as an example, theory $T = \{\alpha \to \bot, \beta \to \bot, \neg\alpha \wedge \neg\beta \wedge \gamma \to \bot\}$ and prove $\neg\gamma$.

1	$\alpha \to \bot$	given
2	$\beta \to \bot$	given
3	$\neg\alpha \wedge \neg\beta \wedge \gamma \to \bot$	given
4	γ	hypothesis
5	α	hypothesis
6	$\bot$	$\to E(1,5)$
7	$\neg\alpha$	$\neg I(5,6)$
8	β	hypothesis
9	$\bot$	$\to E(2,8)$
10	$\neg\beta$	$\neg I(8,9)$
11	$\neg\alpha \wedge \neg\beta$	$\wedge I(7,10)$
12	$\neg\alpha \wedge \neg\beta \wedge \gamma$	$\wedge I(4,11)$
13	$\bot$	$\bot I(4,12)$
14	$\neg\gamma$	$\neg I(4,13)$

This proof can be visualized as a tree with the root being the outer derivation of $\neg\gamma$ and with children the (sub-)derivations of $\neg\alpha$ and $\neg\beta$, as shown in Figure 2.2. The exact algorithm for visualizing a proof will be given in chapter 7.

For a directly consistent theory, if $NACC^T(\{\phi\}, \{\})$ holds for some ϕ, then there is a RAND derivation of $\neg\phi$ from that theory (Kakas, Toni, and Mancarella, 2012, p. 18).

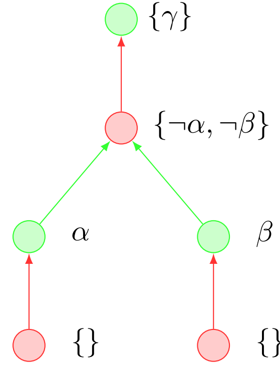

Figure 2.2: Visualization of the proof of example in section 2.3.4

Genuine Absurdity Property

For the rest of this section, theories are assumed to be expressed using conjunction and negation only. This gives rise to the following property: all sub-derivations of any derivation of a theory are RAND sub-derivations. The following notation (adapted from the technical report) can be used to represent RAND derivations:

$$[\phi : c(\phi_1), ..., c(\phi_k); \neg\psi_1, ..., \neg\psi_l : \bot]$$

where $k, l \geq 0$ and

- ϕ is the hypothesis of this derivation
- ϕ_i are the hypothesis of parent derivations that this derivation has access to and can make use of
- ψ_i are the hypotheses of the children derivations, the negations of which can be used by this derivation

A very important thing to note is that such derivation contains copies of ancestor hypotheses that were explicitly copied into the sub-derivation. It does not necessarily hold all of its ancestor hypotheses. This is indicated by $c(\phi_i)$ instead of just ϕ_i in the notation above. In the Argumentation Logic paper, there is an explicit copy rule that copies a hypothesis from an ancestor in order to be used inside the sub-derivation. Copying in the natural deduction style used in this paper will be explained later.

This notation can also nest derivations inside one another. From the last example with theory $T = \{\alpha \to \bot, \beta \to \bot, \neg\alpha \wedge \neg\beta \wedge \gamma \to \bot\}$ and proof of $\neg\gamma$, the RAND derivations that took place can be described as follows, using this notation:

$$[\gamma : -; [\alpha : c(\gamma); - : \bot], [\beta : c(\gamma); - : \bot] : \bot]$$

Note that the outer derivation has no inherited ancestral hypotheses, and the inner derivations (which correspond to the leaves in the tree) have no child hypotheses; therefore "-" is used to represent empty sequences in the notation. Note the copies of the hypothesis γ of the ancestor derivation $[\gamma ... \bot]$ in the sub-derivations $[\alpha ... \bot]$ and $[\beta ... \bot]$.

As a consequence of a RAND derivation, $T \cup \{\phi\} \cup \{\phi_1, ..., \phi_k\} \cup \{\neg\psi_1, ..., \neg\psi_l\} \vdash_{MRA} \bot$. That is to say, the theory along with the hypothesis of that derivation and the assistance of parent and child hypotheses can be used to derive a contradiction (which gives ground for the deduction of $\neg\phi$ using the Reductio ad Absurdum rule).

The Genuine Absurdity Property then states that a RAND derivation satisfies this property if the hypothesis of the derivation must be used in order to derive a contradiction. In other words, the hypothesis must be relevant and without it a contradiction cannot be established in any way. In formal notation this means that $T \cup \{\phi_1, ..., \phi_k\} \cup \{\neg\psi_1, ..., \neg\psi_l\} \nvdash_{MRA} \bot$ (note the absence of ϕ). In addition, all RAND sub-derivations must also follow this property, making it a recursive property. Note that the set $\{\phi_1, ..., \phi_k\}$ contains only the ancestor hypotheses copied into the sub-derivation, not necessarily all of them.

In terms of argumentation, the violation of the genuine absurdity property can be thought of an argument with premises irrelevant to the conversation at hand (as was discussed in subsection 2.1.5). RAND derivations satisfying the genuine absurdity property are not guaranteed to exist, except for classically consistent theories (Kakas, Toni, and Mancarella, 2012, p. 9). It can be shown that for a directly consistent theory, if there is a RAND derivation of $\neg\phi$ that fully satisfies the genuine absurdity property then $NACC^T(\{\phi\}, \{\})$ holds (Kakas, Toni, and Mancarella, 2012, p. 19).

Genuine Absurdity Property and Substitution Shortcut

A very important thing to note is that the Genuine Absurdity Property is defined only in natural deduction that does not make use of the substitution shortcut (which can be thought of as similar to a cut in sequent calculus). This shortcut is useful in avoiding to re-prove parts of the derivation by proving them once in a more general context and then using them repeatedly in (possibly) more specific contexts. To illustrate the use of the shortcut, consider the two proofs shown in Figure 2.3.

1	$\neg(\alpha \wedge \beta)$	given
2	$\neg(\alpha \wedge \gamma \wedge \neg\beta)$	given
3	$\neg(\alpha \wedge \neg\gamma \wedge \neg\beta)$	given
4	α	hypothesis
5	β	hypothesis
6	$\alpha \wedge \beta$	$\wedge I(4,5)$
7	$\bot$	$\bot I(1,6)$
8	$\neg\beta$	$\neg I(5,7)$
9	γ	hypothesis
10	$\alpha \wedge \gamma$	$\wedge I(4,9)$
11	$\alpha \wedge \gamma \wedge \neg\beta$	$\wedge I(10,8)$
12	$\bot$	$\bot I(2,11)$
13	$\neg\gamma$	$\neg I(9,12)$
14	$\alpha \wedge \neg\gamma$	$\wedge I(4,13)$
15	$\alpha \wedge \neg\gamma \wedge \neg\beta$	$\wedge I(14,8)$
16	$\bot$	$\bot I(3,15)$
17	$\neg\alpha$	$\neg I(4,16)$

1	$\neg(\alpha \wedge \beta)$	given
2	$\neg(\alpha \wedge \gamma \wedge \neg\beta)$	given
3	$\neg(\alpha \wedge \neg\gamma \wedge \neg\beta)$	given
4	α	hypothesis
5	β	hypothesis
6	$\alpha \wedge \beta$	$\wedge I(4,5)$
7	$\bot$	$\bot I(1,6)$
8	$\neg\beta$	$\neg I(5,7)$
9	γ	hypothesis
10	$\alpha \wedge \gamma$	$\wedge I(4,9)$
11	β	hypothesis
12	$\alpha \wedge \beta$	$\wedge I(4,11)$
13	$\bot$	$\bot I(1,12)$
14	$\neg\beta$	$\neg I(11,13)$
15	$\alpha \wedge \gamma \wedge \neg\beta$	$\wedge I(10,14)$
16	$\bot$	$\bot I(2,15)$
17	$\neg\gamma$	$\neg I(9,16)$
18	$\alpha \wedge \neg\gamma$	$\wedge I(4,17)$
19	$\alpha \wedge \neg\gamma \wedge \neg\beta$	$\wedge I(18,8)$
20	$\bot$	$\bot I(3,19)$
21	$\neg\alpha$	$\neg I(4,20)$

Figure 2.3: Proofs showing the use of a shortcut in natural deduction. The proof on the right derives $\neg\beta$ again whereas the proof on the left simply reuses the previous derivation of $\neg\beta$

These two proofs are the same in a sense. The difference lies in that the first proof does not derive $\neg\beta$ again by re-using the derivation made on lines 5-8. The second proof is more explicit, by re-proving $\neg\beta$ inside the derivation of $\neg\gamma$. To see how this affects the definition of the Genuine Absurdity Property, take the child hypotheses for the derivation of $\neg\gamma$ for each proof. In the former proof, there is none, whereas in the latter, there is β. The definition of the Genuine Absurdity Property depends on the set of children hypotheses of each (sub)derivation, and allowing the use of the shortcut affects that set. This is because in the first proof the check for the property will be $T \cup \{\alpha\} \cup \{\} \nvdash_{MRA} \bot$, whereas in the second it will be $T \cup \{\alpha\} \cup \{\neg\beta\} \nvdash_{MRA} \bot$. The fact that $\neg\beta$ is excluded from the check could affect whether the proof is correctly attributed as following the Genuine Absurdity Property or not. This is addressed in chapter 6 where an extension is given to this definition so that proofs that use the shortcut can still follow this property.

Copying of Ancestor Hypotheses

In the Argumentation Logic paper, sub-derivations have access to ancestor hypotheses by copying them explicitly in the derivation, indicated by a line $c(\phi_i)$ where ϕ_i is the ancestor derivation being copied. This paper does not have an explicit "copy" rule, therefore the copying of ancestor hypotheses will happen implicitly. If a sub-derivation features a step that has a line reference in its reason (justification) pointing to a hypothesis, then that hypothesis can be considered as having been implicitly copied into the sub-derivation. An example is shown in Figure 2.4.

Line	Formula	Justification
1	$\neg(\alpha \wedge \beta)$	given
2	$\neg(\alpha \wedge \neg\beta)$	given
3	$\neg\beta$	given
4	α	hypothesis
5	β	hypothesis
6	$\alpha \wedge \beta$	$\wedge I(4,5)$
7	$\bot$	$\bot I(1,6)$
8	$\neg\beta$	$\neg I(5,7)$
9	$\alpha \wedge \neg\beta$	$\wedge I(4,8)$
10	$\bot$	$\bot I(2,9)$
11	$\neg\alpha$	$\neg I(4,10)$

Line	Formula	Justification
1	$\neg(\alpha \wedge \beta)$	given
2	$\neg(\alpha \wedge \neg\beta)$	given
3	$\neg\beta$	given
4	α	hypothesis
5	β	hypothesis
6	$\bot$	$\bot I(3,5)$
7	$\neg\beta$	$\neg I(5,6)$
8	$\alpha \wedge \neg\beta$	$\wedge I(4,7)$
9	$\bot$	$\bot I(2,8)$
10	$\neg\alpha$	$\neg I(4,9)$

Line	Formula	Justification
1	$\neg(\alpha \wedge \beta)$	given
2	$\neg(\alpha \wedge \neg\beta)$	given
3	$\neg\beta$	given
4	α	hypothesis
5	β	hypothesis
6	$\alpha \wedge \beta$	$\wedge I(4,5)$
7	$\bot$	$\bot I(3,5)$
8	$\neg\beta$	$\neg I(5,7)$
9	$\alpha \wedge \neg\beta$	$\wedge I(4,8)$
10	$\bot$	$\bot I(2,9)$
11	$\neg\alpha$	$\neg I(4,10)$

Figure 2.4: Proofs showing the implicit use of copying of ancestor hypotheses: the proof in the middle makes no reference to the ancestor hypothesis α whereas the other two implicitly copy it

Figure 2.4 shows the proof on the left implicitly copying the ancestor hypothesis α because line 6 of the $[\beta ... \bot]$ sub-derivation refers to line 4 in its justification ($\wedge I(4,5)$). The proof on the right implicitly copies the ancestor hypothesis too, albeit not using it (note that line 7 refers to lines 3 and 5 of the proof, not line 6). The middle proof makes no mention of the ancestor hypothesis α whatsoever and so it does not implicitly copy it.

Acceptability Semantics With Respect to Propositional Logic

The technical report presents a series of theorems and proofs that closely relate Argumentation Logic's acceptability semantics to propositional logic (Kakas, Toni, and Mancarella,

2012, pp. 10-11). Below is a summary of properties of the notions of acceptability that demonstrate their connection to notions in propositional logic:

- For a classically consistent theory, and a RAND derivation of a negated formula, there exists another RAND derivation of that negated formula that fully satisfies the genuine absurdity property.
- For a classically consistent theory T, if $T \vdash \phi$ then both $ACC^T(\{\phi\}, \{\})$ and $NACC^T(\{\neg\phi\}, \{\})$ hold.
- For a classically inconsistent theory T such that $NACC^T(\{\neg\phi\}, \{\})$ holds, $T \vdash \phi$.

The definition of $NACC^T$-entailment is the following: For a classically consistent theory T, ϕ is $NACC^T$-entailed (written as $T \models_{NACC^T} \phi$) by T if and only if $NACC^T(\{\neg\phi\}, \{\})$. Hence the last connection between the acceptability semantics of Argumentation Logic and propositional logic is the following: For a classically consistent theory T, $T \models \phi$ if and only if $T \models_{NACC^T} \phi$. This result is a direct consequence of the properties mentioned above, and in the case of inconsistent theories a natural generalization can be obtained with the addition of an extra condition defined later.

2.3.5 Disjunction and Implication Connectives

Argumentation Logic establishes an equivalence, for classically consistent theories, between itself and propositional logic, under the notion of $NACC^T$-entailment and standard entailment respectively. This equivalence however works with the conjunction and negation connectives. The relations between conjunction and negation and disjunction and implication are given by $\alpha \vee \beta \equiv \neg(\neg\alpha \wedge \neg\beta)$ and $\alpha \rightarrow \beta \equiv \neg(\alpha \wedge \neg\beta)$. In order to include the disjunction and implication connectives it must be shown that $NACC^T$-entailment can be established both ways for both relations. In other words, for disjunction, it must be shown that $\{\alpha \vee \beta\} \models_{NACC^T} \neg(\neg\alpha \wedge \neg\beta)$ and $\{\neg(\neg\alpha \wedge \neg\beta)\} \models_{NACC^T} \alpha \vee \beta$, and for implication, it must be shown that $\{\alpha \rightarrow \beta\} \models_{NACC^T} \neg(\alpha \wedge \neg\beta)$ and $\{\neg(\alpha \wedge \neg\beta)\} \models_{NACC^T} \alpha \rightarrow \beta$.

Fortunately, for the disjunction, both entailments can be shown (Kakas, Toni, and Mancarella, 2012, pp. 11-12). However, in the case of the implication, only $\{\alpha \rightarrow \beta\} \models_{NACC^T} \neg(\alpha \wedge \neg\beta)$ can be shown. In order for $\{\neg(\alpha \wedge \neg\beta)\} \models_{NACC^T} \alpha \rightarrow \beta$ to be possible, the attacking semantics should be changed to account for this case explicitly (Kakas, Toni, and Mancarella, 2012, pp. 12-13). This topic remains a topic for future work.

2.3.6 Paraconsistency

Argumentation Logic is equivalent to propositional logic for consistent theories, under the notion of $NACC^T$-entailment and standard entailment respectively. In the case of classically inconsistent theories, Argumentation Logic define two new notions, each generalizing from the previous (Kakas, Toni, and Mancarella, 2012, pp. 13-15):

Directly Consistent Theories

For a directly consistent theory T, ϕ is AL-entailed by T (written as $T \models_{AL} \phi$ if and only if $ACC^T(\{\phi\}, \{\})$ and $NACC^T(\{\neg\phi\}, \{\})$. This is a generalization of the $NACC^T$-entailment for classically consistent theories, based on the argumentation perspective of an acceptable argument being successfully defended and not successfully objected against.

AL-entailment leads to a form of para-consistency, since it does not trivialize in the case of the application of the Reductio ad Absurdum rule (as is the case with standard

entailment) due to the notion of non-acceptability. However, even with this addition, AL-entailment is still not applicable to directly inconsistent theories.

Directly Inconsistent Theories

The notion of AL-entailment above does not work for directly inconsistent theories, as from Argumentation Logic's point of view, the theory is in layman's terms as wrong (inconsistent) as it can get. In terms of Argumentation Logic, if the theory is directly inconsistent (in addition to being classically inconsistent as well), then no matter what argument is put forth, it can always be attacked by the empty argument since $T \vdash_{MRA} \perp$ to begin with. For this reason, the entire theory cannot be considered as a whole, and hence a new notion of entailment is established, one that makes use of maximally consistent closure sets. For theory T, and $Cn(T) = \{\phi | T \vdash_{MRA} \phi\}$ being the set of all direct consequences of T, ϕ is $AL+$-entailed by T (written as $T \vdash_{AL+} \phi$) if and only if $T' \vdash_{AL} \phi$ for all maximally directly consistent sets $T' \subseteq Cn(T)$. For directly consistent theories $AL+$-entailment is equivalent to AL-entailment.

2.4 Existing Visual Argumentation Tools

There is a plethora of existing argumentation tools, many of which provide visualization of arguments. These tools range in purpose from academic and research or educational to commercial tools used in analyzing legal arguments for analyzing the rationality of arguments presented in a courtroom.

No universal agreement exists on the type of argument maps that should be supported by each tool. Some tools support the Toulmin Model of Argument, which is a model proposed by Stephen Toulmin for analysing arguments in legal matters, later realized to have a wider application than just law (Toulmin, 2003). Another popular representation format is the Wigmore chart, targeting analysis of legal evidence in trials, which is the work of John Henry Wigmore (Anderson, Schum, and Twining, 2005, pp. 123-144). According to Kadane and Schum, a Wigmore chart represents an early version of a Bayesian network (Kadane and Schum, 1996, pp. 66-76). However, since many of the available tools target specific applications (sometimes in domains outside of law), they opt to visualize their arguments in ways more fitting to the applications they are intended for.

A conscious effort is being made to consolidate the representation of arguments into a single standard format that will allow the exchange of arguments between different applications. One of the proposed standards is the Argument Interchange Format, which is currently under construction. A short-term problem with this format is that different application-specific requirements, which result in different flavors of the format being implemented, need to be tracked in order to improve compatibility; at the same time, a long-term problem might be the time at which the standard will be cast in stone: if this happens too early, then it will probably not account for all the requirements that might emerge from different argumentation applications, however, if this happens too late, then the ramifications will be too deviant to be brought together into a standard (Simari and Rahwan, 2009, p. 401). Another format already in use is the Legal Knowledge Interchange Format (LKIF), an XML schema that extends Web Ontology Language in order to represent legal concepts.

A non-exhaustive list of existing argumentation (visualization) tools is given below:

- Araucaria
- Argkit & Dungine

- ArguGRID
- Arguing Agents Competition (AAC)
- ASPARTIX
- Carneades
- Cohere
- Compendium
- InterLoc
- quaestio-it
- Rationale

Since AIF remains volatile at the time of this writing, and LKIF is only concerned with legal matters, neither of these formats will be used. The visualization tool produced as part of this project will use its own format to store data, and adoption of AIF might be revisited as a possible extension.

Chapter 3

Solution Overview

This chapter firstly re-introduces the steps from the introduction in more detail, and explains how each step was implemented. This chapter also aims to introduce the overall structure of the solution and provide justification as to the decisions made that constitute the solution.

3.1 Exploring Argumentation Logic

As briefly described in the introduction to this paper, the project tries to explore Argumentation Logic in seven steps. The last three steps were not implemented, and instead focus was redirected to providing a better overall experience for the first 4 steps.

3.1.1 Step 1: Basic Natural Deduction Proof System

The first stage requires a natural deduction proof system that can find the steps required to reach a goal, given the theory and the goal that must be met. Natural deduction proofs are generated using the proof system built in this step (chapter 4), in order to later on (in step 3) check which proofs constitute valid Argumentation Logic proofs, and which can be converted to valid Argumentation Logic proofs (step 5).

3.1.2 Step 2: Improving the Proof System

The theorem prover was built to be very flexible, with the ability to turn on and off different rules, as required by step 3, checking for the Genuine Absurdity Property.

3.1.3 Step 3: Genuine Absurdity Property

The third step (chapter 5) involves the processing of produced natural deduction proofs (from step 1 and step 1+) in order to check the presence of the Genuine Absurdity Property as discussed in the Argumentation Logic section before. This property is closely tied to the identification of natural deduction proofs that are compatible with (that is, supported by) Argumentation Logic. Compatible proofs can be visualized as arguments between two debaters as in the following step, which is the aim of the next stage. The property was extended to work with natural deduction proofs using shortcuts, as explained later by step 3+.

3.1.4 Step 4: Argumentation Logic Visualization

The fourth step requires the construction of a GUI that allows the visualization of Argumentation Logic proofs as sets of arguments. The proofs can originate from natural deduction proofs created automatically from stage 1 (with the help of stage 3), or constructed by the user using the GUI (step 1+). A lot of effort was put into making the GUI more approachable, which dictated that more surrounding features were implemented in order to provide a more well-rounded experience. This resulted in steps 4+ and 4++ explained later. The implementation of this step is described in chapter 7.

3.1.5 Step 5: Converting Natural Deduction Proofs to Argumentation Logic Proofs

The fifth stage revolves around the conversion of natural deduction proofs that are unsupported by Argumentation Logic (proofs that do not follow the Genuine Absurdity Property) to compatible ones (generated from stages 1 and 3). It can be shown in the technical report on Argumentation Logic that any proof not following the Genuine Absurdity Property can be converted to one that does for a consistent theory. The aim of this step is to allow the possibility of natural deduction proofs to be visualized from an argumentative view. However there was not enough time to implement this step. It therefore remains part of future work.

3.1.6 Step 6: Re-Introduction of Disjunction and Implication Connectives

The sixth stage involves the introduction of the disjunction and implication connectives. It is shown in the technical report on Argumentation Logic that for consistent theories using only conjunction and negation, Propositional Logic is equivalent to Argumentation Logic. The use of disjunction and implication remains partly subject for future work. The aim of this step was to explore further this area, but unfortunately this step was scrapped in favor of extra steps taken below in order to provide a more complete and well-rounded experience for the basic steps.

3.1.7 Step 7: Paraconsistency

The seventh and final stage ventures into how Argumentation Logic can allow for reasoning within an inconsistent environment. The aim of this step is to probe the notion of paraconsistency of Argumentation Logic. An implementation of the notions of AL-entailment and $AL+$-entailment was to be attempted in order to provide visual mapping of arguments coming from (directly or classically) inconsistent theories. This step was not implemented again in favor of a better experience regarding the first 4 steps.

3.1.8 Step 1+: Proof Builder

After the theorem prover in step 1 was created, proofs could be generated from it by providing the theory and desired goal. However, many times, looking through a large amount of generated proofs in order to find the desired one was found to be tedious, which called for the creation of a proof builder that would allow the user to input proofs directly. This feature is discussed in chapter 11.

3.1.9 Step 3+: Extending the Genuine Absurdity Property

This step enables the Genuine Absurdity Property to be extended so that it covers natural deduction proofs that make use of the substitution shortcut. This allows a wider range of proofs that can be considered, which are usually preferred since they look more pleasant to the human eye because of their conciseness. The process for extending the original definition of the Genuine Absurdity Property is shown in chapter 6.

3.1.10 Step 4+: Extracting proofs from arguments

The idea behind this step is to write a procedure that can create a natural deduction proof from a given theory and argument. The resulting proof, if visualized again should give back the same argument, essentially forming an inverse of the visualization algorithm in the original step 4. This step allows a more circular data flow by enabling the transition from arguments back to natural deduction. This step is discussed in chapter 8.

3.1.11 Step 4++: Argument Builder

In the same way as a proof builder feature made sense for complementing the theorem prover, an argument builder can not only aid the creation of arguments, but also allow arguments to be a data entry point, rather than existing only as post-processing data. This way, the natural deduction - argument cycle (made using steps 4 and 4+) can be started from the argument side as well. The argument builder is discussed in detail in chapter 12.

3.2 Solution Architecture

The entire project has been split into three parts, namely the core, the server and the client. The core contains algorithms and code written in order to implement procedures that allow for the exploration of Argumentation Logic. The server stands between the core and the client, and serves client requests by parsing and converting the requests to Prolog, using the core to run the requests and then replying with the results (after conversion again). The overall architecture is shown in Figure 3.1. The three parts will now be discussed further.

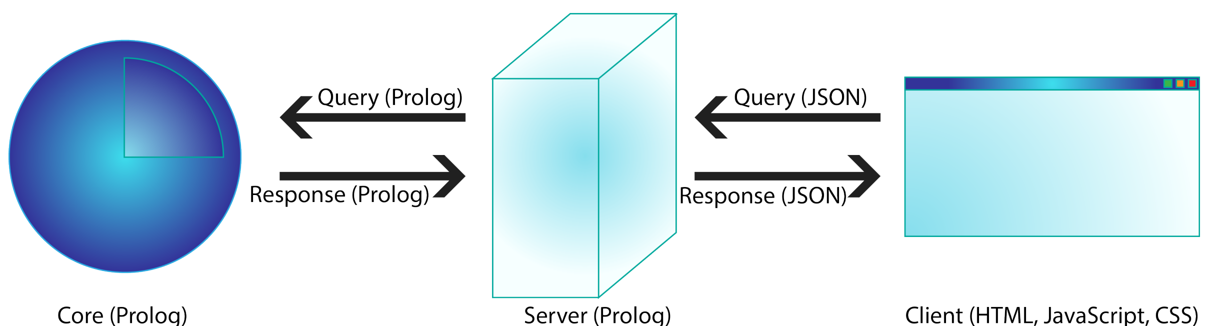

Figure 3.1: The high-level system architecture for the chosen solution

3.2.1 Core

The first (and arguably most important) part is the actual algorithms and code written to explore Argumentation Logic (henceforth referred to as ”core”). The core is written

entirely in Prolog, a procedural declarative programming language frequently used in the fields of artificial intelligence, logic and theorem proving.

Prolog was chosen as the implementation language because it often leads to clean and concise code. Concise code arguably leads to fewer bugs, something very important to this project, as soundness is mandatory. Prolog offers pattern matching, backtracking and unification, making it ideal for creating a theorem proving system.

The Prolog language has many implementations, out of which SWI-Prolog[1] was chosen because it is very fast and reliable, open-source and has a large array of helpful libraries and a large community.

3.2.2 Server

The second part of the project is the server. The server is used to load the core, and then serve HTTP (JSON) requests from the client by querying the core and replying with the results.

The server is written in Prolog as well. This has the obvious advantage of interoperability between the core and itself. The core's code (predicates) are loaded as part of the server's code and are used directly, and thus no middleware is required for the communication between the two parts.

The Prolog flavor used for the server is SWI-Prolog, which provides an HTTP package[2] that can set up and run a server with just a few lines of code. This virtually eliminated testing of this part of the project.

The server has exactly two responsibilities:

- to serve files needed for the client to run - this includes HTML, JavaScript and CSS files
- to respond to requests by converting the JSON requests to Prolog, querying the core, converting the results back to JSON and replying to the client

The server is discussed in further detail in chapter 9.

3.2.3 Client

The third and final part of the project is the client. The client serves as a front-end to the core, providing a helpful GUI as an alternative to the Prolog (interpreter's) command-line interface. It aims to provide useful facilities such as storage, importing and exporting of proofs and other data, syntax checking and so on.

The client consists of HTML, JavaScript and CSS files that are mainly served by the Prolog server. HTML 5 and JavaScript have become a very powerful combination that allow fast creation of elegant graphical interfaces, with many libraries available for free, making them a natural choice for a GUI.

The client is discussed in further detail in chapter 10.

3.3 Justification of Solution Architecture

There are several advantages in splitting the project as mentioned in section 3.2. However, as with any implementation, there are a few disadvantages as well. The pros and cons are discussed in this section.

[1] http://www.swi-prolog.org/
[2] http://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/http.html%27)

3.3.1 Advantages of Chosen Architecture

The focus of the design is modularity and reusability.

By completely detaching the core as a separate, self-contained module, testing becomes easier and more manageable. The responsibilities of this module become clearer. Since the core is completely detached, it can be run as-is, on the Prolog interpreter command-line just by itself, or it can be attached to a completely different program or GUI implementation or used in a different context.

The server can be swapped with a different implementation if required. This particular implementation is very small, largely due to the nature of the SWI-Prolog HTTP package that allows for a quick set-up of a webserver. The server currently has two responsibilities:

- to serve files needed for the client to run - this includes HTML, JavaScript and CSS files
- to respond to requests by converting the JSON requests to Prolog, querying the core, converting the results back to JSON and replying to the client

An alternate setup can have this server respond only to requests and have a different, finely-tuned server (such as Apache) handle the load-balancing and HTTP requests of the clients. This webserver chaining is similar to how the SWI-Prolog website works at the time of writing - there is an exposed Apache server that redirects requests to a SWI-Prolog server hidden from the outside world.

Since the project consists of a server and an HTML client, it can readily be hosted online and made available to the public. This way, a wider audience can be reached more quickly, which can result to valuable feedback.

The client is written in HTML, JavaScript and CSS, making it a web application. Advantages of web applications are fast, easy and transparent (to the user) updates, not requiring the user to re-download or update any software and cross-platform compatibility and hence larger availability. Web applications are ideal for incremental improvement, as a consequence of the previous advantages, and since this is largely a research project, in the future, it will be easier for features to be added as they are discovered. Finally, web applications integrate very well with other server-side services such as database access and account management, so the project can easily be extended in the future to support sharing of data and content via user accounts on the server.

3.3.2 Disadvantages of Chosen Architecture

As with every design, there are some drawbacks as well. There is some unavoidable duplication in data structures as different languages are used for the core and the client (Prolog vs JavaScript), and each language has to store and represent the data in some way. Some extra work is needed in order to keep the three parts of the system as modular as possible. Effort has to be made in synchronizing the different parts of the system when external interfaces are changed. For example, if the server is altered to accept a different data structure for the requests, then the client has to be update to provide requests that use the new data structure required by the server. Fortunately, internal changes (for example adding extra predicates or features to the core, adding extra UI elements to the client or revamping the GUI, or optimizing any of the three parts of the system) do not affect each module.

There are some disadvantages that come with web applications as well, but some of them fortunately do not apply to this system. Compatibility and performance issues (compared

to native applications) are some of the common problems with web interfaces. Since the server does the heavy lifting the client is left with the responsibility of running the a simple GUI, ruling out any potential issues with performance. Every effort is made to make the Client adhere to standards, reducing the chances of running into compatibility issues with the browser. The main disadvantage however is the need for constant connection to the server. Unless the server is run locally, the client cannot be used as all calculations take place server-side.

3.4 Functional Overview

The core is made of different predicates or procedures that work together in order to provide an ecosystem of functions that can be used to create and manipulate natural deduction proofs and arguments. The client further provides varying functionality that can work in conjunction with the core procedures. Figure 3.2 provides a functional map that illustrates the different procedures as well as the data flow around them.

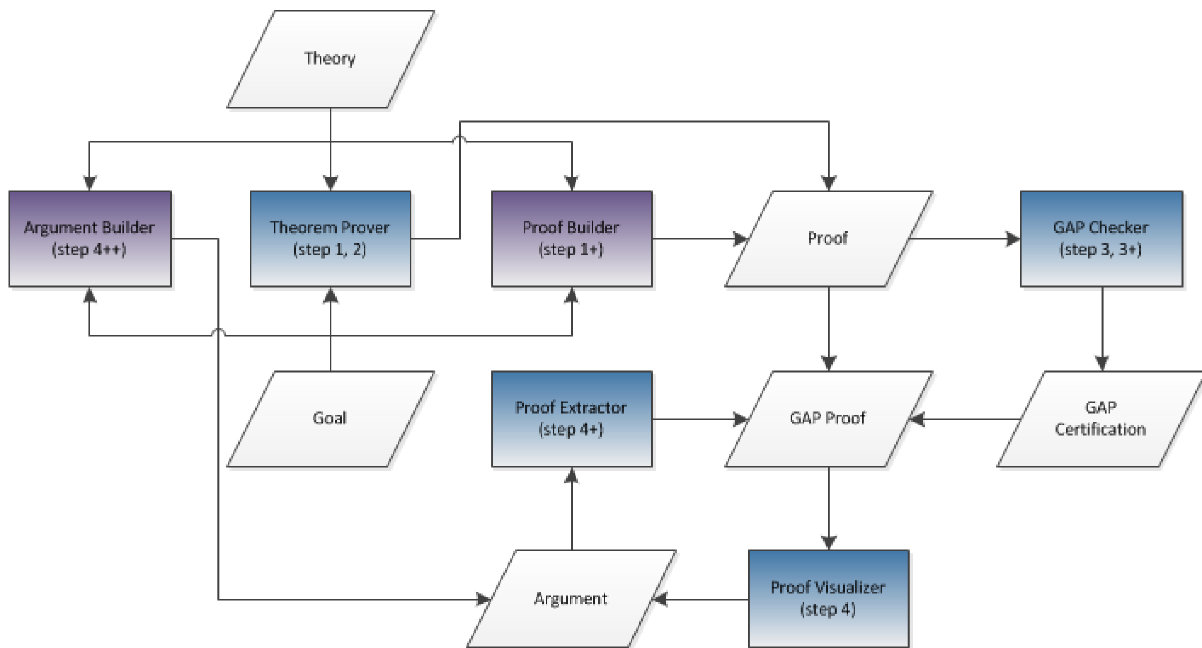

Figure 3.2: The high-level functional map for the core. White parallelograms represent data, purple boxes represent core predicates and blue boxes represent client functionality

The main procedures involved, shown in purple boxes in the diagram are the following:

- Theorem Prover (chapter 4): the theorem proof system provides proofs for the given theory and goal

  This is step 1 and 2 of the initial plan

- GAP Checker (chapter 5 and chapter 6): the Genuine Absurdity Property checker takes a proof and succeeds if the given proof indeed follows the property

  This procedure comprises steps 3 and 3+

- Proof Visualizer (chapter 7): the proof visualization predicate takes a proof that follows the Genuine Absurdity Property and returns an argument that can be used to visualize the given proof

This is step 4 of the original plan

- Proof Extractor (chapter 8): this predicate can take an argument (a visualization of a proof) and extract a proof from it

  This represents step 4+

In addition to the core module predicates, there are a few client features that can still provide data that can be used by the core modules listed above. These features are represented by the blue boxes in the diagram. These are:

- Proof Builder (chapter 11): provides a method of allowing the user to construct a particular proof

  This is step 1+ of the new steps

- Argument Builder (chapter 12): provides a method of allowing the user to construct an argument

  This is step 4++ of the originally unplanned steps

Chapter 4

Theorem Proving System

The first step in approaching this project is to build a theorem prover or a proof search system capable of generating proofs given a set of theory and a goal. Most existing implementations of such software use a different proof theory system (other than natural deduction) in order to generate proofs. Natural deduction, as discussed in subsection 2.2.3, is not the favorite choice for creating proofs. There are still however, a few implementations of a natural deduction theorem prover, but they did not meet all the requirements of this project and therefore were not suitable to use.

Different natural deduction provers use different sets of rules, some of which are derived. There are different styles of natural deduction as well, for example the Gentzen style that looks like sequent calculus where proofs following this format resemble trees, and the Fitch style natural deduction which resembles the format used in this paper.

Apart from these variations, the ability to turn off a few rules, such as the Reductio ad Absurdum rule (and the derived "proof by contradiction" rule which combines a Reductio ad Absurdum rule and a $\neg E$ rule), in order to enable $\vdash_{MRA}$ (or $\nvdash_{MRA}$) proving used by the Genuine Absurdity Property (chapter 5).

The theorem prover should also be flexible enough so that it can be used to stitch together sub-proofs in order to form a larger proof as required by the proof extraction algorithm discussed in chapter 8.

The latter two concerns were the main subject of step 2. A more interactive and direct way to generate proofs is provided by the implementation of step 1+ with the creation of the proof builder, discussed in chapter 11.

4.1 Ruleset Used

The implementation tries to approach the proof both backwards (from the bottom up) and forwards (from the top down). Several backwards and forwards rules have been implemented. Backwards rules try to close the gap between the theory and the goal from the bottom, and they are the ones that complete goals in order to finish the proof. The forwards rules are there in order to break down complicated formulas that have already been derived so that the backwards rules can complete their work. The forwards rules do not progress the proof in the sense that they never deal with the goal. They just provide simpler formulas that can be of use to the backwards rules.

The implementation was partly modeled after (or at the very least inspired by) the Carnegie Mellon Proof Tutor (introduced in subsection 2.2.4, the full paper is (Sieg and Scheines, 1992)). However, the implementation follows a Fitch-style natural deduction. There are also explicit rules for contradiction. No distinction exists between the available

assumptions α and derivable (using elimination rules) context β. Thus the whole context $\alpha\beta$ will be referred to as κ. The notation is borrowed from the full paper.

Listed below are the forward rules:

- $\wedge E : \kappa?G, \phi_1 \wedge \phi_2 \in \kappa, \phi_i \notin \kappa \Rightarrow \kappa, \phi_i?G, i = 1\, or\, 2\, or\, both$

  (this rule is the same as the one used in the Carnegie Mellon Proof Tutor)

- $\neg E : \kappa?G, \neg\neg\phi \in \kappa, \phi \notin \kappa \Rightarrow \kappa\phi?G$

  (if a double-negated formula is in the context, add the sub-formula without the double negation)

- $\perp I : \kappa?G, \phi \in \kappa, \neg\phi \in \kappa, \perp \notin \kappa \Rightarrow \kappa, \perp?G$

  (if two opposite formulas are in the context, add a contradiction)

None of the forward rules touches the goal. They just simplify derived conjunctions, double negations and add a contradiction if one can be established.

The backward rules are listed here:

- $\checkmark : \kappa?G, \gamma \in \kappa \Rightarrow \checkmark$

  (if the current goal is already derived, the proof is essentially complete)

- $\wedge I : \kappa?\gamma_1 \wedge \gamma_2, \gamma_1 \in \kappa, \gamma_2 \in \kappa \Rightarrow \checkmark$

  $\wedge I : \kappa?\gamma_1 \wedge \gamma_2, \gamma_i \notin \kappa \Rightarrow \kappa?\gamma_i, i = 1\, or\, 2\, or\, both$

  (if the goal is a conjunction, then prove whatever constituent parts are missing)

- $\vee I : \kappa?\gamma_1 \vee \gamma_2, \gamma_i \in \kappa \Rightarrow \checkmark$

  $\vee I : \kappa?\gamma_1 \vee \gamma_2, \gamma_i \notin \kappa \Rightarrow \kappa?\gamma_i, i = \ 1 or\, 2$

  (this rule is the same as the one used in the Carnegie Mellon Proof Tutor)

- $\vee E : \kappa?G, \phi_1 \vee \phi_2 \in \kappa \Rightarrow \kappa, \phi_1?G, \kappa, \phi_2?G$

  (use a disjunction to prove the goal using case by case analysis)

- $\perp E : \kappa?G, \perp \in \kappa \Rightarrow \checkmark$

  (anything can be derived if a contradiction has already been established)

- $\rightarrow I : \kappa?\gamma_1 \rightarrow \gamma_2 \Rightarrow \kappa, \gamma_1?\gamma_2$

  (this rule is the same as the one used in the Carnegie Mellon Proof Tutor)

- $\neg I : \kappa?\neg\gamma \Rightarrow \kappa, \gamma?\perp$

  (this rule is the same as the one used in the Carnegie Mellon Proof Tutor)

- $\rightarrow E : \kappa?G, \phi_1 \rightarrow \phi_2 \in \kappa \Rightarrow \kappa?\phi_1,\ \kappa, \phi_2?G$

  (this rule lies in between the forward and backward rule category, since it does not consume the goal, but it does however require that ϕ_1 be proven; what it says is try to prove the first part of the implication, and hence add the second part to the context before moving on to prove the goal)

- $PC : \kappa?\gamma, \gamma \neq \perp \Rightarrow \kappa, \neg\gamma?\perp$

  (proof by contradiction, implemented as a combination of $\neg I$ and $\neg E$ rules - this rule is the same as the one used in the Carnegie Mellon Proof Tutor)

- $\perp IE : \kappa?\gamma, \perp \notin \kappa, \neg\phi \in \kappa \Rightarrow \kappa?\phi$

  (take any negated formula and try to prove its positive subformula, essentially proving a contradiction; the goal is then vacuously true - composite rule implemented using $\perp I$ and $\perp E$ rules)

4.2 Propositional Logic Format

This implementation uses Prolog predicates in order to specify the different logic constructs. Figure 4.1 summarizes these predicates.

Logic Construct	Prolog Term Used By Implementation
Conjunction	`and(A, B)`
Disjunction	`or(A, B)`
Implication	`implies(A, B)`
Negation	`n(A, B)`
Contradiction	`falsity`

Figure 4.1: The Prolog constructs accepted and used by the theorem prover

As an example, consider the formula $a \wedge b \rightarrow \neg c$. This would be represented by the following data structure: `implies(and(a, b), n(c))`.

4.3 High-Level Description of Implementation

The theorem prover was built to be quite modular and offers both high-level and low level predicates. For the most part, high-level predicates are enough to cover the needs of the different algorithms created to check for the Genuine Absurdity Property, or to aid the building of an argument. It was necessary however at some point to use the low-level plumbing of the theorem prover in order to stitch together specific proofs generated with more complicated context for the proof extractor algorithm.

A summary of the high-level predicates is given below:

- `prove(Givens, Goal, Proof)`: The user supplies the theory as a list of formulas and a goal as a singleton list of one formula that needs to be proven. The predicate returns with `Proof` bound to one possible proof. The `prove/3` predicate offers choice-points, with each choice-point offering one solution. All solutions for the given theory and goal can be acquired by executing a `findall/3` predicate like this: `findall(Proof, prove([and(a,b)], [and(b,a)], Proof), AllProofs)` with `AllProofs` containing all the proofs what prove the goal.

- `proveMRA(Givens, Goal, Proof)`: This works in the same way as `prove/3`, but it makes no use of the $\neg I$ and (derived) proof by contradiction rules. This can be used to check for the Genuine Absurdity Property or to help build arguments (chapter 12).

- `provable(Givens, Goal, MRA, Verdict)` This works similarly to the above predicates, but instead of supplying a proof upon return, it only responds with a `yes` or `no` atom, depending on whether the supplied goal can be proven using the given theory. The `MRA` field takes a `yes` or `no` atom and defines whether $\vdash$ or $\vdash_{MRA}$ is used to prove the goal (ie whether $\neg I$ and proof by contradiction can be used). This predicate always finishes without any choice-points.

The above predicates handle the burden of converting the input into steps (see section 4.4), and then calling the necessary low-level predicate to fire off the process. The theorem prover however was built to work even if the user chooses to execute a low-level predicate instead (assuming correct input was given). This enables the user to, for example, start a proof with the application of a particular rule. The available low-level predicates are summarized below:

- `backwardProve(MRA, Steps, Context, Extras, Goal, Proof)`: This predicate starts the backwards proving of the goal, by trying to use any of the available rules depending on whether `MRA` is set to `yes` or `no`. The steps parameter contains the actual steps of the current context, and the context parameter specifies additional context (that was inherited by ancestors).
- `check(MRA, Steps, Context, Extras, Goal, Proof)`
- `falsityI(MRA, Steps, Context, Extras, Goal, Proof)`
- `andI(MRA, Steps, Context, Extras, Goal, Proof)`
- `orI(MRA, Steps, Context, Extras, Goal, Proof)`
- `orE(MRA, Steps, Context, Extras, Goal, Proof)`
- `impliesI(MRA, Steps, Context, Extras, Goal, Proof)`
- `notI(MRA, Steps, Context, Extras, Goal, Proof)`
- `forward(MRA, Steps, Context, Extras, Goal, Proof)`
- `falsityIE(MRA, Steps, Context, Extras, Goal, Proof)`
- `impliesE(MRA, Steps, Context, Extras, Goal, Proof)`
- `proofByContradiction(MRA, Steps, Context, Extras, Goal, Proof)`

All of the backward rules are available to be called immediately by the user, and they will generate a proof with an initial application of that rule. If given the right input the rules will take care of the rest of the proof as well and come back with a complete proof of the goal. The "forward rule" is a "rule" that tries to apply all the forward rules and call `backwardProve/6` again.

4.4 Output Format

The output of the theorem prover is a Prolog list of steps or boxes. A step is defined as `step(Derivation, Reason, LineNumber)` where `Derivation` is a formula using Prolog terms, the reason is a list with at least one element inside, and a line number is just a non-negative number that is used to uniquely identify the step (so that other steps' reasons can reference that step). A box contains sub-proofs (that begin with a hypothesis) that are essentially lists of more steps and boxes (defined as `box(SubProof)`). There is also a `dbox(SubProof1, SubProof2)` that is used only by the $\vee E$ rule which contains two boxes side by side for the case by case analysis.

The reason is a list where the first element is the name of the justification for the derivation in that step as a Prolog atom. Some reasons are required to reference other steps in the proof in order to fully justify how the formula in the step was derived. For

example, for an application of the $\wedge I$ rule, both subformulas of the conjunction need to be referenced by including the numbers of the steps containing those subformulas in the list.

Reasons include the natural deduction rules specified before. They also contain a few more reasons such as `hypothesis` to indicate a hypothesis or `given` to indicate a part of the theory. Figure 4.2 lists the reasons and the number of reference line numbers required by each reason.

Reason Prolog Term	Reference Line Numbers	Description
`check`	1	Reiteration of an existing formula
`andI`	2	$\wedge I$ from subformulas on referenced lines
`andE`	1	$\wedge E$ from conjunction on referenced line
`orI`	1	$\vee I$ from subformulas on referenced lines
`orE`	5	$\vee E$ from disjunction, two subformulas of the disjunction as hypotheses and two same conclusions on referenced lines
`impliesI`	2	$\rightarrow I$ from subformulas on referenced lines
`impliesE`	2	$\rightarrow E$ from implication and its first part indicated by referenced lines
`notI`	2	$\neg I$ from hypothesis and contradiction on referenced lines
`notE`	1	$\neg E$ from double-negated formula on referenced line
`falsityI`	2	$\perp I$ from opposite formulas on referenced lines
`falsityE`	1	$\perp E$ from contradiction indicated by referenced line
`hypothesis`	0	a hypothesis put forth
`given`	0	part of the theory

Figure 4.2: The Prolog constructs accepted and used by the theorem prover

The list contains elements in the opposite order as one would read a proof on paper. That is, the first element in the list will be the conclusion of the proof, and the last elements of the list will be the givens (theory) that are always located at the beginning of the proof. The reason as to why the proof is in the opposite order is that Prolog offers a concise way to append elements to a list, and the backward rules tend to add steps to the end of the proof instead of the beginning. Therefore the proof is build backwards in a sense. Predicates are provided that can flip the proof as necessary and are indeed used when responding to the client with generated proofs from the theorem prover.

As an example of the output generated by the theorem prover, consider the proof and its corresponding data structure as generated by the theorem prover in Figure 4.3.

4.5 Remarks

As mentioned in the evaluation of the theorem prover in section 13.1, one of the improvements that could be made is to increase the performance of the theorem prover. This could involve either a more efficient implementation altogether, or the introduction of optimizations with regards to the search space or pruning of (paths that lead to) "bad" proofs. The latter would involve defining explicitly what "bad" proofs are with respect to Argumentation Logic.

1	$\neg(\alpha \wedge \beta)$	given
2	$\neg(\alpha \wedge \neg\beta)$	given
3	$\neg\beta$	given
4	α	hypothesis
5	β	hypothesis
6	$\alpha \wedge \beta$	$\wedge I(4,5)$
7	$\bot$	$\bot I(1,6)$
8	$\neg\beta$	$\neg I(5,7)$
9	$\alpha \wedge \neg\beta$	$\wedge I(4,8)$
10	$\bot$	$\bot I(2,9)$
11	$\neg\alpha$	$\neg I(4,10)$

```
[
        step(n(a), [notI, 4, 10], 11),
        box([
                step(falsity, [falsityI, 2, 9], 10),
                step(and(a, n(b)), [andI, 4, 8], 9),
                step(n(b), [notI, 5, 7], 8),
                box([
                        step(falsity, [falsityI, 1,
                            6], 7),
                        step(and(a, b), [andI, 4, 5],
                            6),
                        step(b, [hypothesis], 5)
                ]),
                step(a, [hypothesis], 4)
        ]),
        step(n(b), [given], 3),
        step(n(and(a, n(b))), [given], 2),
        step(n(and(a, b)), [given], 1)
]
```

Figure 4.3: A natural deduction proof and the corresponding output from the theorem prover

A change of scope may also be in order, if it turns out that the ”find all proofs” functionality is no longer needed. This would allow the theorem prover to take on more aggressive pruning and optimization that might reduce the search space of the prover, but at the same time making it faster in responding appropriately when tasked with finding out whether a given goal can be proven from a given theory (which is the main focus of the Genuine Absurdity Property check described in chapter 5).

Chapter 5

Checking for Genuine Absurdity Property

This chapter refers to the implementation of the Genuine Absurdity Property check and comprises step 3 of the initial plan.

Recall from section 2.3.4 that for a RAND derivation, $T \cup \{\phi\} \cup \{\phi_1, ..., \phi_k\} \cup \{\neg\psi_1, ..., \neg\psi_l\} \vdash_{MRA} \bot$, where $k, l \geq 0$ and

- ϕ is the hypothesis of this derivation
- ϕ_i are the hypothesis of parent derivations that this derivation has access to and can make use of
- ψ_i are the hypotheses of the children derivations, the negations of which can be used by this derivation

Moreover, recall that the Genuine Absurdity Property forms a kind of relevance by requiring that the hypothesis ϕ is necessary for the derivation of the contradiction. In other words, without ϕ, a contradiction cannot be established. In formal notation, this would be described as

$$T \cup \{\phi_1, ..., \phi_k\} \cup \{\neg\psi_1, ..., \neg\psi_l\} \nvdash_{MRA} \bot$$

Recall, further, that the Genuine Absurdity Property is a recursive property in that any sub-derivations of an application of the Reductio ad Absurdum rule must also follow this property, and finally, that this property is defined only over proofs using conjunction and negation only. Note once again, that the set $\{\phi_1, ..., \phi_k\}$ includes all ancestor hypotheses copied from outside the sub-derivation.

5.1 Short Description of Algorithm

In a nutshell, the algorithm for checking for the Genuine Absurdity Property traverses the proof and keeps track of ancestor hypotheses and child hypotheses as well as the theory given initially. At the beginning the algorithm checks that the given proof is a RAND proof and that it only contains conjunctions and negations of atoms. In addition, it checks that the proof does not make use of any shortcuts. Those are dealt in the extended version of the Genuine Absurdity Property discussed in chapter 6.

The ancestor hypotheses are tracked by crawling the proof backwards from each line in the sub-derivation, using the justifications as (potentially forking) paths to guide the

search. The child hypotheses are tracked by reading the hypothesis at the top of each sub-derivation and negating it.

Before reaching the end of the current application of the $\neg I$ rule, each sub-derivation is also checked that it follows the Genuine Absurdity Property by calling the algorithm again recursively.

At the end of each application of the Reductio ad Absurdum rule the Genuine Absurdity Property is checked by asking the theorem prover in chapter 4 to try and prove a contradiction using the given theory, and the ancestor and (negations of the) child hypotheses. That is, we ask the prover to prove that $T \cup \{\phi_1, ..., \phi_k\} \cup \{\neg\psi_1, ..., \neg\psi_l\} \vdash_{MRA} \bot$. If that succeeds, then we know that a contradiction could have been reached without the use of the hypothesis in the current application of the Reductio ad Absurdum rule, thus making the proof not follow the property under check.

5.2 Details of Implementation

The overall algorithm is split into four stages. The first stage involves checking whether the given proof is a RAND proof. That is, apart from the theory, the proof should only contain an application of the Reductio ad Absurdum rule on the top level. The second stage involves checking whether the given proof has formulas consisting of only conjunction and negation. The third stage checks that the proof does not make any use of the substitution rule. The fourth and final stage is the actual check which verifies the Genuine Absurdity Theory.

```
1  % Checks that the given proof follows the GAP property
2  % Gap checker assumes valid propositional logic proofs
3  checkGAP(Proof) :-
4         reverse(Proof, RevProof),
5         checkRAND(RevProof), !,
6         checkRestrictedRules(RevProof), !,
7         getTheoryAndRevBox(RevProof, Theory, RevBox), !,
8         checkRestrictedTheory(Theory), !,
9         checkPureND(Theory, RevBox), !,
10        checkGAP(Theory, _, [], [], RevBox), !.
```

Listing 5.1: Genuine Absurdity Property top level predicate

Listing 5.1 shows the top-level predicate where the four stages (checks) can be seen at line 5 for RAND proof, lines 6 and 7 for the restricted use of propositional logic, line 9 for the lack of shortcuts and finally line 10 for the actual check for the definition of the Genuine Absurdity Property. The format of the proof is the same as that output by the theorem proving algorithm in chapter 4. The four stages are further discussed below.

5.2.1 Checking for RAND Proof

The format of a RAND proof is the following: first, there may or may not be a few steps containing the theory. These are steps that are justified with "given". Following should be a box, and finally, there should be either the goal (derived using the box), or the double negation of the goal (derived using the box) and the goal (derived by $\neg E$).

```
1  % Checks to see if this proof is a RAND proof to start with
2  % A RAND proof is of the form: [givens]*, [box], ([step:notE,
      step:notI]||[step:notI])
3  checkRAND(Proof) :- checkRAND(givens, Proof).
```

```
4  checkRAND(givens, [box(_)|Proof]) :- checkRAND(box, Proof).
5  checkRAND(givens, [step(_, [given], _)|Proof]) :- checkRAND(givens, Proof).
6  checkRAND(box, [step(_, [notI|_], _)]).
7  checkRAND(box, [step(_, [notI|_], _), step(_, [notE, _], _)]).
```

Listing 5.2: Checking whether a proof is a RAND proof

On line 3 of Listing 5.2 the `checkRAND/1` predicate starts the check by calling `checkRAND/2` and specifying that it is at the "givens" stage (ie that it is now going through the theory). Line 5 unwinds the proof by iterating through the theory until a box is found and picked up by line 4, which marks the "box" stage. Finally, lines 6 and 7 check that the last step of the proof is the conclusion, which might be preceded by its double negation (which is the product of the $\neg I$ rule and the box before.

5.2.2 Checking for Restricted Formulas

It suffices to check in a proof that the theory is constructed using conjunction and negation, and that the rules applied subsequently belong to the set of $\{\wedge I, \wedge E, \neg I, \neg E, \bot I, \bot E\}$.

```
1  % Checks to see if the proof consists of ruleset defined over argumentation logic
2  validRules([andI, andE, notI, notE, falsityI, falsityE, given, check,
       hypothesis]).
3  checkRestrictedRules([]).
4  checkRestrictedRules([step(_, [Reason|_], _)|Proof]) :-
5          validRules(ValidRules),
6          m2(Reason, ValidRules),
7          checkRestrictedRules(Proof).
8  checkRestrictedRules([box(SubProof)|Proof]) :-
9          checkRestrictedRules(SubProof),
10         checkRestrictedRules(Proof).
11 checkRestrictedTheory([]).
12 checkRestrictedTheory([Given|Theory]) :-
13         checkRestrictedFormula(Given),
14         checkRestrictedTheory(Theory).
15 checkRestrictedFormula(X) :-
16         atom(X).
17 checkRestrictedFormula(and(X, Y)) :-
18         checkRestrictedFormula(X),
19         checkRestrictedFormula(Y).
20 checkRestrictedFormula(n(X)) :-
21         checkRestrictedFormula(X).
```

Listing 5.3: Checking whether a proof uses only conjunction and negation

The predicate `checkRestrictedTheory/1` as shown in Listing 5.3 looks at the theory contained in the proof and makes sure that it consists only of atoms, conjunctions and negations. The predicate `checkRestrictedRules/1` shown in Listing 5.3 again ensures that rules that could potentially introduce a new construct (such as implication or disjunction) are not used. This predicate probes nested boxes as well so that the entire proof is covered. The predicate `m2/2` on line 6 acts as an alias to the standard `member/2` Prolog predicate.

5.2.3 Ensuring the Lack of Substitution

This stage looks at the line number references that the steps inside a derivation address, and checks that all line numbers either refer to steps inside the current context, theory, or

parent hypotheses. Any other references are considered extraneous.

```
1  % Checks to see if this proof does not make references to external derivations
2  checkPureND(Theory, Proof) :-
3         length(Theory, TheoryLength),
4         checkPureND(TheoryLength, _, Proof, Proof).
5  checkPureND(_, _, [], _) :- !.
6  checkPureND(L, _, [step(_, [hypothesis], LN)|Proof], WholeProof) :-
7         !, checkPureND(L, LN, Proof, WholeProof).
8  checkPureND(L, LN, [step(_, [_|ReasonLines], _)|Proof], WholeProof) :-
9         forall(m2(RL, ReasonLines), (RL >= LN; RL < L; getStep(RL, WholeProof,
              step(_, [hypothesis], RL)))),
10        checkPureND(L, LN, Proof, WholeProof).
11 checkPureND(L, LN, [box(BoxProof)|Proof], WholeProof) :-
12        checkPureND(L, _, BoxProof, WholeProof),
13        checkPureND(L, LN, Proof, WholeProof).
```

Listing 5.4: Checking whether a proof uses any shortcuts

Listing 5.4 shows predicate checkPureND/2 which takes the theory and the proof under examination. It measures the size of the theory, and passes this information along with the proof to checkPureND/4. Line 7 makes a note of the line number of the hypothesis, which is the first step of each sub-derivation. Line 9 checks that all referenced steps by the line currently under consideration are either internal steps (bigger than the hypothesis line number measured by line 7 of the code) or point to the theory (any line references smaller than the length of the theory are considered acceptable as the theory always appears at the top of the proof), or that they point to a hypothesis (of an ancestor). The last clause of checkPureND makes sure that all sub-derivations are visited.

5.2.4 Checking for Genuine Absurdity Property

The final part of the algorithm deals with the definition of the Genuine Absurdity Property directly.

```
1  % Checks for the actual GAP for each (sub)derivation in the proof
2  checkGAP(Theory, AncestorHypotheses, ChildHypotheses, SiblingHypotheses, [], _,
       _) :-
3         a4(Theory, AncestorHypotheses, ChildHypotheses, SiblingHypotheses,
              Context),
4         not(proveMRA(Context, [falsity], _)).
5  checkGAP(Theory, [], ChildHypotheses, SiblingHypotheses, [step(_, [hypothesis],
       HL)|Proof], WholeProof, _) :-
6         !, checkGAP(Theory, [], ChildHypotheses, SiblingHypotheses, Proof,
              WholeProof, HL).
7  checkGAP(Theory, AncestorHypotheses, ChildHypotheses, SiblingHypotheses, [step(_,
       [_|Reason], _)|Proof], WholeProof, HL) :-
8         getUsedHypotheses(Theory, Reason, WholeProof, HL, NewAncHypotheses,
              NewSibHypotheses),
9         a2(NewSibHypotheses, SiblingHypotheses, NewSiblingHypotheses),
10        a2(NewAncHypotheses, AncestorHypotheses, NewAncestorHypotheses),
11        checkGAP(Theory, NewAncestorHypotheses, ChildHypotheses,
              NewSiblingHypotheses, Proof, WholeProof, HL).
12 checkGAP(Theory, AncestorHypotheses, ChildHypotheses, SiblingHypotheses,
       [box(BoxProof)|Proof], WholeProof, HL) :-
13        checkGAP(Theory, [], [], [], BoxProof, WholeProof, _),
14        BoxProof = [step(ChildHypothesis, [hypothesis], _)|_],
```

```
15          (
16                 ChildHypothesis = n(X),
17                 NegatedChildHypothesis = X;
18
19                 NegatedChildHypothesis = n(ChildHypothesis)
20          ),
21          checkGAP(Theory, AncestorHypotheses,
                [NegatedChildHypothesis|ChildHypotheses], SiblingHypotheses, Proof,
                WholeProof, HL).
```

Listing 5.5: Checking whether a proof follows the Genuine Absurdity Property

The format of the proof passed in this predicate is the opposite of that output by the theorem prover in chapter 4 in that the steps are in increasing order as one would read the proof on paper. This is due to the work of the `getTheoryAndRevBox/3` predicate called before the check as shown in Listing 5.1.

The clause on line 4 of Listing 5.5 makes a note of the location (line number) of the hypothesis. It is the one and only clause that Prolog chooses upon entering a new box (application of the Reductio ad Absurdum rule). All intermediate steps bear little significance in what they actually do, but it is very important to check what steps of the proof they refer to, and extract potential ancestor hypotheses. This is done by the clause on line 7, using `getUsedHypotheses/6`, which returns the ancestor and sibling hypotheses used. The latter set will of course be empty and will not affect the check for the original definition. This is because it is known from the previous check for the lack of substitutions (shortcuts) that there are no references to external conclusions (sibling derivations).

Child proofs however must be checked for the Genuine Absurdity Property as well as this property is recursive. Line 7 takes that into account. It adds the hypothesis of the current context into the ancestor hypotheses of a new call to the `checkGAP/7` predicate (line 9). If the sub-derivation follows the property, the algorithm continues to add the negation of the sub-derivation hypothesis to the child hypotheses.

At the end of the current context, when all the child hypotheses have been gathered and checked that they follow the property, the algorithm checks that the property holds for this context as well. This is done by the clause on line 2. The predicate `a3/4` ("append 3 lists") on line 3 merges the theory, ancestor hypotheses and child hypotheses gathered so far into one bundle, and calls for the theorem prover to try and prove a contradiction while purposely excluding the hypothesis of the current context from the bundle (line 4). That is, the theorem prover is given the task of proving $T \cup \{\phi_1, ..., \phi_k\} \cup \{\neg\psi_1, ..., \neg\psi_l\} \vdash_{MRA} \bot$. If it succeeds, then the Genuine Absurdity Theory does not hold. If it fails, then so far the property holds.

Line 8 scans the line references of intermediate steps in the current derivation to find references to sibling derivations. The scanning and gathering of referenced siblings is done by `getUsedSiblingHypotheses/6` which is shown in Listing 5.6.

```
1  % Uses the line references to find referenced sibling derivations
2  getUsedHypotheses(_, [], _, _, [], []) :- !.
3  getUsedHypotheses(Theory, Reason, WholeProof, HL, NewAncHypotheses,
       NewSibHypotheses) :-
4          length(Theory, L),
5          findall(R, (m2(R, Reason), R < HL, R >= L), External),
6          findall([H1, E1], (m2(E1, External), getStep(E1, WholeProof, step(H1,
               [notI|_], _))), HLN1),
7          unzip(Hs1, LNs1, HLN1),
8          subtract(External, LNs1, Rest1),
```

```
9         findall([H2, E2], (m2(E2, Rest1), getStep(E2, WholeProof, step(H2,
              [hypothesis], _))), HLN2),
10        unzip(Hs2, LNs2, HLN2),
11        subtract(Rest1, LNs2, Rest2),
12        findall(Reason2, (m2(R2, Rest2), getStep(R2, WholeProof, step(_,
              [_|Reason2], _))), Reasons),
13        append(Reasons, Reasons2),
14        getUsedHypotheses(Theory, Reasons2, WholeProof, HL, NewAncHypotheses2,
              NewSibHypotheses2),
15        append(Hs1, NewSibHypotheses2, NewSibHypotheses),
16        append(Hs2, NewAncHypotheses2, NewAncHypotheses).
17
18 unzip([], [], []).
19 unzip([L|Ls], [R|Rs], [[L,R]|Ps]) :- unzip(Ls, Rs, Ps).
```

Listing 5.6: Gathering of referenced ancestor derivations for the original Genuine Absurdity Property definition

The predicate `getUsedSiblingHypotheses/6` works by measuring the size of the theory in steps (line 4 of the code). Any line references smaller than this number indicate a reference to the theory (since the theory appears first in the proof, its line numbers are always smaller than the size of the theory in steps). When called, this predicate knows the line number of the hypothesis of the current derivation. Any reference to a line greater than (or equal to) the line number of the current hypothesis indicates an internal reference and can therefore be ignored. This filtering is done by line 5, that only gathers extraneous references that do not point to the theory.

Out of these, all steps that are conclusions are filtered and stored (line 6). These constitute sibling conclusion references (indirectly, references to sibling hypotheses). Since the check performed by the `checkPureND/2` guarantees that there are no references to siblings, this line has no effect. This piece of code plays an important role in the extended version of the Genuine Absurdity Property discussed in chapter 6.

The remaining extraneous references are checked whether they point to an ancestor hypothesis (line 9) and if that is the case, the hypothesis is stored. The remaining extraneous references are then recursively explored by lines 12-14 of the code until the recursion bottoms out. The extracted ancestor conclusions from the recursion at deeper levels are appended to the ones found in the current level of recursion and when the recursion bubbles back up, all ancestor hypotheses referenced by the examined step are returned. The sibling hypotheses returned when checking for the original definition of the property is just an empty list.

5.3 Example Walkthrough

This example will focus on the `checkGAP/7` predicate shown in Listing 5.5. Consider a proof with theory $\{\neg(\beta \wedge \alpha), \neg(\alpha \wedge \gamma), \neg(\alpha \wedge \neg\beta \wedge \neg\gamma)\}$ and goal $\neg\alpha$. The proof is given below:

1	$\neg(\beta \wedge \alpha)$	given
2	$\neg(\neg\beta \wedge \gamma)$	given
3	$\neg(\alpha \wedge \neg\beta \wedge \neg\gamma)$	given
4	α	hypothesis
5	β	hypothesis
6	$\beta \wedge \alpha$	$\wedge I(5,4)$
7	$\bot$	$\bot I(1,6)$
8	$\neg\beta$	$\neg I(5,7)$
9	γ	hypothesis
10	$\alpha \wedge \gamma$	$\wedge I(4,9)$
11	$\bot$	$\bot I(2,10)$
12	$\neg\gamma$	$\neg I(9,11)$
13	$\alpha \wedge \neg\beta$	$\wedge I(4,8)$
14	$\alpha \wedge \neg\beta \wedge \neg\gamma$	$\wedge I(13,12)$
15	$\bot$	$\bot I(3,14)$
16	$\neg\alpha$	$\neg I(4,15)$

The `checkGAP/5` predicate is called with theory $[\neg(\beta \wedge \alpha), \neg(\alpha \wedge \gamma), \neg(\alpha \wedge \neg\beta \wedge \neg\gamma)]$, and empty lists for the ancestor and child hypotheses. It is also given the outer box (lines 4 to 15 of the proof).

Line 5 of the code in Listing 5.5 picks up the location (line number) of the hypothesis α on line 4 of the proof while at the same time consuming this line of the proof and calling itself again, passing in the hypothesis location just picked up. The current context looks like this:

- hypothesis line number: 4
- ancestor hypotheses: $\emptyset$
- child hypotheses: $\emptyset$

Right away a box is encountered, and the clause on line 12 of the code is executed, which calls itself again and re-initializing the ancestor and child hypotheses to an empty list for the recursive call.

Inside the recursive call, in the new context, line 5 executes again, which picks up the new hypothesis β location and consumes line 5 of the proof. The current context looks like this:

- hypothesis line number : 5
- ancestor hypotheses: $\emptyset$
- child hypotheses: $\emptyset$

Next, the clause on line 7 executes which uses `getSiblingHypotheses/6` to find the reference (the implicit copy) of ancestor hypothesis α. The aforementioned predicate explores all line references smaller than the current hypothesis line number (5 in this case) to find referenced ancestor hypotheses. The hypothesis is then added to the ancestor hypotheses set. The current context, after consuming line 6 of the proof looks like this:

- hypothesis line number : 5

- ancestor hypotheses: $\{\alpha\}$
- child hypotheses: $\emptyset$

Line 7 of the code executes again for line 7 of the proof but no change is made as this line makes only internal references. The box is now empty.

Since the box is now empty, the first clause (line 2) is executed, which checks that a contradiction cannot be proven just by using the theory and ancestor hypothesis α without the need for the current hypothesis β. That is, it checks that $T \cup \{\alpha\} \nvdash_{MRA} \bot$.

Having dealt with the child derivation, the algorithm bubbles back up to the execution of line 13 of the code where it previously left off. The box containing lines 5-7 of the proof has been consumed. The child hypothesis is extracted and the negation of it is added to the child hypotheses before recursively calling itself once more. Note that in this context α is the hypothesis and is not a member of the ancestor hypotheses. The current context looks like this:

- hypothesis line number: 4
- ancestor hypotheses: $\emptyset$
- child hypotheses: $\{\neg\beta\}$

Line 7 of the code steps over line 8 of the proof.

Here, another box is encountered and the clause on line 12 executes once more, re-initializing the ancestor and child hypotheses for the recursive call.

Inside the recursive call, in the new context, line 5 executes again, which picks up the new hypothesis γ line number and consumes line 9 of the proof. The current context looks like this:

- hypothesis line number: 9
- ancestor hypotheses: $\emptyset$
- child hypotheses: $\emptyset$

Lines 10 and 11 of the proof are handled by the clause on line 7 of the code listing. In the first line consumed the copied hypothesis α is found and the current context looks like the following:

- hypothesis line number: 9
- ancestor hypotheses: $\{\alpha\}$
- child hypotheses: $\emptyset$

Since this box is now empty, the first clause (line 2) is executed and checks that a contradiction cannot be proven just by using the theory and ancestor hypothesis α without the need for the current hypothesis γ. That is, it checks that $T \cup \{\alpha\} \nvdash_{MRA} \bot$.

This child derivation is dealt with as well and the algorithm goes back to the execution of line 13 of the code where it left off. The box containing lines 9-11 of the proof has been consumed, and the negation of the child hypothesis $\neg\gamma$ is added to the list of child hypotheses. The current context looks like this:

- hypothesis line number: 4
- ancestor hypotheses: $\emptyset$

- child hypotheses: $\{\neg\beta, \neg\gamma\}$

Lines 12-15 of the proof are processed in vain by the clause on line 7 of the code listing, which results in an empty box.

The empty box calls for the clause on line 2 of the code a final time, which checks that a contradiction cannot be proven just by using the theory and children hypotheses $\neg\beta$ and $\neg\gamma$ without the need for the current hypothesis α. That is, it checks that $T \cup \{\neg\beta, \neg\gamma\} \nvdash_{MRA} \bot$. After this check, the algorithm ends as the top-level RAND derivation has been proven to follow the Genuine Absurdity Property.

5.4 Remarks and Limitations

It is worth noting that for the algorithm to work, the theorem prover must return true whenever $T \cup \{\phi_1, ..., \phi_k\} \cup \{\neg\psi_1, ..., \neg\psi_l\} \vdash_{MRA} \bot$ is the case. This means that the theorem prover must be complete, in addition to being sound. If the use of an incomplete theorem prover is employed, then a proof might be deemed to be following the Genuine Absurdity Property because the theorem prover might fail to prove the contradiction of a $\neg I$ rule application (without the hypothesis), rather than because it really might be the case that $T \cup \{\phi_1, ..., \phi_k\} \cup \{\neg\psi_1, ..., \neg\psi_l\} \nvdash_{MRA} \bot$.

A limitation of the Genuine Absurdity Property as it is currently defined is that it is not defined over proofs that make use of the substitution rule, that is, reusing previously derived formulas in order to cut down on the proof length. An example of a proof that goes the long way in proving its goal and a similar one that takes a shortcut can be seen in Figure 2.3. This limitation is overcome by taking sibling derivations into account as discussed in chapter 6.

Chapter 6

Extending the Genuine Absurdity Property

As briefly mentioned in section 2.3.4, the standard definition for the Genuine Absurdity Property comes with a shortcoming. It only considers proofs where the substitution rule is not permitted, which is why only ancestor and child hypotheses are taken into account. This chapter explains how the current definition of the Genuine Absurdity Property can be extended to accommodate proofs of that kind, as many proof search software or even humans tend to use shortcuts or derived rules in order to shorten proofs by decreasing repetition and reduce the amount of effort needed to build a proof. This chapter covers step 3+ of section 3.1.

6.1 Arriving at the Definition

This section starts from the classic definition of the Genuine Absurdity Property and arrives at the final definition. This section can be skipped without any loss of context. The final definition is given again in the next section.

As a first step, extend the definition of the Genuine Absurdity Property from $T \cup \{\phi_1, ..., \phi_k\} \cup \{\neg\psi_1, ..., \neg\psi_l\} \nvdash_{MRA} \perp$ to $T \cup \{\phi_1, ..., \phi_k\} \cup \{\neg\psi_1, ..., \neg\psi_l\} \cup \{\neg\chi_1, ..., \neg\chi_m\} \nvdash_{MRA} \perp$, where ϕ_i is an ancestor hypothesis, ψ_i is a child hypothesis and χ_i is a sibling hypothesis.

The above suggestion could be considered as being incorrect, because of the following: assume $\neg\chi_i$ is a sibling conclusion that happens after the box under examination. Clearly because it happens later in the proof it cannot be referenced by the current context, so allowing it to influence the result is unfair.

The next obvious move could be to enforce some ordering and redefine the extension definition (henceforth known as "GAPX") to be $T \cup \{\phi_1, ..., \phi_k\} \cup \{\neg\psi_1, ..., \neg\psi_l\} \cup \{\neg\chi_1, ..., \neg\chi_m\} \nvdash_{MRA} \perp$, where $\chi_m < \phi$ (and $\chi_i < \chi_j$ for $i < j$), where ϕ is the current hypothesis. The relation $<(x, y)$ means that x precedes y in the proof. In other words, all sibling hypotheses that happened before are taken into account only.

This definition may be regarded as a bit naive, since by changing the ordering of sibling derivations the outcome can change. As an abstract example, consider sibling derivations A and B. Consider whether A follows GAPX or not. It could be the case that if B happens before A, A is not GAPX-compliant. If B happens after A, then A might follow the property. Consider a more practical example as shown in Figure 6.1.

Different results are produced just by swapping the order of the sibling derivations. This is because in the $[\gamma ... \perp]$ box, in the first proof, $\neg\beta$ is available and can be used in conjunction with the last bit of theory $\neg(\alpha \wedge \neg\beta)$; in the second proof, it is not available

Line	Formula	Justification
1	$\neg(\alpha \wedge \beta)$	given
2	$\neg(\alpha \wedge \gamma)$	given
3	$\neg(\alpha \wedge \neg\beta \wedge \neg\gamma)$	given
4	$\neg(\alpha \wedge \neg\beta)$	given
5	α	hypothesis
6	β	hypothesis
7	$\alpha \wedge \beta$	$\wedge I(5,6)$
8	$\bot$	$\bot I(1,7)$
9	$\neg\beta$	$\neg I(6,8)$
10	γ	hypothesis
11	$\alpha \wedge \gamma$	$\wedge I(4,10)$
12	$\bot$	$\bot I(2,11)$
13	$\neg\gamma$	$\neg I(9,12)$
14	$\alpha \wedge \neg\beta$	$\wedge I(4,8)$
15	$\alpha \wedge \neg\beta \wedge \neg\gamma$	$\wedge I(14,13)$
16	$\bot$	$\bot I(3,15)$
17	$\neg\alpha$	$\neg I(4,16)$

GAP: ✓
GAPX: ✗

Line	Formula	Justification
1	$\neg(\alpha \wedge \beta)$	given
2	$\neg(\alpha \wedge \gamma)$	given
3	$\neg(\alpha \wedge \neg\beta \wedge \neg\gamma)$	given
4	$\neg(\alpha \wedge \neg\beta)$	given
5	α	hypothesis
6	γ	hypothesis
7	$\alpha \wedge \gamma$	$\wedge I(5,6)$
8	$\bot$	$\bot I(2,7)$
9	$\neg\gamma$	$\neg I(6,8)$
10	β	hypothesis
11	$\alpha \wedge \beta$	$\wedge I(4,10)$
12	$\bot$	$\bot I(1,11)$
13	$\neg\beta$	$\neg I(9,12)$
14	$\alpha \wedge \neg\beta$	$\wedge I(4,13)$
15	$\alpha \wedge \neg\beta \wedge \neg\gamma$	$\wedge I(14,8)$
16	$\bot$	$\bot I(3,15)$
17	$\neg\alpha$	$\neg I(4,16)$

GAP: ✓
GAPX: ✓

Figure 6.1: Proofs showing that imposing an ordering on the sibling derivations makes the extension definition results dependent on that ordering

at the time.

The conclusion so far is that by involving sibling hypotheses and imposing an ordering, the different derivations are made dependent on each other and their relative position. By changing the ordering of sibling derivations, GAPX can give different answers. This may be considered unintuitive, so another definition will be given later. For now, call this definition "Human GAP" or "GAPH". As seen from the example above, GAPH "remembers" past conversations (argument branches/siblings) and can always bring them up if they have been discussed already. This might be considered closer to human nature, as people tend to refer back to examples they've talked about recently! If they have yet to talk about those examples, then they simply cannot use them (hence the ordering) and the conversation may take a different route.

The next definition of GAPX may seem fairer to siblings because it makes them independent unless they reference each other explicitly. The definition is $T \cup \{\phi_1, ..., \phi_k\} \cup \{\neg\psi_1, ..., \neg\psi_l\} \cup \{\neg\chi_1, ..., \neg\chi_m\} \nvdash_{MRA} \bot$, where χ_i has explicitly been referenced by line number in the current box/derivation, so obviously $\chi_i < \phi$ although that is not very important. So the difference is that siblings not referring to each other are completely independent and do not influence the outcome. The intuition will be given later, but for now the example given in Figure 6.1 will be revisited in Figure 6.2. In this figure, GAPH is the previous definition of GAPX, GAPX is now the latest definition. GAP remains the classic definition given in the technical report.

In the first proof, the classic GAP says that $[\gamma ... \bot]$ follows the property as it ignores completely the sibling derivation and the presence of $\neg\beta$ (as it follows the strict version of natural deduction that forbids shortcuts). GAPX says that $[\beta ... \bot]$ had nothing to do with $[\gamma ... \bot]$ so it should stay independent of it. In other words, the sibling derivation was not referenced, so it should bear no effect on the result. The second proof probably needs no explanation. For the third proof, the difference (between that and the first) is that

1	$\neg(\alpha \wedge \beta)$	given
2	$\neg(\alpha \wedge \gamma)$	given
3	$\neg(\alpha \wedge \neg\beta \wedge \neg\gamma)$	given
4	$\neg(\alpha \wedge \neg\beta)$	given
5	α	hypothesis
6	β	hypothesis
7	$\alpha \wedge \beta$	$\wedge I(5,6)$
8	$\bot$	$\bot I(1,7)$
9	$\neg\beta$	$\neg I(6,8)$
10	γ	hypothesis
11	$\alpha \wedge \gamma$	$\wedge I(5,10)$
12	$\bot$	$\bot I(2,11)$
13	$\neg\gamma$	$\neg I(9,12)$
14	$\alpha \wedge \neg\beta$	$\wedge I(4,9)$
15	$\alpha \wedge \neg\beta \wedge \neg\gamma$	$\wedge I(14,13)$
16	$\bot$	$\bot I(3,15)$
17	$\neg\alpha$	$\neg I(5,16)$

GAP: ✓
GAPH: ✗
GAPX: ✓

1	$\neg(\alpha \wedge \beta)$	given
2	$\neg(\alpha \wedge \gamma)$	given
3	$\neg(\alpha \wedge \neg\beta \wedge \neg\gamma)$	given
4	$\neg(\alpha \wedge \neg\beta)$	given
5	α	hypothesis
6	γ	hypothesis
7	$\alpha \wedge \gamma$	$\wedge I(5,6)$
8	$\bot$	$\bot I(2,7)$
9	$\neg\gamma$	$\neg I(6,8)$
10	β	hypothesis
11	$\alpha \wedge \beta$	$\wedge I(5,10)$
12	$\bot$	$\bot I(1,11)$
13	$\neg\beta$	$\neg I(9,12)$
14	$\alpha \wedge \neg\beta$	$\wedge I(4,13)$
15	$\alpha \wedge \neg\beta \wedge \neg\gamma$	$\wedge I(14,9)$
16	$\bot$	$\bot I(3,15)$
17	$\neg\alpha$	$\neg I(5,16)$

GAP: ✓
GAPH: ✓
GAPX: ✓

1	$\neg(\alpha \wedge \beta)$	given
2	$\neg(\alpha \wedge \gamma)$	given
3	$\neg(\alpha \wedge \neg\beta \wedge \neg\gamma)$	given
4	$\neg(\alpha \wedge \neg\beta)$	given
5	α	hypothesis
6	β	hypothesis
7	$\alpha \wedge \beta$	$\wedge I(5,6)$
8	$\bot$	$\bot I(1,7)$
9	$\neg\beta$	$\neg I(6,8)$
10	γ	hypothesis
11	$\alpha \wedge \neg\beta$	$\wedge I(5,8)$
12	$\bot$	$\bot I(4,11)$
13	$\neg\gamma$	$\neg I(9,12)$
14	$\alpha \wedge \neg\beta$	$\wedge I(4,9)$
15	$\alpha \wedge \neg\beta \wedge \neg\gamma$	$\wedge I(14,13)$
16	$\bot$	$\bot I(3,15)$
17	$\neg\alpha$	$\neg I(5,16)$

GAP: undefined
GAPH: ✗
GAPX: ✗

Figure 6.2: Comparisson of different candidate definitions for extending the Genuine Absurdity Property

the $[\beta ... \bot]$ derivation does indeed get involved in the $[\gamma ... \bot]$ derivation, providing a way to prove contradiction without using the hypothesis. Hence the check now for GAPX becomes $T \cup \{\alpha\} \cup \{\} \cup \{\neg\beta\} \nvdash_{MRA} \bot$ which fails. For the first proof, the check for GAPX was $T \cup \{\alpha\} \cup \{\} \cup \{\} \nvdash_{MRA} \bot$ because there was no cross-referencing.

So what is the intuition here? The intuition is that classic GAP does not take siblings into account because it works on a natural deduction that forbids the use of the substitution rule. If a sibling were to be used, it had to be copied in the current derivation and act as a child derivation. The natural deduction style used in this paper simulates copying of sibling derivations by allowing their referencing. Thus referencing a sibling derivation really makes it a child (under the pure natural deduction sense). Thus for GAPX, the $\{\neg\chi_1, ..., \neg\chi_m\}$ set is an extended child set for the derivation/box under examination, that includes all sibling derivations (nothing more, nothing less) that would otherwise be child derivations (and in the $\{\neg\psi_1, ..., \neg\psi_l\}$ set) if the restricted natural deduction was employed. In other words, GAPX is a convenient workaround that fakes copying sibling derivations as child derivations, allowing the user to work in a more expressive and concise environment. Sibling derivations that are not referenced are the derivations that would not have appeared as children in the classic GAP.

Having said that, there is one slight adjustment that needs to be made. So far, only direct sibling derivations were considered, that is, boxes next to the box under consideration. How about referencing an ancestor's sibling conclusion? The natural deduction style used here allows this kind of referencing. Consider the figure shown in Figure 6.3:

1	$\neg(\alpha \wedge \beta)$	given
2	$\neg(\alpha \wedge \gamma \wedge \neg\delta)$	given
3	$\neg(\alpha \wedge \neg\beta \wedge \neg\gamma)$	given
4	$\neg(\delta \wedge \neg\beta)$	given
5	α	hypothesis
6	β	hypothesis
7	$\alpha \wedge \beta$	$\wedge I(5,6)$
8	$\bot$	$\bot I(1,7)$
9	$\neg\beta$	$\neg I(6,8)$
10	γ	hypothesis
11	δ	hypothesis
12	$\delta \wedge \neg\beta$	$\wedge I(11,9)$
13	$\bot$	$\bot I(4,12)$
14	$\neg\delta$	$\neg I(11,13)$
15	$\alpha \wedge \gamma$	$\wedge I(5,10)$
16	$\alpha \wedge \gamma \wedge \neg\delta$	$\wedge I(15,14)$
17	$\bot$	$\bot I(2,16)$
18	$\neg\gamma$	$\neg I(10,17)$
19	$\alpha \wedge \neg\beta$	$\wedge I(4,9)$
20	$\alpha \wedge \neg\beta \wedge \neg\gamma$	$\wedge I(19,18)$
21	$\bot$	$\bot I(3,21)$
22	$\neg\alpha$	$\neg I(5,21)$

Figure 6.3: Referencing of an ancestor's sibling (uncle) derivation

It can be seen that $[\beta ... \bot]$ is not a sibling to $[\delta ... \bot]$ but rather an ancestor's sibling (or uncle) derivation. In this case, current GAPX does not apply. It can therefore simply

extend the $\{\neg\chi_1...\neg\chi_m\}$ set to contain any sibling derivation (that came before) of any ancestor up the "tree" that has been referenced, as once again, such derivation would be copied as a child derivation if it were a strict natural deduction proof that forbids shortcuts.

6.2 Definition of Extension

The previous section showed the thinking behind the arrival at the final definition for the Genuine Absurdity Property extension. This definition is given below:

A certain (sub)derivation $[\phi...\bot]$ follows the extended Genuine Absurdity Property if $T \cup \{\phi_1, ..., \phi_k\} \cup \{\neg\psi_1, ..., \neg\psi_l\} \cup \{\neg\chi_1, ..., \neg\chi_m\} \nvdash_{MRA} \bot$, where χ_i is any sibling or any ancestor's sibling derivation hypothesis whose conclusion is referenced by the (sub)derivation currently examined. Child derivations must also follow this property.

6.3 Correctness of Extension

Take an arbitrary (sub)derivation in the proof under examination. It can be categorized as either making no references to siblings, or indeed referring to sibling derivations.

6.3.1 Case 1: Not Referencing Sibling Derivations

If the derivation makes no references to siblings then, the classic Genuine Absurdity Property is defined as $T \cup \{\phi_1, ..., \phi_k\} \cup \{\neg\psi_1, ..., \neg\psi_l\} \nvdash_{MRA} \bot$. The extension is defined as $T \cup \{\phi_1, ..., \phi_k\} \cup \{\neg\psi_1, ..., \neg\psi_l\} \cup \{\} \nvdash_{MRA} \bot$. This is because the χ-set is empty, since in this case there are no references to siblings.

The two definitions/checks match and so are equivalent for this case.

6.3.2 Case 2: Referencing Sibling Derivations

If the derivation makes references to siblings, then the extension is defined as $T \cup \{\phi_1, ..., \phi_k\} \cup \{\neg\psi_1, ..., \neg\psi_l\} \cup \{\neg\chi_1, ...\neg\chi_m\} \nvdash_{MRA} \bot$. The classic definition is not defined as the box makes use of a shortcut, but take the equivalent strict natural deduction proof where siblings are re-proven inside this derivation as children $\neg\psi_{l+1}, ..., \neg\psi_{l+1+m}$. The classic definition is then defined as $T \cup \{\phi_1, ..., \phi_k\} \cup \{\neg\psi_1, ..., \neg\psi_{l+1+m}\} \nvdash_{MRA} \bot$ or $T \cup \{\phi_1, ..., \phi_k\} \cup \{\neg\psi_1, ..., \neg\psi_l\} \cup \{\neg\psi_{l+1}, ...\neg\psi_m\} \nvdash_{MRA} \bot$ if we break the child-set into two sets containing the old children and the new. But $\{\neg\psi_{l+1}, ...\neg\psi_m\} = \{\neg\chi_1, ...\neg\chi_m\}$ since the former set contains all the siblings that were now included as further children in the strict natural deduction equivalent.

The two definitions/checks are ultimately the same for this case as well.

6.3.3 Effects of More Specific Context of Implicitly Copied Siblings

Copying a sibling derivation does not change the derivation itself, and does not affect its Genuine Absurdity Property status either. That is to say, when a sub-derivation is copied to a new (more specific) context, then it exhibits the Genuine Absurdity Property if it did where it was located initially.

Despite the addition of new context, namely the new ancestor hypotheses that surround the copy of the sub-derivation, the sub-derivation retains its property status because of the definition of the Genuine Absurdity Property. The property definition says that the check is $T \cup \{\phi_1, ..., \phi_k\} \cup \{\neg\psi_1, ..., \neg\psi_l\} \nvdash_{MRA} \bot$ where $\{\phi_1, ..., \phi_k\}$ refers to only the ancestor

hypotheses references (implicitly) copied by the sub-derivation. Since the referenced sub-derivation existed in a more general context, it can not make use of the new (more specific) context it has been copied into, and therefore the same check will still be performed if the Genuine Absurdity Property is checked again, resulting in the same status as the original sub-derivation in its original context.

This perhaps is easier to show using an example. Consider again the proof in Figure 6.3. The sub-derivation $[\delta ... \bot]$ implicitly copies the sub-derivation $[\beta ... \bot]$ into its own context. The context of the copied sub-derivation in its original location contains ancestor hypothesis α only. The context in the imported location (inside sub-derivation $[\delta ... \bot]$) includes ancestor hypotheses $\{\alpha, \gamma, \delta\}$. The new additions (ie the more specific context) $\{\gamma, \delta\}$ will not affect whether the sub-derivation $[\beta ... \bot]$ follows the Genuine Absurdity Property in its new environment because it is oblivious to that new context as it was imported from a more general one. Thus its Genuine Absurdity Property check will still be $T \cup \{\alpha\} \cup \{\} \nvdash_{MRA} \bot$ in either context, yielding the same results. The same would apply to its children sub-derivations if it had any.

6.3.4 Assumption of Proof Sketch

This proof assumes that an equivalent proof can always be generated that does not make use of any substitutions. Removing a substitution from a (sub)derivation can always be done by replacing the reference to the other derivation by one that points to a local copy of that derivation. The generated proof is always a valid natural deduction proof; however, a proof of this will not be given in this paper.

6.4 Details of Implementation

Compared to the classic implementation of the Genuine Absurdity Property, the extension implementation remains largely the same. The differences lie in that the initial checks do not include the use of substitution check (since shortcuts are allowed here) and that intermediate steps may also reference external sibling hypotheses.

```
1 % Checks that the given proof follows the extended GAP property
2 checkGAPX(Proof) :-
3         reverse(Proof, RevProof),
4         checkRAND(RevProof), !,
5         checkRestrictedRules(RevProof), !,
6         getTheoryAndRevBox(RevProof, Theory, RevBox), !,
7         checkRestrictedTheory(Theory), !,
8         checkGAPX(Theory, _, [], [], [], RevBox, RevBox, _), !.
```

Listing 6.1: Checking whether a proof is follows the extended Genuine Absurdity Property

Listing 6.1 shows that the implementation is comparable to the implementation of the original definition (compare with Listing 5.1 and Listing 5.5). As mentioned in the previous chapter the code checking for the Genuine Absurdity Property (in Listing 5.5) scans the line references of intermediate steps in the current derivation to find references to sibling derivations. The scanning and gathering of referenced siblings is done by `getUsedSiblingHypotheses/6` which is shown again in Listing 6.2.

```
1 % Uses the line references to find referenced sibling derivations
2 getUsedSiblingHypotheses(_, [], _, _, []) :- !.
3 getUsedSiblingHypotheses(Theory, Reason, WholeProof, HL, NewHypotheses) :-
```

```
4         length(Theory, L),
5         findall(R, (m2(R, Reason), R < HL, R >= L), External),
6         findall([H, E], (m2(E, External), getStep(E, WholeProof, step(H,
              [notI|_], _))), HLN),
7         unzip(Hs, LNs, HLN),
8         subtract(External, LNs, Rest),
9         findall(Reason2, (m2(R2, Rest), getStep(R2, WholeProof, step(_,
              [_|Reason2], _))), Reasons),
10        append(Reasons, Reasons2),
11        getUsedSiblingHypotheses(Theory, Reasons2, WholeProof, HL,
              NewHypotheses2),
12        append(Hs, NewHypotheses2, NewHypotheses).
13
14 unzip([], [], []).
15 unzip([L|Ls], [R|Rs], [[L,R]|Ps]) :- unzip(Ls, Rs, Ps).
```

Listing 6.2: Gathering of referenced sibling derivations for the extended Genuine Absurdity Property definition

The predicate `getUsedSiblingHypotheses/6` works much in the same way as it works for the original definition of the Genuine Absurdity Property, with the only difference being that line 6 of the code may actually yield results. Sibling hypotheses are more or less treated the same way as ancestor hypotheses, in that they are dug up in this predicate, given back to `checkGAP/7`, carried over until the sub-derivation under check is consumed and then merged with the other (ancestor and child) hypotheses in order to perform the provability check.

6.5 Remarks

The extension to the original Genuine Absurdity Property definition considers only consistent theories and proofs that use conjunction and negation only. It may not necessarily be the case that the extension works for more general proofs. Testing its correctness for inconsistent proofs or when using more connectors (and subsequent amendments in case it proves not to be correct) remains future work.

Consider again the proof in Figure 6.3. The reason the sub-derivation $[\beta ... \bot]$ can be used in $[\delta ... \bot]$ is because it came before (in a more general context) it. Thus in an iterative implementation of the Genuine Absurdity Property checker, $[\beta ... \bot]$ will already be checked if it followed the property before considering checking $[\delta ... \bot]$. As an optimization checking for the Genuine Absurdity Property for sibling derivations can be avoided, as it remains constant (see subsection 6.3.3) for all of their implicit copies. This optimization is performed by the extended Genuine Absurdity Property checker.

Chapter 7

Visualization of Genuine Absurdity Property Proofs

The Argumentation Logic paper does not provide a formal algorithm for visualizing proofs but rather the idea that proofs having the Genuine Absurdity Property can be seen from an argumentative point of view and any abstract argumentation framework can be visualized as shown previously in subsection 2.1.7, where an example was given by Figure 2.1. This chapter documents the attempt to provide an algorithm for the visualization of proofs that both follow the original Genuine Absurdity Property or the extension given in chapter 6.

This chapter covers step 4 of the original planned steps. The reverse procedure, in step 4+, as well as the argument builder, in step 4++, are discussed in chapter 8 and chapter 12 respectively.

7.1 Assumptions Made by Algorithm

The algorithm provided here assumes that the proofs supplied all follow the Genuine Absurdity Property. This property ensures that arguments made in the (corresponding argumentation framework of the) proof are relevant. It might be possible to visualize proofs that do not have this property. The algorithm can deal with proofs that follow the extended version of the property as well.

A consequence of this assumption is that the formulas in the proofs consist of conjunctions and negations only, as the Genuine Absurdity Property is defined only for proofs consisting of those constructs.

7.2 Description of Algorithm

In a nutshell, the algorithm moves around the proof in a backwards, $\neg I$-rule's-contradiction-driven approach. It assumes that the hypothesis in the outer box forms the initial argument. At the end of the box, where the contradiction is established, clues can be found as to what the attacks were. Recall from subsection 2.2.1 that the $\bot I$ rule points to two formulas in the proof of the form A and $\neg A$. Formula A will be either a conjunction of smaller formulas or just one atom. The negated formulas or (negated) atoms form the attack. Naturally, the defenses are the negations of each of those formulas or atoms which will be hypotheses inside boxes which lead to more contradictions recursively. The algorithm repeats itself recursively, until the attacks gathered from the contradictions only form part of the theory or previous hypotheses (defenses higher up the chain).

1	$\neg(\beta \wedge \alpha)$	given
2	$\neg(\neg\beta \wedge \gamma)$	given
3	$\neg(\alpha \wedge \neg\beta \wedge \neg\gamma)$	given
4	α	hypothesis
5	β	hypothesis
6	$\beta \wedge \alpha$	$\wedge I(5,4)$
7	$\bot$	$\bot I(1,6)$
8	$\neg\beta$	$\neg I(5,7)$
9	γ	hypothesis
10	$\neg\beta \wedge \gamma$	$\wedge I(8,9)$
11	$\bot$	$\bot I(2,10)$
12	$\neg\gamma$	$\neg I(9,11)$
13	$\alpha \wedge \neg\beta$	$\wedge I(4,8)$
14	$\alpha \wedge \neg\beta \wedge \neg\gamma$	$\wedge I(13,12)$
15	$\bot$	$\bot I(3,14)$
16	$\neg\alpha$	$\neg I(4,15)$

Figure 7.1: Example proof for the visualization algorithm

The algorithm's function will now be demonstrated with the aid of an example. Consider the following proof:

The initial argument will be found at the top of the outer-most box - the hypothesis α. The attacks can be found from the contradiction at the bottom, from line 15. The reasons from the contradiction are lines 3 and 14, which are of the form $\neg A$ and A respectively, where $A = \alpha \wedge \neg\beta\neg\gamma$. A is a conjunction of (negated) atoms, and so the attack against α will be $\{\neg\beta, \neg\gamma\}$. Note that α in A was ignored because it is the argument being attacked itself. The attacks were derived at lines 8 and 12. This is known because the conjunction A is accompanied by the line numbers of its constituents. The defense against $\neg\beta$ and $\neg\gamma$ can be found in the two respective sub-derivations where their negations are assumed.

β is the defense against $\neg\beta$, and at the bottom of the first inner box the attack(s) against it can be found. The conjunction consists of itself and α, which is a previous defense. So, after ignoring the hypothesis itself, α remains as the attack. Because this attack was used previously as a defense, it makes no sense to try and defend against it. Thus this part of the algorithm ends here.

γ is the defense against $\neg\gamma$, and at the bottom of the second inner box the attack(s) against it can be found. The conjunction here consists of itself and $\neg\beta$. The defense against it and the attack against that can be found in the first inner box and so this part of the algorithm repeats as before. Note that the use of $\neg\beta$ refers to the sibling derivation $[\beta ... \bot]$. This is normally forbidden by the classic definition of the Genuine Absurdity Property but allowed by the extension. The algorithm finds the defense against $\neg\beta$ regardless of its relative position as it is indicated by the line reference in the justification of the attack.

At the end, the following abstract argumentation framework is extracted, which is drawn in Figure 7.2:

- arguments: $\{\{\alpha\}, \{\neg\beta, \neg\gamma\}, \{\beta\}, \{\neg\beta\}, \{\gamma\}\}$
- attacks: $\{(\{\neg\beta, \neg\gamma\}, \{\alpha\}), (\{\beta\}, \{\neg\beta, \neg\gamma\}), (\{\gamma\}, \{\neg\beta, \neg\gamma\}), (\{\neg\beta\}, \{\gamma\}), (\{\beta\}, \{\neg\beta\})\}$

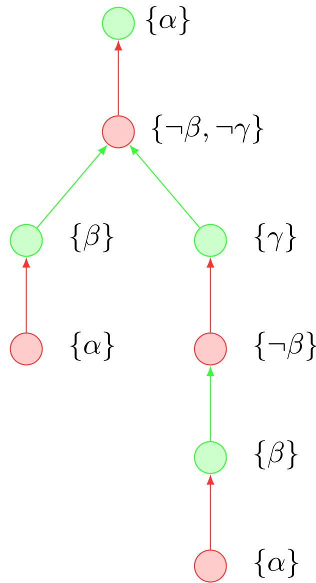

Figure 7.2: Visualization of the proof of example in Figure 7.1

7.3 Observations and Remarks

It may be obvious by now that a certain mapping can be established between the hypotheses and contradictions and defenses and attacks. A hypothesis in the proof always represents a defense from the argumentation point of view, and the individual atoms, negated atoms and negated subformulas whose conjunction (along with the negation of their conjunction) forms the contradiction represent the attack from the argumentation point of view. Some of these atoms, negated atoms and formulas can be traced back to conclusions of $\neg I$ rules, thus it can be said that conclusions of $\neg I$ rules (ie the results of the boxes) form some part of an attack.

When using this algorithm, there is always one attack per argument, but there can be many defenses. Each defense is the negation of an atom of an attack (argument). Both the attacks and defenses follow their respective definitions as described in the original paper and in section 2.3.2 (Argumentation Logic Framework Definition and Defense Against an Attack).

When considering the formulas of the conjunction that constitutes an attack, all formulas that are part of the theory or the argument under attack are ignored. Absence of any remaining formulas signifies an attack by the empty set. This is shown by the example in Figure 7.5.

Although the argumentation framework shown in Figure 7.2 could have been drawn so that argument $\{\beta\}$ on the left side of the ramification attacks $\{\neg\beta\}$ on the right, the algorithm repeats the whole branch in the drawing. This choice has a couple of implications. Firstly the generated diagram is always a tree, as chains of arguments are "expanded" or "ironed out" regardless of whether natural deduction proofs made use of sibling $\neg I$ applications. Take the example given in Figure 7.1. The second inner box makes use of the conclusion $\neg\beta$ that was derived from the first inner box in order to reduce redundancy. The algorithm turns this proof into a framework where the first inner box is essentially cloned (duplicated) into the second box. Because of this "expansion", the visualization algorithm does not distinguish between the proof in Figure 7.1 and the redundant proof in Figure 7.3; both proofs when visualized would give the result shown in Figure 7.2.

The algorithm behaves in accordance to the Argumentation Logic technical report. Using a sibling derivation can be considered a shortcut in natural deduction which is used to avoid repeating the same part of the proof twice. The Argumentation Logic paper uses natural deduction in its "pure" form, where this shortcut is not allowed, and proofs where the conclusion of a sibling derivation is used in another derivation do not exist as such. This is the reason why the definition of the Genuine Absurdity Property (section 2.3.4) does not take into account sibling derivations. For flexibility purposes, the algorithm "irons out" proofs that do make use of sibling derivations instead of rejecting them, since they do follow the extended Genuine Absurdity Property definition that allows the use of shortcuts. Essentially, what the algorithm does, is to remove the shortcuts and produce an argument that would correspond to the corresponding proof that does not make use of shortcuts. Therefore regardless of whether shortcuts were used or not, the argument retains the same logical form.

The algorithm could be made instead to draw an attack from $\{\beta\}$ on the left branch to $\{\gamma\}$ on the right. This would remove the redundancy and produce different visualizations for the different proofs in Figure 7.1 and Figure 7.3. The resulting visualizations however would not necessarily be trees. An interesting topic is the detection of similar subtrees in argument attack/defense chains and their pruning, resulting in less redundant and more concise proofs when the arguments are converted back into proofs. In addition, attacks and defenses could be moved to shallower levels in the tree if possible, in order for their corresponding boxes in the proof to appear in a shallower box-nesting level, so that their conclusions can be used in a wider context. Of course, this pruning and relocating of nodes in the tree must be done carefully, so that the resulting proofs always follow the Genuine Absurdity Property. This topic remains part of future work and is not discussed further in the report.

7.4 Details of Implementation

The algorithm first starts by reversing the proof so that it is in ascending order (opposite of the output by the theorem prover in chapter 4). Listing 7.1 shows that after reversing the proof, the algorithm calls the `getDefence/7` predicate, which is responsible for producing the defense node as well as handling its subtree (which on the top level is the entire tree).

The `getDefence/7` (line 7 of the code) predicate adds the hypothesis of the box in the argumentation framework under construction and then calls `getAttack/6` which returns the attack and its subtree.

The `getAttack/6` (line 10 of the code) looks at the bottom of the box and finds the contradiction that will indicate what the attacks are (lines 11 to 16). If the contradiction is due to a conjunction of formulas then the individual components are found (line 19). If there is only one component (there is no conjunction) it is retrieved by line 22. If the contradiction was reached just by using the hypothesis itself, then the attack has no special components which makes it an empty set attack (line 25). A node is then made from the components for the argumentation framework that is being built (line 27) and then all components that were attempted to defend against are gathered. For each of those components, their respective subtrees are gathered (line 31) and finally all the new nodes and attacks between them are put together and returned (lines 32, 33). The predicate `m2/2` on lines 18, 21, 24 and 29 acts as an alias to the standard `member/2` Prolog predicate.

```
1 % Converts a GAP proof to its argumentation representation
2 convertGAPToArg(Proof, [Nodes, AttDefs]) :-
3       reverse(Proof, RevProof),
```

1	$\neg(\beta \wedge \alpha)$	given
2	$\neg(\neg\beta \wedge \gamma)$	given
3	$\neg(\alpha \wedge \neg\beta \wedge \neg\gamma)$	given
4	α	hypothesis
5	β	hypothesis
6	$\beta \wedge \alpha$	$\wedge I(5,4)$
7	$\bot$	$\bot I(1,6)$
8	$\neg\beta$	$\neg I(5,7)$
9	γ	hypothesis
10	β	hypothesis
11	$\beta \wedge \alpha$	$\wedge I(10,4)$
12	$\bot$	$\bot I(1,11)$
13	$\neg\beta$	$\neg I(10,12)$
14	$\neg\beta \wedge \gamma$	$\wedge I(13,9)$
15	$\bot$	$\bot I(2,14)$
16	$\neg\gamma$	$\neg I(9,15)$
17	$\alpha \wedge \neg\beta$	$\wedge I(4,8)$
18	$\alpha \wedge \neg\beta \wedge \neg\gamma$	$\wedge I(17,16)$
19	$\bot$	$\bot I(3,18)$
20	$\neg\alpha$	$\neg I(4,19)$

Figure 7.3: Example another proof that results in a framework like in Figure 7.2. This proof is redundant but correct under the rules of natural deduction nevertheless.

```
4         getTheoryAndRevBox(RevProof, Theory, Box),
5         getDefence(Box, Box, Theory, 1, 0, Nodes, [_|AttDefs]), !.
6 % Makes a node and defence relation against given attack (and handles its subtree
      as well)
7 getDefence([step(Hypothesis, [hypothesis], _)|Proof], WholeProof, Theory, You,
      Target, [[[Hypothesis], You]|Nodes], [[You, Target]|AttDefs]) :-
8         getAttack(Proof, WholeProof, [Hypothesis|Theory], You, Nodes, AttDefs).
9 % Makes a node and attack relation against given defence (and handles its subtree
      as well)
10 getAttack(Proof, WholeProof, Ignore, N, Nodes, AttDefs) :-
11        reverse(Proof, RevProof),
12        (
13                RevProof = [step(falsity, [falsityI, _, LN], _)|_];
14
15                RevProof = [step(falsity, [check, _], _), step(falsity, [falsityI,
                      _, LN], _)|_]
16        ),
17        (
18                m2(step(_, [andI|_], LN), Proof),
19                findAttackComponents(Proof, WholeProof, Ignore, LN, Components), !;
20
21                m2(step(X, _, LN), Proof),
22                getAttackComponent(X, LN, Ignore, Proof, WholeProof, Components),
                      !;
23
24                not(m2(step(_, _, LN), Proof)),
25                Components = []
```

```
26          ),
27          makeAttackNode(Components, Attack),
28          Ignore = [_|Theory],
29          findall([AttComponent, DefAgainstAtt], m2([AttComponent, DefAgainstAtt],
                Components), DefendedAgainstComponents),
30          ln(N, NextN), ln(NextN, NextNextN),
31          getDefences(WholeProof, Theory, NextNextN, NextN,
                DefendedAgainstComponents, DNodes, DAttDefs),
32          Nodes = [[Attack, NextN]|DNodes],
33          AttDefs = [[NextN, N]|DAttDefs].
```

Listing 7.1: First part of the proof visualization algorithm

The predicate `getDefences/7` as shown in Listing 7.2 is responsible for gathering all defenses and their subtrees. For each defense, it calls `getDefence/7` and it ensures that the nodes returned have different labels from the other defenses. The nodes and attack relations are then merged and returned. The predicate `a2/3` on lines 7 and 8 acts as an alias to the standard `append/3` Prolog predicate.

```
1   % Gathers all defences against an attack (and their subtrees)
2   getDefences(WholeProof, Theory, You, Target, [[_, Box]|Components], Nodes,
        AttDefs) :-
3           getDefence(Box, WholeProof, Theory, You, Target, DNodes, DAttDefs),
4           reverse(DNodes, [[_, LastId]|_]),
5           ln(LastId, NewYou),
6           getDefences(WholeProof, Theory, NewYou, Target, Components, DsNodes,
                DsAttDefs),
7           a2(DNodes, DsNodes, Nodes),
8           a2(DAttDefs, DsAttDefs, AttDefs).
9   getDefences(_, _, _, _, [], [], []).
```

Listing 7.2: Second part of the proof visualization algorithm

The predicates `findAttackComponents/5` and `getAttackComponent/6` as shown in Listing 7.3 work closely together in order to breakdown the conjunction of attack components. Conjunctions are gradually broken down into individual smaller parts (lines 9-15), until all that remains is atoms, negated atoms and negated formulas. If the attack component is part of the theory or the current hypothesis, it is ignored (line 17). If, on the other hand, it is a previous hypothesis (in other words a previous defense used earlier), then it is added as a terminal attack (an attack that cannot be defended against) (line 18). Alternatively, the attack component forms an attack that was attempted to defend against, and is returned along with the box containing the defense attempt (line 19). The individual components are then placed in one list and are returned (line 13). The predicate `m2/2` on lines 10 and 17 acts as an alias to the standard `member/2` Prolog predicate. The predicate `a2/3` on line 13 acts as an alias to the standard `append/3` Prolog predicate.

```
1   % Breaks down an attack into individual components and the defence attempts
        against them
2   % returns a list of a mixture of:
3   % [A] (for part of the attack used as defences up the tree - there's no way to
        attack them)
4   % [A, defAgainstA] (for part of the attack that was attempted to defend against,
        plus the box of the proof that does so)
5   % example: given attack a&b&c&d returns [[a], [[b], [...]], [[c], [...]]],
6   % assuming a was used as a defence above (you cannot defend against your defence,
```

```
    so this is a terminal part of the attack),
7 % there was an attempt to defend against b and c (given by the parts of the proof
    [...]),
8 % and d was part of the theory hence there's no defence against that
9 findAttackComponents(Proof, WholeProof, Ignore, LN, Components) :-
10       m2(step(and(A, B), [andI, LN1, LN2], LN), Proof),
11       getAttackComponent(A, LN1, Ignore, Proof, WholeProof, Components1),
12       getAttackComponent(B, LN2, Ignore, Proof, WholeProof, Components2),
13       a2(Components1, Components2, Components).
14 getAttackComponent(and(_, _), LN, Ignore, Proof, WholeProof, Components) :-
15       !, findAttackComponents(Proof, WholeProof, Ignore, LN, Components).
16 getAttackComponent(A, LN, Ignore, _, WholeProof, Component) :-
17       m2(A, Ignore), Component = [], !;
18       getStep(LN, WholeProof, step(A, [hypothesis], LN)), Component = [[A]], !;
19       getBox(LN, WholeProof, _, Box), Component = [[A, Box]].
```

Listing 7.3: Third part of the proof visualization algorithm

7.5 Example Walkthough

The example will revolve around the example proof given in section 7.2. The algorithm changes the format of the proof so that steps are in increasing order, in the same way as the proof would be printed on paper. The outer box is selected and `getDefence/7` is called upon to return the entire tree (lines 3-5 of Listing 7.3).

The predicate `getDefence/7` automatically creates and adds a node for the defense α, the hypothesis of the outer box. It then assigns the duty of creating the rest of the tree to `getAttack/6` (line 8). Note that it adds the hypothesis to the ignore list.

This predicate looks at the bottom of the proof (line 15 of the proof, line 13 of the code) and then picks up the conjunction of components $\alpha \wedge \neg\beta \wedge \neg\gamma$ (line 14 of the proof, line 18 of the code). The predicate `findAttackComponents/5` is called to find and return the attack components.

`findAttackComponents/5` splits the conjunction in two parts, namely α and $\neg\beta \wedge \neg\gamma$ (line 10 of Listing 7.3). `getAttackComponent/6` is called on both parts.

For the first call, `getAttackComponent/6` realizes that the component α is an atom, and that it is included in the ignore list. Thus it returns empty-handed (line 17 of the code).

For the second call, `getAttackComponent/6` realizes that the component is actually another conjunction and calls `findAttackComponents/5` to split the conjunction into individual parts (line 15). The two parts are $\neg\beta$ and $\neg\gamma$. `getAttackComponent/6` is called on both parts.

`getAttackComponent/6` realizes that $\neg\beta$ is a negated atom and is not e member of the ignore list nor it is a previous defense. This means that an attempt to defend against it must have been done. Thus the component itself, along with its accompanying attempted defense box (lines 5-7 in the proof, line 19 in the code) are returned.

Similarly for the case of $\neg\gamma$, the component itself along with its box (lines 9-11 in the proof) are returned. The components find their way back to the first call of `findAttackComponents/5` which returns them in a single list back to `getAttack/6` (line 19 in Listing 7.1).

Continuing on line 27, the attack node $\{\neg\beta, \neg\gamma\}$ is made, and from the components of the attack, all that were attempted to be defended against are gathered (line 29). In this case there were attempts at defending against both $\neg\beta$ and $\neg\gamma$. `getDefences/7` is called to return the subtrees of the two attack components.

`getDefences/7` is a recursive predicate that finds the defenses (and the rest of the subtree) for each attack component. It also handles the labels for the subtrees so that they

don't use the same identifiers. Note that the hypothesis α has been dropped from the ignore list.

`getDefence/7` is called in order to find the defense against $\neg\beta$. This is where the algorithm repeats all the steps of the walkthrough so far again, though this time the hypothesis is β. The major difference is that `getAttackComponent/6` will ignore the β attack component (line 6 of the proof) but it will accept α as an attack that was previously used as a defense, making it a terminal component (ie there is no defending against that).

Similarly for $\neg\gamma$, the procedure is repeated and the results bubble back up to `getDefences/7`, where those results are merged with the results of the defense attempt against $\neg\beta$ (lines 7-8 of Listing 7.2). The results that this predicate returns are merged with the attack node $\{\neg\beta, \neg\gamma\}$ from line 27 of Listing 7.1 in order to produce the entire subtree of the attack $\{\neg\beta, \neg\gamma\}$ against α (lines 32-33). The results are then pushed upwards and the end result is the tree shown in Figure 7.2.

7.6 Visualization Examples

This section illustrates the function of the algorithm with more examples. For each example, the proof is given alongside its visual representation according to the visualization algorithm.

1	$\neg(\beta \wedge \alpha)$	given
2	$\neg(\alpha \wedge \gamma)$	given
3	$\neg(\alpha \wedge \neg\beta \wedge \neg\gamma)$	given
4	α	hypothesis
5	β	hypothesis
6	$\beta \wedge \alpha$	$\wedge I(5,4)$
7	$\bot$	$\bot I(1,6)$
8	$\neg\beta$	$\neg I(5,7)$
9	γ	hypothesis
10	$\alpha \wedge \gamma$	$\wedge I(4,9)$
11	$\bot$	$\bot I(2,10)$
12	$\neg\gamma$	$\neg I(9,11)$
13	$\alpha \wedge \neg\beta$	$\wedge I(4,8)$
14	$\alpha \wedge \neg\beta \wedge \neg\gamma$	$\wedge I(13,12)$
15	$\bot$	$\bot I(3,14)$
16	$\neg\alpha$	$\neg I(4,15)$

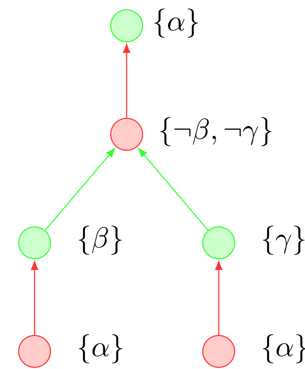

Figure 7.4: Visualization example of 2-level boxes

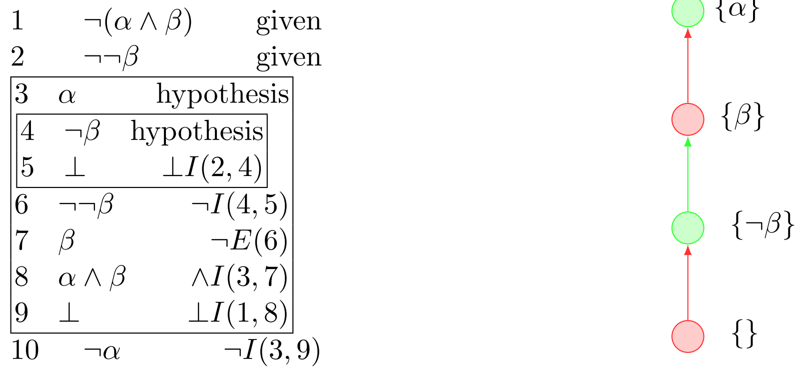

Figure 7.5: Visualization example of empty set attack

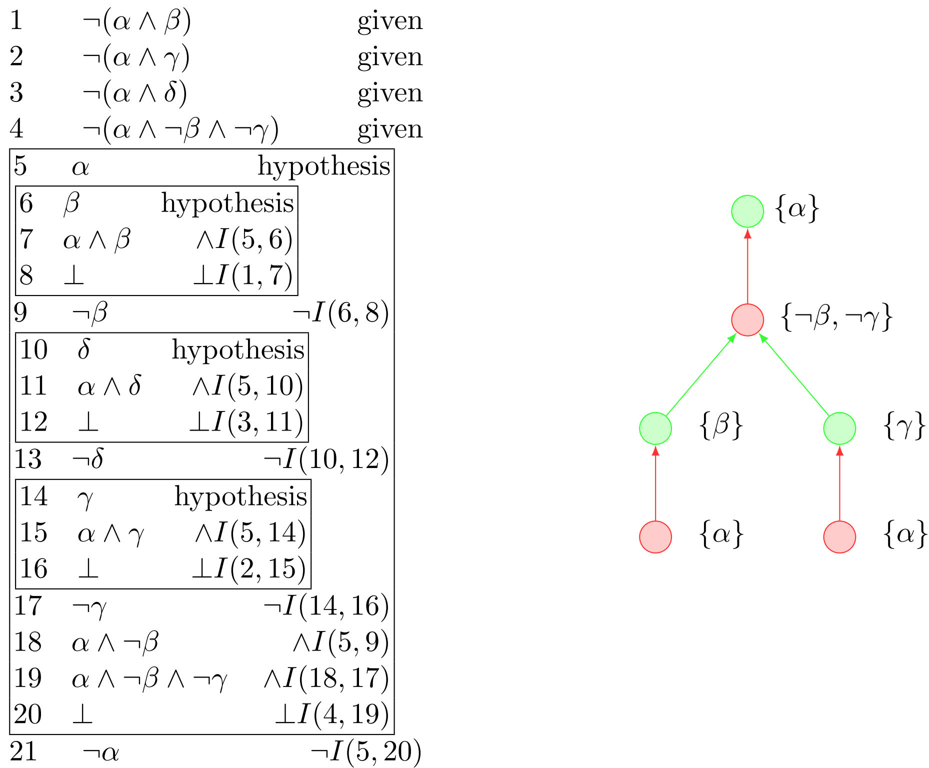

Figure 7.6: Visualization example of ignored successful defense

1	$\neg(\neg\alpha \wedge \delta)$	given
2	$\neg(\alpha \wedge \neg\beta)$	given
3	$\neg(\alpha \wedge \gamma)$	given
4	$\neg(\alpha \wedge \beta \wedge \neg\gamma)$	given
5	δ	hypothesis
6	α	hypothesis
7	$\neg\beta$	hypothesis
8	$\alpha \wedge \neg\beta$	$\wedge I(6,7)$
9	$\bot$	$\bot I(2,8)$
10	β	$\neg I(7,9)$
11	γ	hypothesis
12	$\alpha \wedge \gamma$	$\wedge I(6,11)$
13	$\bot$	$\bot I(3,12)$
14	$\neg\gamma$	$\neg I(11,13)$
15	$\alpha \wedge \beta$	$\wedge I(6,10)$
16	$\alpha \wedge \beta \wedge \neg\gamma$	$\wedge I(15,14)$
17	$\bot$	$\bot I(4,17)$
18	$\neg\alpha$	$\neg I(6,17)$
19	$\neg\alpha \wedge \delta$	$\wedge I(18,5)$
20	$\bot$	$\neg I(1,19)$
21	$\neg\delta$	$\neg I(5,20)$

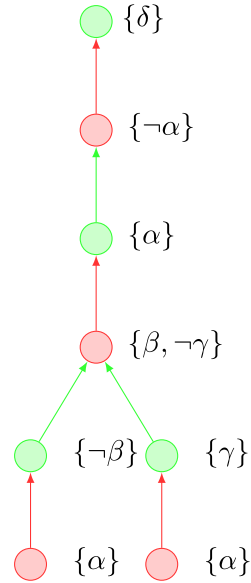

Figure 7.7: Visualization example of 3-level boxes

1	γ	given
2	$\neg(\alpha \wedge \beta \wedge \gamma)$	given
3	$\neg(\alpha \wedge \neg\beta)$	given
4	α	hypothesis
5	$\neg\beta$	hypothesis
6	$\alpha \wedge \neg\beta$	$\wedge I(4,5)$
7	$\bot$	$\bot I(3,6)$
8	$\neg\neg\beta$	$\neg I(5,7)$
9	β	$\neg E(8)$
10	$\alpha \wedge \beta$	$\wedge I(4,9)$
11	$\alpha \wedge \beta \wedge \gamma$	$\wedge I(10,1)$
12	$\bot$	$\bot I(2,11)$
13	$\neg\alpha$	$\neg I(4,12)$

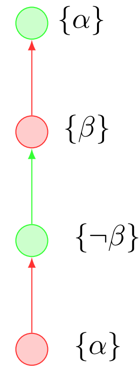

Figure 7.8: Visualization example of theory attack

1	$\neg(\neg\alpha \wedge \delta \wedge \epsilon)$	given
2	$\neg(\alpha \wedge \neg\beta)$	given
3	$\neg(\alpha \wedge \gamma)$	given
4	$\neg(\alpha \wedge \beta \wedge \neg\gamma)$	given
5	$\neg(\neg\delta \wedge \epsilon)$	given
6	$\neg(\zeta \wedge \eta)$	given
7	$\neg(\zeta \wedge \neg\eta \wedge \neg\epsilon)$	given
8	ζ	hypothesis
9	ϵ	hypothesis
10	α	hypothesis
11	$\neg\beta$	hypothesis
12	$\alpha \wedge \neg\beta$	$\wedge I(10, 11)$
13	$\bot$	$\bot I(2, 12)$
14	β	$\neg I(11, 13)$
15	γ	hypothesis
16	$\alpha \wedge \gamma$	$\wedge I(10, 15)$
17	$\bot$	$\bot I(3, 16)$
18	$\neg\gamma$	$\neg I(15, 17)$
19	$\alpha \wedge \beta$	$\wedge I(10, 14)$
20	$\alpha \wedge \beta \wedge \neg\gamma$	$\wedge I(19, 18)$
21	$\bot$	$\bot I(4, 20)$
22	$\neg\alpha$	$\neg I(10, 21)$
23	$\neg\delta$	hypothesis
24	$\neg\delta \wedge \epsilon$	$\wedge I(23, 9)$
25	$\bot$	$\bot I(5, 24)$
26	δ	$\neg I(23, 25)$
27	$\neg\alpha \wedge \delta$	$\wedge I(22, 26)$
28	$\neg\alpha \wedge \delta \wedge \epsilon$	$\wedge I(27, 9)$
29	$\bot$	$\bot I(1, 28)$
30	$\neg\epsilon$	$\neg I(9, 29)$
31	η	hypothesis
32	$\zeta \wedge \eta$	$\wedge I(8, 31)$
33	$\bot$	$\bot I(6, 32)$
34	$\neg\eta$	$\neg I(31, 33)$
35	$\zeta \wedge \neg\eta$	$\wedge I(8, 34)$
36	$\zeta \wedge \neg\eta \wedge \neg\epsilon$	$\wedge I(35, 30)$
37	$\bot$	$\bot I(7, 36)$
38	$\neg\zeta$	$\neg I(8, 37)$

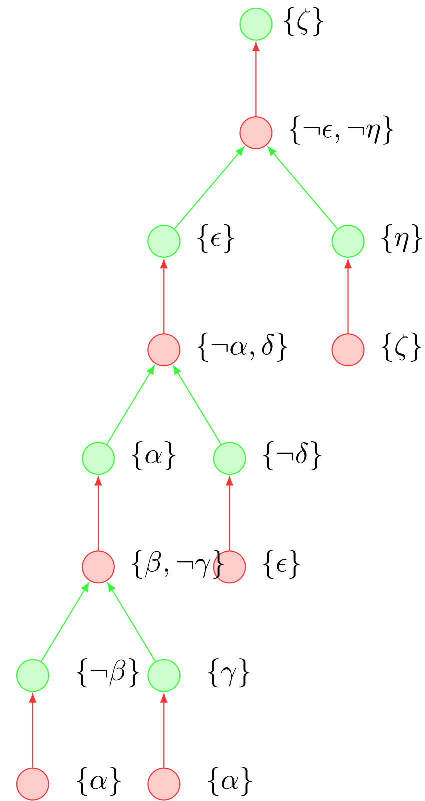

Figure 7.9: Visualization example of 4-level boxes

Chapter 8

Extracting Proofs from Arguments

chapter 7 shows how proofs following the Genuine Absurdity Property can be seen from an argumentative point of view (step 4). An algorithm can be devised to work the opposite way. This chapter discusses how an argument can be used along with the theory it was drawn from to produce a proof that corresponds to this argument. This proof, if visualized again using the algorithm from chapter 7 will produce the same argument used to generate the proof. This covers step 4+, as discussed in section 3.1.

8.1 Assumptions Made by Algorithm

Valid, complete arguments are the primary assumption of this algorithm. This means that the argument has the form of a tree, in the same way the visualization algorithm produces arguments. Odd layers of the tree represent defenses and even layers represent attacks. The definitions used for attacks and defenses are those found in Argumentation Logic Framework Definition and Defense Against an Attack (section 2.3.2). Furthermore, the argument must be about the non-acceptability of the initial hypothesis (argument at the root) as described in section 2.3.3. In other words, the argument should be a proof of $NACC^T(\{a\}, \{\})$, where a is the initial hypothesis posited. It might be possible to visualize incomplete arguments, but this is not guaranteed by the algorithm.

The algorithm assumes that the theory accompanying the argument is expressed in conjunction and negation only. The generated proof will have a conjunction (or a single atom) at the end of each box in the generated proof which will be used alongside its negation (that usually appears in the theory) in order to establish a contradiction. Implicitly, the proofs will be expressed in conjunction and negation only.

8.2 Description of the Algorithm

In a nutshell, the algorithm works by stitching together small proofs that each contains an application of the Reductio ad Absurdum rule. The process starts bottom-up, from the leaves of the argument tree, and builds the proof starting from the inner-most boxes building outwards (to the top-level box).

Consider the example in Figure 8.1. This will be used to describe the workings of the algorithm.

The algorithm works with defense-attack pairs, and as mentioned before, bottom-up. It will choose either the left branch or the right branch (it makes no difference in the final result).

1	$\neg(\beta \wedge \alpha)$	given
2	$\neg(\alpha \wedge \gamma)$	given
3	$\neg(\alpha \wedge \neg\beta \wedge \neg\gamma)$	given
4	α	hypothesis
5	β	hypothesis
6	$\beta \wedge \alpha$	$\wedge I(5,4)$
7	$\bot$	$\bot I(1,6)$
8	$\neg\beta$	$\neg I(5,7)$
9	γ	hypothesis
10	$\alpha \wedge \gamma$	$\wedge I(4,9)$
11	$\bot$	$\bot I(2,10)$
12	$\neg\gamma$	$\neg I(9,11)$
13	$\alpha \wedge \neg\beta$	$\wedge I(4,8)$
14	$\alpha \wedge \neg\beta \wedge \neg\gamma$	$\wedge I(13,12)$
15	$\bot$	$\bot I(3,14)$
16	$\neg\alpha$	$\neg I(4,15)$

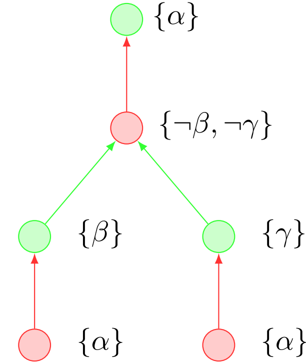

Figure 8.1: Proof extraction example

Assume that the pair $\{\beta\}$ - $\{\alpha\}$ is chosen from the left branch. A $\neg I$ proof will be generated with the ultimate goal being the negation of the defense, $\neg\beta$. Inside the box of the $\neg I$ application, the defense β will be the hypothesis. The generated proof will be such that the contradiction is derived from the attack (in this case $\{\alpha\}$) and the defense. This ensures that if the final proof is visualized again, the attack node (argument) generated will be $\{\alpha\}$. Note that while generating this proof, all parent defenses (in this case $\{\alpha\}$) are temporarily added as givens so that they are available for use. Recall from the mapping discussed in the visualization algorithm (chapter 7) that defenses translate to hypotheses, thus parent defenses (up the argument tree) should be available (as givens) to child proofs. By now, lines 5-8 of the final proof shown in Figure 8.1 have been built.

The same procedure is used to generate another proof for the $\{\gamma\}$ - $\{\alpha\}$ pair on the right branch. The result is a Reductio ad Absurdum proof shown on lines 9-12 in Figure 8.1. During the generation of this proof, α is again accessible as a given.

The algorithm takes the final pair $\{\alpha\}$ - $\{\neg\beta, \neg\gamma\}$. Once more, it strives to create a proof with the end-goal being the negation of the defense, $\neg\alpha$, with a $\neg I$ application with hypothesis the defense α and the contradiction given by the attack $\{\neg\beta, \neg\gamma\}$. In this case, there are no parent defenses so the only givens are the theory. There are however child hypotheses available and this is the main reason as to why the algorithm is bottom-up. When opening the box for the $\neg I$ application, the two sub-proofs can be added at the beginning of the box in addition to the hypothesis (defense) α. The conclusions of the sub-proofs, $\neg\beta$ and $\neg\gamma$ respectively, can be used to fill in the gaps of the box and help find a contradiction for the attack $\{\neg\beta, \neg\gamma\}$. This way, if the resulting proof is visualized again, the same nodes will be generated again. By now, lines 4-16 in Figure 8.1 are generated.

There is also a small bit of housekeeping that takes place in the process as well. This is mainly to ensure that the line numbers of the sub-proofs are updated when included in higher-level proofs, and that references to givens that constituted parent defenses are rewired to point to the actual parent hypotheses as they become available by including sub-proofs in higher-level proofs. As a final step before returning the proof, the theory is stitched to the beginning of the proof and the line numbers are updated again.

8.3 Observations, Remarks and Future Work

It may be obvious by now that the proof extraction algorithm works as a counterpart of the visualization algorithm discussed previously in chapter 7. The algorithm ensures that the right boxes are created at the right parts of the proof so that the inverse of the mapping discussed in the visualization algorithm is followed.

Recall from chapter 7 that some information is lost when converting a proof to an argument regarding sibling proofs and child proofs. The visualization algorithm makes no distinction between the two, so the generated arguments seem "spread out". Thus, when converting back to a proof from the generated argument, only proofs that refer to child and ancestor hypotheses are created since shortcuts are effectively eliminated. Extracted proofs from arguments generated by visualizing proofs that were valid only under the extended definition of the Genuine Absurdity Property will now also follow the original definition.

Another bit of information that is dropped when visualizing proofs is the exact part of theory that may have been used along with the attack to reach a contradiction. This is because each argument (node) in the tree shows only its own claims, and hides the part of the theory with which, along with the defense under attack, lead to a contradiction. The example in Figure 8.2 shows two proofs that visualize to the same argument using the visualization algorithm, and either of those proofs can be extracted from the argument using the proof extraction algorithm. Note that the difference only lies in the theory used by the outermost box.

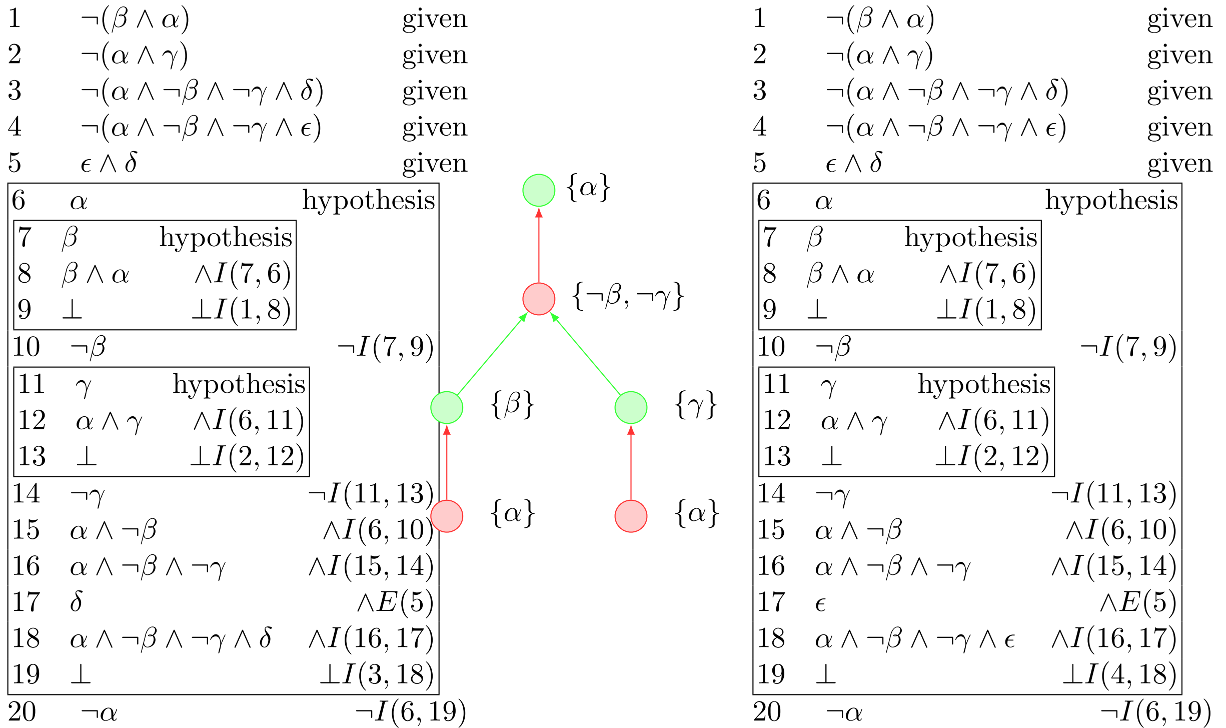

Figure 8.2: Proof extraction example of two proofs generated from the same argument

This means that when using the visualization algorithm to turn a proof into an argument, and then the extraction algorithm to turn the argument back into a proof, the final proof may not be the same as the initial. If the final proof is visualized again, then the first and final arguments will be the same however.

As discussed in chapter 7, the proof extraction algorithm could be upgraded with a

feature that enables it to identify repeated arguments and generate proofs that are more concise than the current output. Proofs can then be visualized and processed by this algorithm in order to receive more optimized versions than those input into the algorithm.

8.4 Details of Implementation

The algorithm first starts with the argument given in the same format as the output of the visualization algorithm and the theory on which the argument is based. The format of the argument is a list of two elements: a list of pairs of nodes (the actual arguments) and their IDs, and a list of pairs of node IDs indicating the attacks and defenses. Listing 8.1 shows that the proof is built from the ground up (line 2), and then the theory is stitched on it at the end (lines 3-4).

The call to `convertArgToGAP/4` is recursive, and although it starts from level 1 of the argument tree (as called by `convertArgToGAP/3`) it recursively bottoms out before starting to assemble the proof at each level. This can be seen on lines 6-8. First, all of the defenses against the attack on the current defense are found. These will make up the lower level boxes. Then, a sub-proof for each of those defenses is found by recursively calling itself. Hence by the time this is finished, the procedure bottomed out and then sub-proofs bubbled back up from the bottom with all the lower levels taken care of. Finally, `convertArgToGAP/4` acts on the current level of the tree by generating a proof using the theory and sub-proofs.

The predicate `makeSubProof/5` is responsible for the making of the sub-proof at each level of the tree. It takes the argument as a first parameter, the theory and child sub-proofs as the second and third parameters, and the node ID for the defense it is supposed to generate a proof for. The predicate returns the generated (sub-)proof as the last parameter. `parentSet/3` provides all the higher-level hypotheses that should be available when building the proof. `childSet/2` returns the negated hypotheses of the children. Its use will be discussed shortly. The `lineFix/4` predicate updates the line numbers of the previously generated child derivations so that they can fit nicely in the current-level proof. The theory and the parent set (masqueraded as more theory) are mixed together on line 15. Line 18 finds the actual defense that will be used as a hypothesis given that the defense node's ID is known and line 19 puts together the hypothesis and the lower-level derivations.

Now everything is set for the theorem prover to find a proof that brings the theory, ancestor hypotheses, current hypothesis and child derivations together with the attack. The proof is created on line 20 of Listing 8.1. In order to ensure that the theorem prover made use of all the components of the attack, the predicate `childGAPsUsed/4` performs a check that fails if the "wrong" proof is found. This is to keep the theorem prover in check so that it does not deviate from the attack plan. If the generated proof were to have a different cause for the contradiction, some of the child derivations could possibly not be referenced. In that case, re-visualizing the final proof would result in a different argument.

After passing this check, the resulting proof is the box containing the hypothesis and contradiction. The trailing conclusion is stuck at the end of the box (on the outside) by lines 23-29. These lines of code decide whether a $\neg E$ rule is used directly after the $\neg I$ rule and calculate the line numbers for those steps.

Before moving on, the `parentSet/3` and `childSet/2` predicates may deserve a quick glance. The former predicate gathers all the parent hypotheses by crawling the tree upwards from the indicated defense node. Line 35 finds the node ID of the attack that the defense tries to defend against, and line 36 finds the ID of the defense which is attacked by the aforementioned attack. Line 37 looks into the "nodes directory" to find the actual defense indicated by its ID. This is added to the parent set and then the predicate calls itself again

but this time to act on the higher-level defense. The `childSet/2` predicate acts slightly differently. The `makeSubProof/5` predicate returns proofs that have the conclusion drawn on the last line of the proof. Thus the negated hypotheses (which are the conclusions of each proof) can be gathered by reading the last step of each of the child derivations.

The predicate `m2/2` on lines 6-7, 18 and 35-37 acts as an alias to the standard `member/2` Prolog predicate. The predicate `a2/3` on lines 4 and 15 acts as an alias to the standard `append/3` Prolog predicate.

```
1  convertArgToGAP(Argument, Theory, TheoryAndProof) :-
2          convertArgToGAP(Argument, Theory, 1, Proof),
3          toSteps(Theory, TheorySteps),
4          a2(Proof, TheorySteps, TheoryAndProof).
5  convertArgToGAP([Nodes, AttDefs], Theory, NodeID, Proof) :-
6          findall(Def, (m2([Att, NodeID], AttDefs), m2([Def, Att], AttDefs)), Defs),
7          findall(SubProof, (m2(X, Defs), convertArgToGAP([Nodes, AttDefs], Theory,
               X, SubProof)), SubProofs),
8          makeSubProof([Nodes, AttDefs], Theory, SubProofs, NodeID, Proof).
9
10 makeSubProof([Nodes, AttDefs], Theory, ChildGAPs, NodeID, SubProof) :-
11         parentSet([Nodes, AttDefs], NodeID, ParentSet),
12         childSet(ChildGAPs, ChildSet),
13         lineFix(ParentSet, Theory, ChildGAPs, FixedChildGAPs),
14         append(FixedChildGAPs, MergedCGAPs),
15         a2(Theory, ParentSet, Context),
16         toSteps(Context, ContextSteps), !,
17         ln(ContextSteps, LineNumber1),
18         m2([[Node], NodeID], Nodes),
19         a2(MergedCGAPs, [step(Node, [hypothesis], LineNumber1)],
               HypothesisChildren),
20         backwardProve(no, HypothesisChildren, ContextSteps, [[], []], [falsity],
               BoxProof),
21         childGAPsUsed(BoxProof, Theory, [Node|ChildSet], ParentSet),
22         ln(BoxProof, NextLineNumber), is(LineNumber2, NextLineNumber - 1),
23         (
24                 Node = n(A),
25                 ln(NextLineNumber, NextNextLineNumber),
26                 SubProof = [step(A, [notE, NextLineNumber], NextNextLineNumber),
                       step(n(Node), [notI, LineNumber1, LineNumber2],
                       NextLineNumber), box(BoxProof)], !;
27
28                 SubProof = [step(n(Node), [notI, LineNumber1, LineNumber2],
                       NextLineNumber), box(BoxProof)], !
29         ),
30         !.
31
32 % Return the parent hypotheses of the given defense node
33 parentSet([_, _], 1, []).
34 parentSet([Nodes, AttDefs], NodeID, [Parent|ParentSet]) :-
35         m2([NodeID, AttackNodeID], AttDefs),
36         m2([AttackNodeID, ParentNodeID], AttDefs),
37         m2([[Parent], ParentNodeID], Nodes),
38         parentSet([Nodes, AttDefs], ParentNodeID, ParentSet).
39
40 % Returns a list of the negated child hypotheses for the given child proofs
41 childSet([], []).
42 childSet([[step(NegHyp, _, _)|_]|ChildGAPs], [NegHyp|ChildHypotheses]) :-
```

```
43        childSet(ChildGAPs, ChildHypotheses).
```

Listing 8.1: First part of the proof extraction algorithm

The childGAPsUsed/4 predicate is responsible for making sure that the generated proof at each level makes use of the lower-level proofs. Lines 3-8 of Listing 8.2 retrieve the step of the proof responsible for the contradiction (the attack), and then it is broken down into individual components on line 9. Line 10 checks that the given child set is included in the attack components. This ensures that all the lower-level subtrees of the argument tree will be present should the resulting proof be visualized again. As a further precaution, the child set is subtracted from the set of used components and the result is checked against a set containing components from the theory and parent set. This is to ensure that the generated proof does not contain any extra items that would add more branches to the argument received by visualizing the extracted proof again. This is done by lines 11-15.

The predicate getConjunctionComponents/2 is fairly simple. It breaks down conjunctions into individual components. The predicate m2/2 on lines 8 and 12 acts as an alias to the standard member/2 Prolog predicate. The predicate a2/3 on lines 14 and 22 acts as an alias to the standard append/3 Prolog predicate.

```
1  % Make sure that all the (negations of the) child hypotheses were used in the
       attack
2  childGAPsUsed(Proof, Theory, ChildSet, ParentSet) :-
3         (
4                Proof = [step(falsity, [falsityI, _, LN], _)|_];
5
6                Proof = [step(falsity, [check, _], _), step(falsity, [falsityI, _,
                     LN], _)|_]
7         ),
8         m2(step(X, _, LN), Proof),
9         getConjunctionComponents(X, UsedComponents),
10        subset(ChildSet, UsedComponents),
11        subtract(UsedComponents, ChildSet, Rest),
12        findall(TheoryComponents, (m2(T, Theory), getConjunctionComponents(T,
              TheoryComponents)), TheoryComponentsList),
13        append(TheoryComponentsList, AllowedTheoryComponents),
14        a2(AllowedTheoryComponents, ParentSet, AllowedComponents),
15        subset(Rest, AllowedComponents).
16
17 % Returns the components making up a conjunction
18 % Example: a&b&c&d -> [a,b,c,d]
19 getConjunctionComponents(and(A, B), Components):-
20        getConjunctionComponents(A, Component1),
21        getConjunctionComponents(B, Component2),
22        a2(Component1, Component2, Components), !.
23 getConjunctionComponents(X, [X]).
```

Listing 8.2: Second part of the proof extraction algorithm

The predicate lineFix/4 (shown in Listing 8.3) is used to update the line numbers in the proofs for lower-level nodes. Its usage is solely related to housekeeping. Each of the child derivations considers the first few lines to be the theory and the parent hypotheses. Thus all proofs start from a certain line number $n = length(Theory) + length(ParentSet)$. Thus every line number proof has to be "reset" by subtracting this n and adding an offset which is calculated for each proof as they will appear one after another. The offset is the last line number of each proof for intermediate proofs. Care must be taken to not shift all

line references as some references refer to the theory or parent hypotheses. Shifting those numbers (numbers smaller than n) will result in wrong references.

```
% Changes the lines of the proofs of the child proofs so that no two steps
% have the same line number. References are updated as well
lineFix(ParentSet, Theory, ChildGAPs, FixedChildGAPs) :-
        length(ParentSet, L),
        length(Theory, T),
        lineFix(L, ChildGAPs, T, T, RevFixedChildGAPs),
        reverse(RevFixedChildGAPs, FixedChildGAPs).
lineFix(_, [], _, _, []).
lineFix(From, [ChildGAP|ChildGAPs], Offset, TheoryOffset,
    [FixedChildGAP|FixedChildGAPs]) :-
        shiftLines(From, Offset, TheoryOffset, ChildGAP, FixedChildGAP),
        ln(ChildGAP, NewOffset),
        lineFix(From, ChildGAPs, NewOffset -1, TheoryOffset, FixedChildGAPs).
shiftLines(_, _, _, [], []).
shiftLines(From, Offset, TheoryOffset, [step(A, [R|Reason], LineNumber)|Proof],
    [step(A, [R|FixedReason], FixedLineNumber)|FixedProof]) :-
        shiftNumbers(From, Offset, TheoryOffset, [LineNumber|Reason],
            [FixedLineNumber|FixedReason]),
        shiftLines(From, Offset, TheoryOffset, Proof, FixedProof).
shiftLines(From, Offset, TheoryOffset, [box(BoxProof)|Proof],
    [box(FixedBoxProof)|FixedProof]) :-
        shiftLines(From, Offset, TheoryOffset, BoxProof, FixedBoxProof),
        shiftLines(From, Offset, TheoryOffset, Proof, FixedProof).
shiftNumbers(_, _, _, [], []).
shiftNumbers(From, Offset, TheoryOffset, [N|Numbers], [FN|FixedNumbers]) :-
        (
                N > From + TheoryOffset, is(FN, N + Offset - TheoryOffset - From);

                FN = N
        ),
        shiftNumbers(From, Offset, TheoryOffset, Numbers, FixedNumbers).
```

Listing 8.3: Third part of the proof extraction algorithm

8.5 Example Walkthrough

The example will revolve around the example argument given in Figure 8.1. Assume that the format of the input argument is $Args = [(\{\alpha\}, 1), (\{\neg\beta, \neg\gamma\}, 2), (\{\beta\}, 3), (\{\alpha\}, 4), (\{\gamma\}, 5), (\{\alpha\}, 6)]$ for the nodes and their IDs, and $Att = [(1, 0), (3, 2), (4, 3), (5, 2), (6, 5)]$ for the attacks/defenses. In the Prolog implementation, lists, tuples and sets were all implemented as lists, but for the sake of readability a richer format will be used.

Listing 8.1 shows `convertArgToGAP/3` starting up `convertArgToGAP/4` with node ID 1, the root of the tree. Line 6 of the code finds all defenses that defend against this node's attack from Att and puts them in a list, which in this case is $[3, 5]$. Line 7 of the code calls `convertArgToGAP/4` for each of those defenses, passing in the appropriate ID.

For the call with node ID 3, line 6 of the code executes and finds no defenses against attack node 4. Hence no sub-proofs are generated by line 7. Line 8 executes. The `makeSubProof/5` predicate runs and calculates the parent set $\{\alpha\}$ and child set $\{\}$ (since there are no child derivations). Predicate `lineFix/4` has no effect for the same reason. Line 15 of the code puts together the theory and parent set. Line 18 looks up node ID 3 in $Args$

in order to find the actual defense $\{\beta\}$. The theorem prover then is tasked with finding a proof with hypothesis β, context theory plus parent set and goal a contradiction. Assume that the following proof was returned:

5	β	hypothesis
6	$\beta \wedge \alpha$	$\wedge I(5, 4)$
7	$\bot$	$\bot I(1, 6)$

Line 21 calls `childGAPsUsed/4`. Listing 8.2 shows the definition of this predicate. Lines 3-8 of the code find the step responsible for the contradiction. This is line 6 in the proof. It is split into individual components by line 9 of the code. The components are $[\beta, \alpha]$. The given child set is not the empty set not because there are child derivations but rather because the defense itself was appended to it by `makeSubProof/5`. This is because the hypothesis can be present in the conjunction resulting in the contradiction (in fact it has to if the proof is to follow the Genuine Absurdity Property). The first test (line 10) passes. The theory and parent are split into individual components giving $[\neg(\beta \wedge \alpha), \neg(\alpha \wedge \gamma), \neg(\alpha \wedge \neg\beta \wedge \neg\gamma), \alpha]$. The theory did not have any conjunction so it was left intact. If, as an example, $\delta \wedge \epsilon$ was part of the theory then it would produce components δ and ϵ. Subtracting the child set from the attack components leaves the list with only one element in it. $[\alpha]$ is part of the allowed remaining components as tested by line 15 and the `childGAPsUsed/4` predicate returns successful.

Back in the `makeSubProof/5` predicate on line 21 (Listing 8.1) the execution continues and lines 22-29 decide that because the defense β is a positive atom, there should be only one line with the conclusion $\neg\beta$ trailing the box. Thus before returning, the predicate assembles this proof:

5	β	hypothesis
6	$\beta \wedge \alpha$	$\wedge I(5, 4)$
7	$\bot$	$\bot I(1, 6)$
8	$\neg\beta$	$\neg I(5, 7)$

In a similar way, a box for node 5 is created that looks like the following:

5	γ	hypothesis
6	$\alpha \wedge \gamma$	$\wedge I(4, 5)$
7	$\bot$	$\bot I(2, 6)$
8	$\neg\gamma$	$\neg I(5, 7)$

Note that the two sub-proofs have the same line numbers because at creation time they were unaware of each other's existence. This issue will be fixed by the `lineFix/4` predicate shortly. On the top tree level now, after line 7 returns in Listing 8.1 for node 1, `makeSubProof5` is run but this time with the two proofs for the child derivations.

On line 11, the parent set is an empty set as node 1 has no parent defenses up the tree (since it's the root). The child set however is non-empty; it is $\{\neg\beta, \neg\gamma\}$. The `lineFix/4` predicate changes the line numbers of the sub-proofs. The first child proof is effectively left intact, but the second proof now starts from where the first left off; line number 9. Hence after the line fix the two proofs are:

5	β	hypothesis
6	$\beta \wedge \alpha$	$\wedge I(5, 4)$
7	$\bot$	$\bot I(1, 6)$
8	$\neg\beta$	$\neg I(5, 7)$

9	γ	hypothesis
10	$\alpha \wedge \gamma$	$\wedge I(4, 9)$
11	$\bot$	$\bot I(2, 10)$
12	$\neg\gamma$	$\neg I(9, 11)$

Line 19 merges the hypothesis (defense) α with the child proofs, and the theorem prover is then tasked to find a proof that results in a contradiction on line 20. Assume that the returned proof is the following:

Line	Formula	Justification
4	α	hypothesis
5	β	hypothesis
6	$\beta \wedge \alpha$	$\wedge I(5, 4)$
7	$\perp$	$\perp I(1, 6)$
8	$\neg\beta$	$\neg I(5, 7)$
9	γ	hypothesis
10	$\alpha \wedge \gamma$	$\wedge I(4, 9)$
11	$\perp$	$\perp I(2, 10)$
12	$\neg\gamma$	$\neg I(9, 11)$
13	$\alpha \wedge \neg\beta$	$\wedge I(4, 8)$
14	$\alpha \wedge \neg\beta \wedge \neg\gamma$	$\wedge I(13, 12)$
15	$\perp$	$\perp I(3, 14)$

Note that lines 4-12 were already there at the start of the proof search as they were stitched together by the `makeSubProof/5` predicate and handed in to the theorem prover for the construction of a contradiction. The next line to execute is line 21 where `childGAPsUsed/4` is called. Listing 8.2 shows this predicate. On lines 3-8 the step contributing to the contradiction is located (line 14 of the proof) and then it is broken down into individual components (line 9 of the code). The components are $[\alpha, \neg\beta, \neg\gamma]$. The child set (which includes the current defense α) is the same, which does make it a subset (not strictly speaking). The subtraction leaves no remaining components to be checked. This step is just a precaution however, as the generated proof could have an extra attack component that could generate a slightly different tree if the result was to be visualized again. The predicate returns successfully. Back in the predicate `makeSubProof/5` shown in Listing 8.1 lines 23-29 decide that the hypothesis is a positive atom, and thus the conclusion is just it's negation (there is no need to apply the $\neg E$ rule afterwards as the conclusion is not doubly-negated). Thus one last line deriving $\neg\alpha$ from the box is appended and the resulting proof looks like the following:

Line	Formula	Justification
4	α	hypothesis
5	β	hypothesis
6	$\beta \wedge \alpha$	$\wedge I(5, 4)$
7	$\perp$	$\perp I(1, 6)$
8	$\neg\beta$	$\neg I(5, 7)$
9	γ	hypothesis
10	$\alpha \wedge \gamma$	$\wedge I(4, 9)$
11	$\perp$	$\perp I(2, 10)$
12	$\neg\gamma$	$\neg I(9, 11)$
13	$\alpha \wedge \neg\beta$	$\wedge I(4, 8)$
14	$\alpha \wedge \neg\beta \wedge \neg\gamma$	$\wedge I(13, 12)$
15	$\perp$	$\perp I(3, 14)$
16	$\neg\alpha$	$\neg I(4, 15)$

The execution returns to `convertArgToGAP/4` and then to `convertArgToGAP/3` on line 2. The theory is finally appended to the proof and the result is the proof shown in Figure 8.1.

Chapter 9

Server Module

The main purpose of the server is to act as a conduit between the core module and the client, since they are written in different programming languages and runtime environments. As mentioned in the overview in chapter 3, the server is written in SWI-Prolog[1], using the provided HTTP package[2] that offers a comprehensive server solution out of the box. SWI-Prolog was used also because the core module was written in the same Prolog implementation, and using the module was as simple as loading the files into the runtime operating the server application.

The server covers two functions:

- it serves the files required by the client to run - these are HTML, CSS and JavaScript files that are requested by the browser
- it serves queries issued by the client as the user operates the graphical frontend in the browser - these are JSON HTTP requests

The exact nature of the server is explained in the next section and alternatives are discussed afterwards.

9.1 Implementation Details

Thanks to the provided HTTP Package, most of the implementation of the server is hidden away. The server code imports the package, configures the server, registers the handlers and fires up the server. These steps will be explained in more detail in this section.

```
1  :- use_module(library(http/thread_httpd)).
2  :- use_module(library(http/http_dispatch)).
3  :- use_module(library(http/http_files)).
4
5  init :- conf(document_root, D), http_handler(root(.), http_reply_from_files(D,
       []), [prefix]), registerQueries.
6  fin :- http_delete_handler(root(.)), deregisterQueries.
7  startServer :- conf(port, P), http_server(http_dispatch, [port(P)]).
8  stopServer :- conf(port, P), http_stop_server(P, []).
9
10 boot :- init, startServer.
11 shutdown :- fin, stopServer.
```

Listing 9.1: The Prolog code that configures and runs the server

[1]http://www.swi-prolog.org/

[2]http://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/http.html%27)

The first three lines of Listing 9.1 load the necessary packages, and line 5 takes the document root from the configuration file, and adds a handler that allows the server to serve any file requests under that document root. This handler is the one that serves the HTML, CSS and JavaScript files of the client. It then proceeds to run the `registerQueries/0` predicate (Listing 9.3) that registers the handlers responsible for serving the queries made by the client. The predicate `startServer/0` on line 7 of the code listing reads the port number from the configuration file and starts an HTTP server that listens on the port specified by the configuration file.

Line 10 is merely a shortcut that provides a predicate `boot/0` that runs the code on lines 5 and 7. Lines 6, 8 and 11 undo what lines 5, 7 and 10 do respectively. They deregister the server handlers and shutdown the HTTP server. This can also be accomplished by closing the Prolog runtime as well.

```
1  conf(document_root, 'C:/users/george/thesis/client/web/').
2  conf(port, 8000).
```

Listing 9.2: The server configuration file written in Prolog

The small configuration file that specifies the port and document root of the server is shown in Listing 9.2. This configuration file can be customized according to the environment in which the server is meant to run. In order to deploy the server in a different environment only this file needs to be edited. The configuration file may be extended to house more settings in future updates.

```
1  :- use_module(library(http/json)).
2  :- use_module(library(http/http_json)).
3  :- use_module(library(http/json_convert)).
4  :- use_module(library(http/http_error)).
5
6  registerQueries :-
7          http_handler(root(query), jsonEcho, []),
8          http_handler(root(query/generateproofs), serveGenerateProofs, []),
9          http_handler(root(query/checkgap), serveGAPCheck, []),
10         http_handler(root(query/visualizegap), serveGAPToArg, []),
11         http_handler(root(query/visualizearg), serveArgToGAP, []),
12         http_handler(root(query/provable), serveProvable, []).
13
14 deregisterQueries :-
15         http_delete_handler(root(query)),
16         http_delete_handler(root(query/generateproofs)),
17         http_delete_handler(root(query/checkgap)),
18         http_delete_handler(root(query/visualizegap)),
19         http_delete_handler(root(query/visualizearg)),
20         http_delete_handler(root(query/provable)).
21
22 :- json_object and('1', '2') + [type=and].
23 :- json_object or('1', '2') + [type=or].
24 :- json_object implies('1', '2') + [type=implies].
25 :- json_object n('1') + [type=n].
26 :- json_object step(derivation, reason, line) + [type=step].
27 :- json_object box(proof) + [type=box].
28 :- json_object proof_query(theory, goal) + [type=proof_query].
29 :- json_object gap_query(proof, check) + [type=gap_query].
30 :- json_object arg_view_query(proof) + [type=arg_view_query].
31 :- json_object arg('1', '2') + [type=arg].
```

```
32 :- json_object gap_view_query(arg, theory) + [type=gap_view_query].
33 :- json_object provable_query(theory, goal, mra) + [type=provable_query].
34
35 jsonEcho(R) :-
36         format('Content-type: text/plain~n~n'),
37         http_read_json(R, J),
38         json_to_prolog(J, P),
39         write(P).
40
41 serveGenerateProofs(Request) :-
42         http_read_json(Request, JSONIn),
43         json_to_prolog(JSONIn, proof_query(Theory, Goal)),
44         findall(X, (prove(Theory, [Goal], Y), reverseRecursive(Y, X)), PrologOut),
45         prolog_to_json(PrologOut, JSONOut),
46         reply_json(JSONOut).
47
48 serveGAPCheck(Request) :-
49         http_read_json(Request, JSONIn),
50         json_to_prolog(JSONIn, gap_query(Proof, Check)),
51         reverseRecursive(Proof, RevProof),
52         (
53                 (
54                         Check = classic,
55                         checkGAP(RevProof);
56
57                         Check = extended,
58                         checkGAPX(RevProof)
59                 ),
60                 prolog_to_json('approved', JSONOut);
61
62                 prolog_to_json('disproved', JSONOut)
63         ),
64         reply_json(JSONOut).
65
66 serveGAPToArg(Request) :-
67         http_read_json(Request, JSONIn),
68         json_to_prolog(JSONIn, arg_view_query(Proof)),
69         reverseRecursive(Proof, RevProof),
70         convertGAPToArg(RevProof, PrologOut1, PrologOut2),
71         prolog_to_json(arg(PrologOut1, PrologOut2), JSONOut),
72         reply_json(JSONOut).
73
74 serveArgToGAP(Request) :-
75         http_read_json(Request, JSONIn),
76         json_to_prolog(JSONIn, gap_view_query(Argument, Theory)),
77         convertArgToGAP(Argument, Theory, Proof),
78         reverseRecursive(Proof, RevProof),
79         prolog_to_json(RevProof, JSONOut),
80         reply_json(JSONOut).
81
82 serveProvable(Request) :-
83         http_read_json(Request, JSONIn),
84         json_to_prolog(JSONIn, provable_query(Theory, Goal, MRA)),
85         provable(Theory, [Goal], MRA, Verdict),
86         prolog_to_json(Verdict, JSONOut),
87         reply_json(JSONOut).
```

Listing 9.3: The server code file that registers handlers that server client queries

Listing 9.3 shows the server code that deals with the queries issued by the client as the user operates the graphical interface. Lines 1-4 of the code import more SWI-Prolog libraries, and lines 6-12 define the `registerQueries/0` predicate that is called by the main server file predicate `init/0`. This predicate registers the handlers for queries by the client under the relative path *query/*. The handlers registered are predicates that take one parameter, which is the actual request made by the client. From this request the JSON string can be extracted and easily converted to Prolog by the JSON object declarations on lines 22-33. The JSON strings include details about the specific request (such as the theory and the goal for generating proofs, or the proof that needs to be checked whether it follows the Genuine Absurdity Property or not). After extracting and converting to Prolog terms, the handler predicate can run code from the core module, and then respond with the results after converting them back to JSON format.

The JSON object declarations are used to automatically convert JSON objects to Prolog terms and vice versa. The values of the keys in a JSON object will become the parameters of the corresponding Prolog term, positioned by their keys as specified by the declaration. The functor (predicate name) is extracted from the value of the special key `type` in the JSON object. For example, line 28 defines the Prolog term `proof_query(theory, goal)` as the equivalence of the JSON object `{"type":"proof_query", "theory":"...", "goal":"..."}`. Line 22 defines conjunction in Prolog terms in a similar way. Thus, a request with JSON object `{"type":"proof_query", "theory":[{"type":"and", "1":"a", "2":"b"}], "goal":"a"}` translates to `proof_query([and(a, b)], [a])`. The predicate `json_to_prolog/2` does this conversion from JSON to Prolog, as the name implies, and predicate `prolog_to_json/2` does the opposite.

Going back to the handler predicates, a number of them are registered by `registerQueries/0`:

- on line 35, `jsonEcho/1` is used to check that the server is set up properly for debugging purposes and is not used by the client
- on line 41, `serveGenerateProofs/1` is used to take a theory and a goal and respond with all the proofs that prove the goal using the given theory (chapter 4)
- on line 48, `serveGAPCheck/1` takes a proof and responds appropriately depending on whether the proof follows the (extended) Genuine Absurdity Property or not (chapter 5, chapter 6)
- on line 66, `serveGAPToArg/1` takes a proof that follows the Genuine Absurdity Property and returns a visualization (chapter 7)
- on line 74, `serveArgToGAP/1` takes an argument (a visualization) and the theory it is based on and extracts a proof that when visualized again, provides the same argument (chapter 8)
- on line 82, `serveProvable/1` takes a theory and a goal and checks whether the goal can be proven; no proof is sent back to the client; used by the argument builder to check for consistent attacks provided by the user (chapter 12)

9.2 Remarks

As seen from the previous section, the server is quite simple to use and due to the fact that it is very minimalistic, it requires virtually no maintenance. Its requirements in terms of its

host environment are very low. A full installation of SWI-Prolog and its libraries suffices to run the server. No other software or libraries are necessary. Throughout the construction of the application, the server has proven to be very reliable and resilient, and no bugs or any type of problems were detected.

9.3 Alternatives

Different alternatives were considered before the construction of the server. This section summarizes the alternate paths explored and justifies the final decision.

Since the core module is written in Prolog, communication with or support for running such an environment was deemed necessary. There were different solutions that mainly fall under two general categories:

- using a library that forms a bridge between Prolog and another programming language (which provides libraries that offer server functionality) that makes use of SWI-Prolog's C/C++ interface

- using a Prolog engine implementation written in another programming language (which provides libraries that offer server functionality)

Regardless of choice, the methodology boils down to the same procedure: use a third-party software to either communicate with a running Prolog instance (through SWI-Prolog's C/C++ interface) or run the core module directly by using the third party's Prolog implementation. Several candidates were considered, including pyswip[3], jpl[4] and C#Prolog[5].

Most of the alternatives were dropped because of their lack of documentation and support, or because the projects were abandoned with bugs and issues still raised. C#Prolog offers its own ISO Prolog implementation in C#, and C# offers great server solutions that are highly popular and reliable, such as Microsoft's IIS[6]. Unfortunately, the employment of C#Prolog was rejected, as many of the SWI-Prolog-specific libraries that are used by the core module would have to be re-implemented.

A completely different approach would be to implement the graphical user interface and all other functionality of the client in pure Prolog. This would eliminate the necessity for the server altogether. This approach and what is entailed is discussed in the client module chapter in section 10.3.

[3]https://code.google.com/p/pyswip/
[4]http://www.swi-prolog.org/packages/jpl/java_api/
[5]http://sourceforge.net/projects/cs-prolog/
[6]http://www.iis.net/

Chapter 10

Client Module

The purpose of the client module is to wrap around the core module in order to provide a better and easier experience than running the different Prolog predicates of the core on a Prolog command line interface.

The client is written in HTML, JavaScript and CSS, and is loaded upon a web browser request from the server (chapter 9).

10.1 Features

The main features of the client are the following:

- easy-to-use interface that allows users to not worry about using the core predicates properly, what their inputs should look like and how to make sense of the output returned by Prolog, since modern UI elements and visualization of the output guide the user
- syntax parser and various input checkers that prevent the user from inputting incorrect theory, goals, attacks, and other forms of input into the system
- modern user interface that allows for intuitive use of the application
- data management functionality such as saving generated results (proofs, visualizations, etc), deleting, importing or exporting them
- additional tools for building proofs and arguments
- a help section with useful information on how to use the client's features

The next section shows how to use the client module and introduces the interface equivalents of the core module predicates, while chapter 11 and chapter 12 are dedicated to using the additional proof builder and argument builder tools respectively, that are exclusive to the client module.

10.2 Usage

This section demonstrates the different components of the client and shows how it can be used.

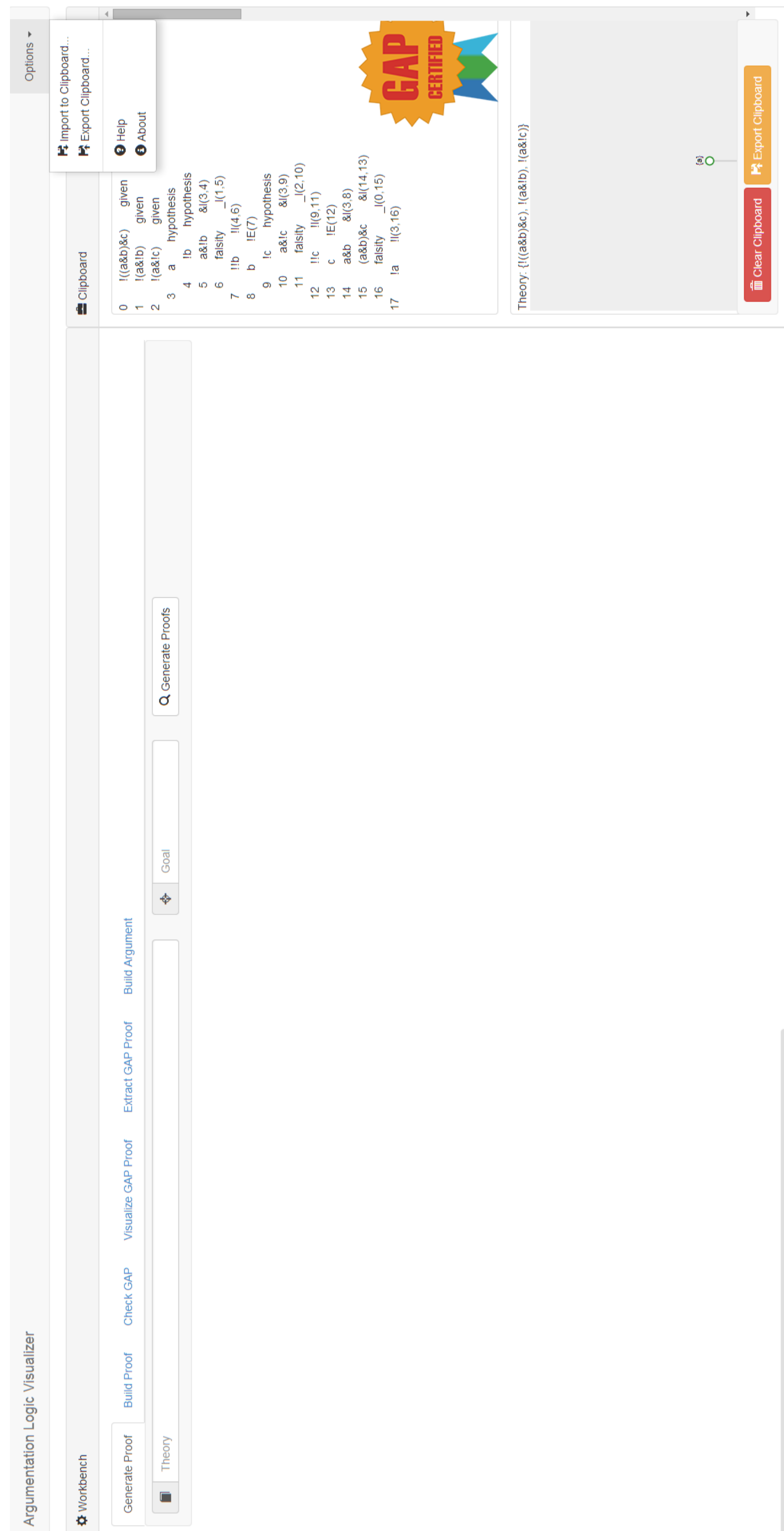

Figure 10.1: The client GUI showing the workbench on the right, the clipboard on the left and the options on the top right corner

10.2.1 Clipboard

The clipboard holds proofs and other items that need to be retained. It acts as storage but also acts as a pool of available items to use on the Workbench.

The clipboard is always visible on the right hand side of client application as shown in Figure 10.1.

Thumbnails

Each item is rendered in its own thumbnail, and can be dragged to its appropriate placeholder on the "workbench" (explained later) or on the "Clear Clipboard" or "Export Clipboard" buttons described later. Figure 10.2 shows three different items, each rendered in a thumbnail: a proof, another proof that follows the Genuine Absurdity Property, and finally, an argument.

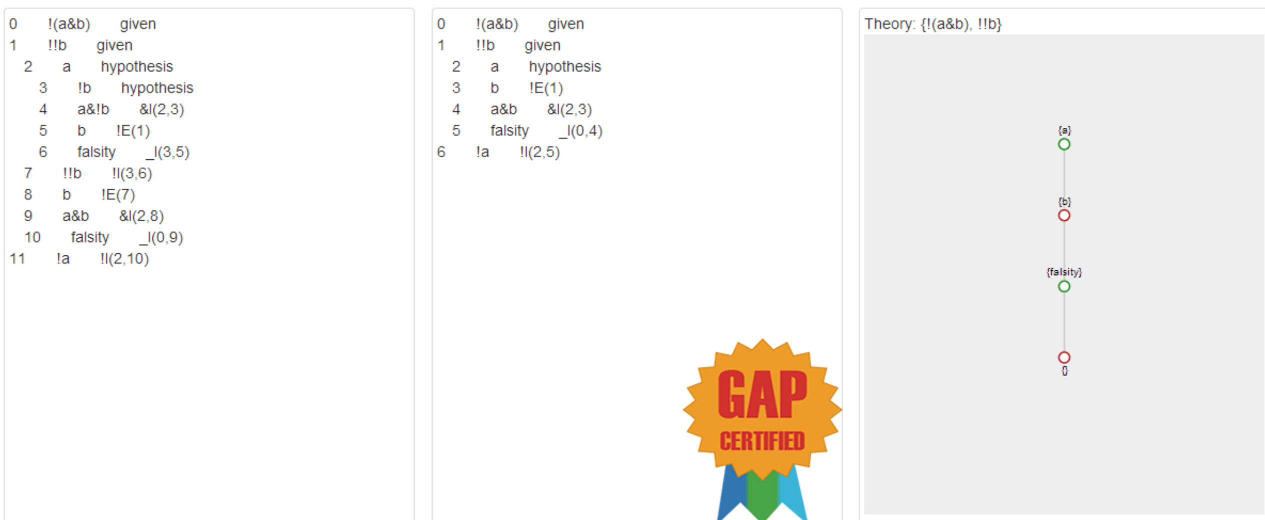

Figure 10.2: Three thumbnails, from left to right: a proof, a verified Genuine Absurdity Property proof, and an argument

Persistent Storage

The application makes use of HTML 5's web storage[1] feature in order to provide persistent storage for the items in the clipboard. The content of the clipboard remains intact even if the webpage is closed or reloaded. However, this storage may be deleted if the browser clears its data, so it should not be heavily relied upon. There is an export clipboard feature that can export the items stored in the clipboard.

The web storage keeps the clipboard in sync with itself. Whenever an item is added or deleted, or when the clipboard is cleared using the "Clear Clipboard" button, the web storage is updated accordingly. The data is automatically loaded back into the clipboard when the page is refreshed.

Managing the Clipboard

The clipboard allows the deletion of its contents, either in its entirety or one by one. Clicking the "Clear Clipboard" button will delete all the contents of the Clipboard, but

[1] http://en.wikipedia.org/wiki/Web_storage

after a warning pop-up appears (as shown by Figure 10.3). Dropping an item from the clipboard onto this button will delete the item from the clipboard. There is no warning pop-up for dragging a single item onto this button and that item is deleted immediately.

Clicking on the "Export Clipboard" button will export the entire clipboard. A thumbnail can also be dragged and dropped onto the "Export Clipboard" button. This will open a new window with the exported item inside. The import/export facility is discussed in subsection 10.2.3.

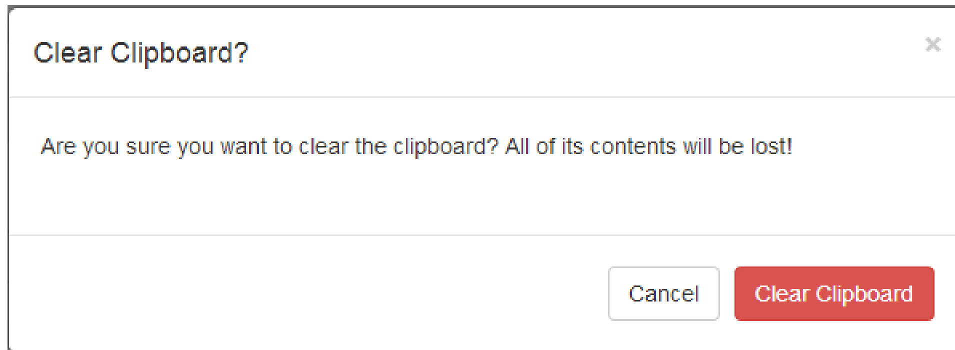

Figure 10.3: A notification pops up when the user clicks on the "Clear Clipboard" button that asks for confirmation

10.2.2 Workbench

The Workbench is where the different actions happen. The different tasks that can be performed have been split into tabs that all fall under the workbench. Some of these tasks require the user to input pre-made proofs or arguments from the Clipboard, or to input logic sentences or natural deduction steps. The syntax used for logic sentences and natural deduction is explained in subsection 10.2.4. The workbench is always located on the left hand side of the screen, and occupies most of the real estate of the client screen as shown in Figure 10.1.

Generate Proof Tab

The purpose of this tab is to let the user enter a theory and a goal and use the theorem prover from chapter 4 to return all the proofs that prove the goal set by the user.

The user enters a comma-separated list of formulas in the theory input box, and a single formula in the goal input box, and clicks on the "Generate Proofs" button in order to send the data to the server. If the parsed input is incorrect, the input field with the incorrect input will be highlighted so that the user can amend their input. This is shown in Figure 10.4. As soon as all the proofs are generated and sent back, the space below the text fields and button will display the proofs as shown in Figure 10.5. The user can then drag the preferred proofs onto the clipboard to save them.

Build Proof Tab

This functionality is only available in the client module. The core module does not have an equivalent function. It is described in its own separate chapter in chapter 11.

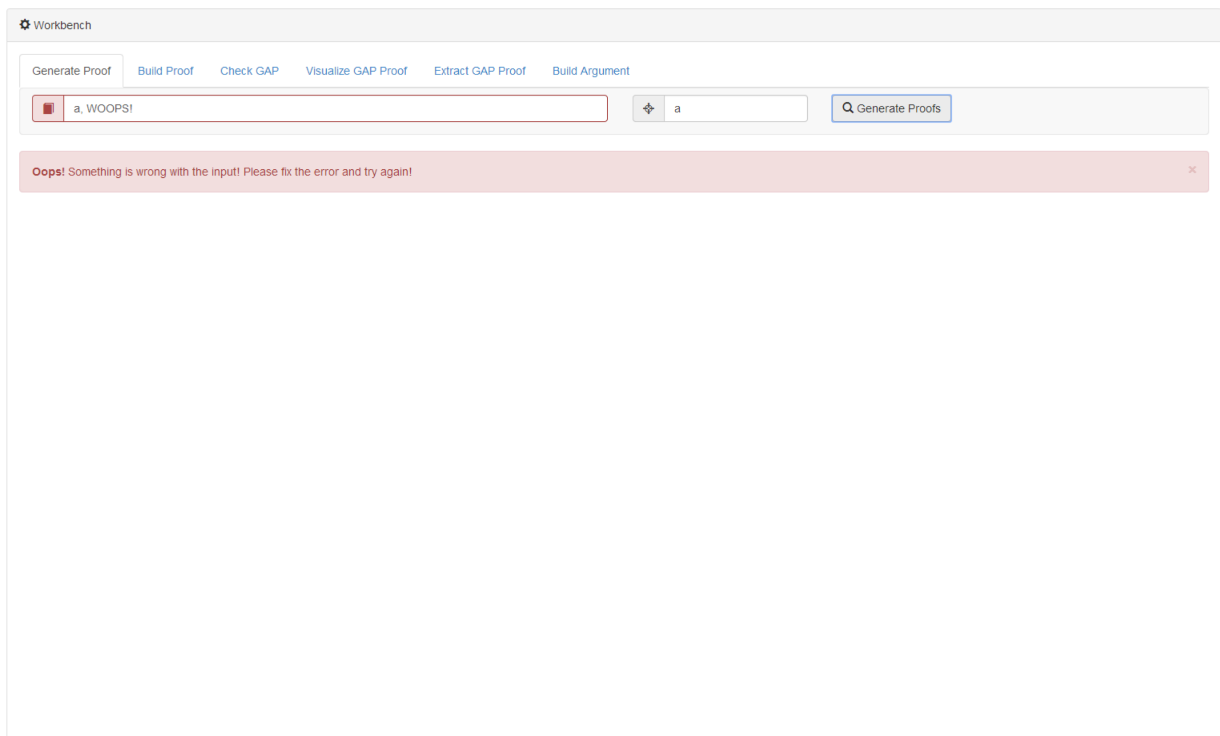

Figure 10.4: Incorrect input is highlighted so that the user can revise it

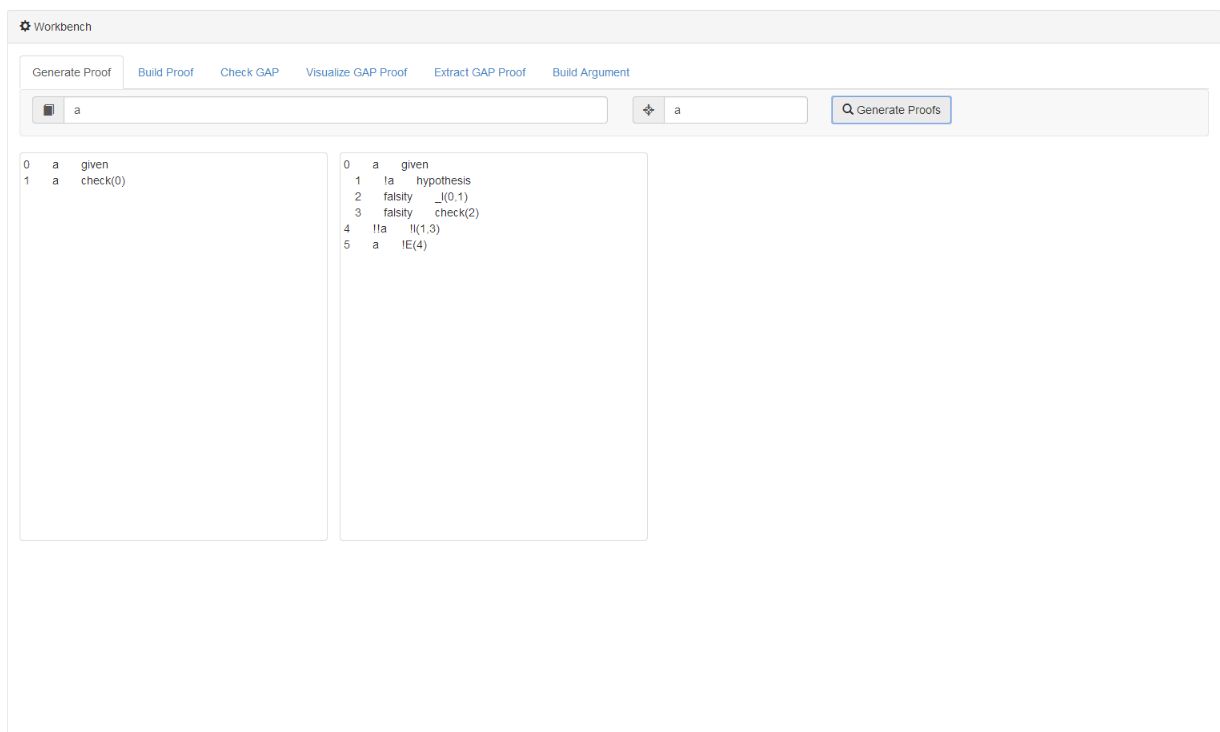

Figure 10.5: The theorem prover generates all the proofs that arrive to the given goal using the supplied theory

Check GAP Tab

This tab is used to validate proofs as following the Genuine Absurdity Property. The user can drag a proof from the clipboard that has not already been verified onto the thumbnail placeholder under that tab. The placeholder will then be filled in with the proof. The next step is to select the version of the Genuine Absurdity Property (discussed in chapter 5 and chapter 6) the user would like to use. This could be either the original definition or the extended definition. This is shown by Figure 10.6.

Finally, the user can click on the "Check GAP" button in order to check whether the proof does indeed follow the Genuine Absurdity Property by querying the server. If the proof follows the property then this is shown by a small ribbon at the bottom right edge of the thumbnail containing the proof under examination as shown in Figure 10.7. The user can then drag the Genuine Absurdity Property-verified proof onto the clipboard.

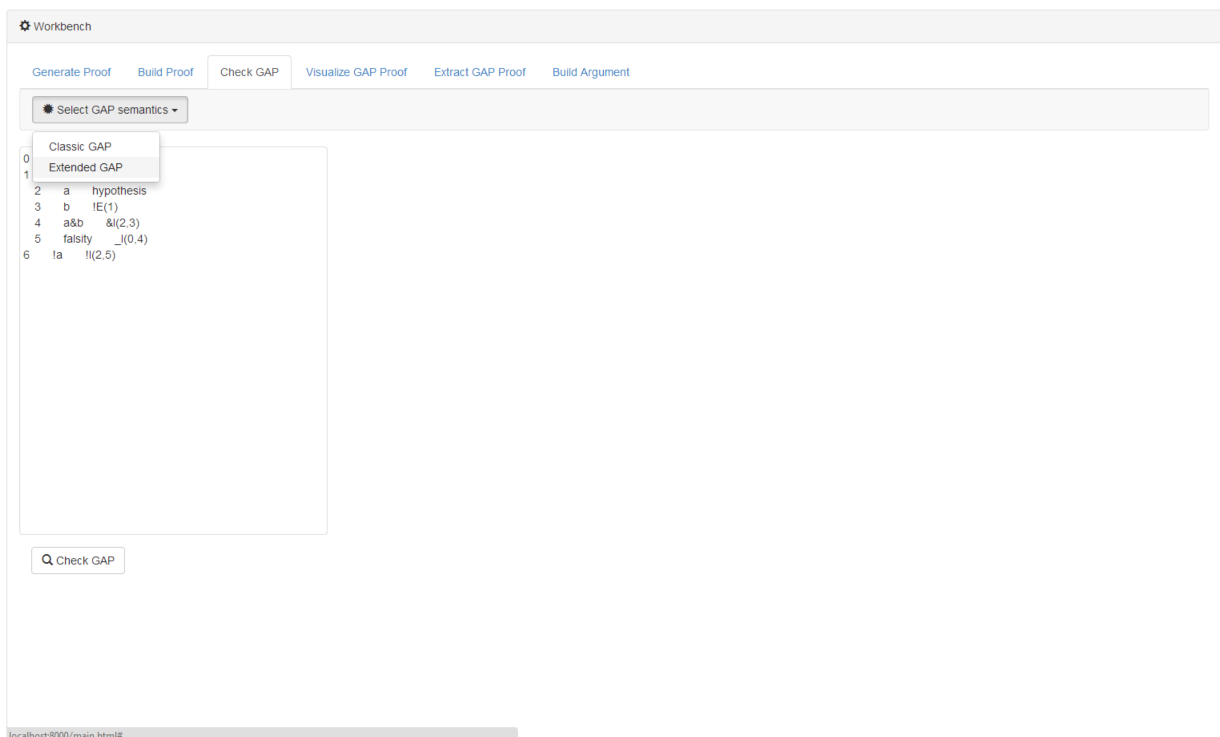

Figure 10.6: The user can drop an unverified proof onto the placeholder and select between the original and extended definitions for the Genuine Absurdity Property

Visualize GAP Tab

This tab allows for Genuine Absurdity Property-compliant proofs to be visualized using the algorithm devised in chapter 7. This tab holds two placeholders. The user can drag a compliant proof from the clipboard onto the left placeholder and click on the "Visualize GAP Proof" button. The proof will then be visualized into an argument and placed onto the right placeholder so that they can be viewed side-by-side. Finally the user can drag the visualization onto the clipboard in order to save it. A proof and its corresponding argument are shown side-by-side in Figure 10.8.

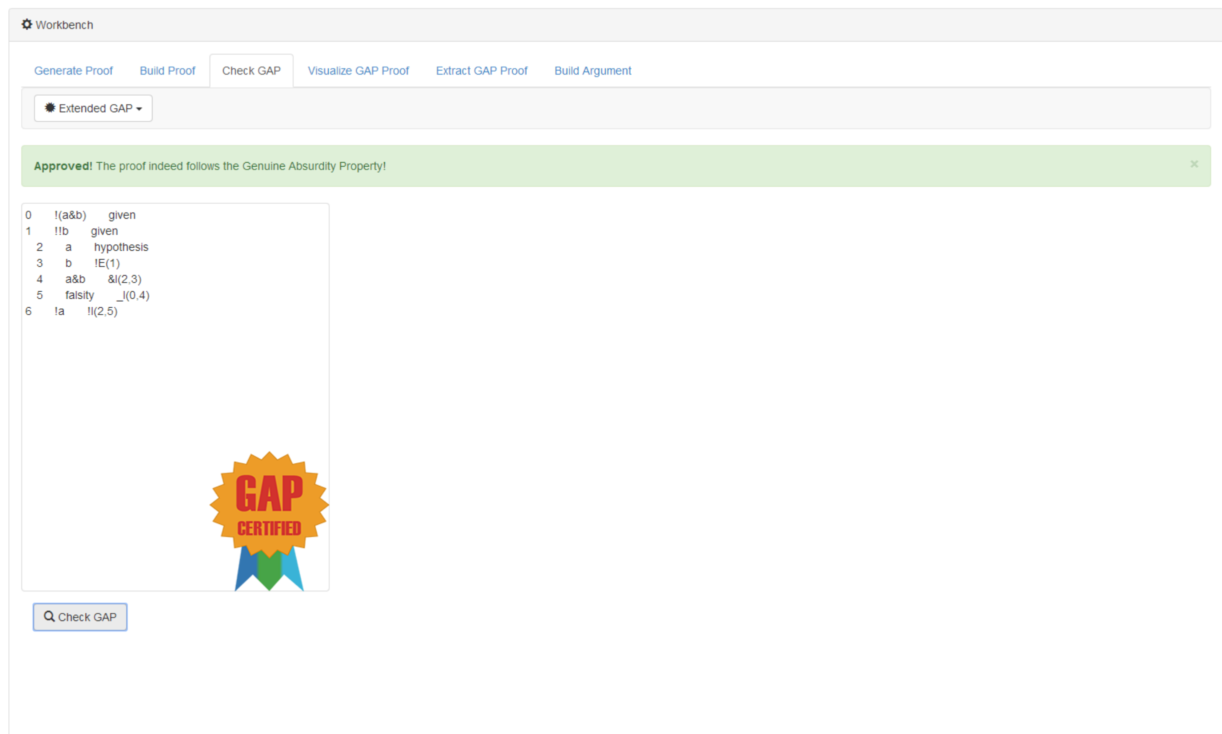

Figure 10.7: If the proof follows the Genuine Absurdity Property it is indicated using a ribbon

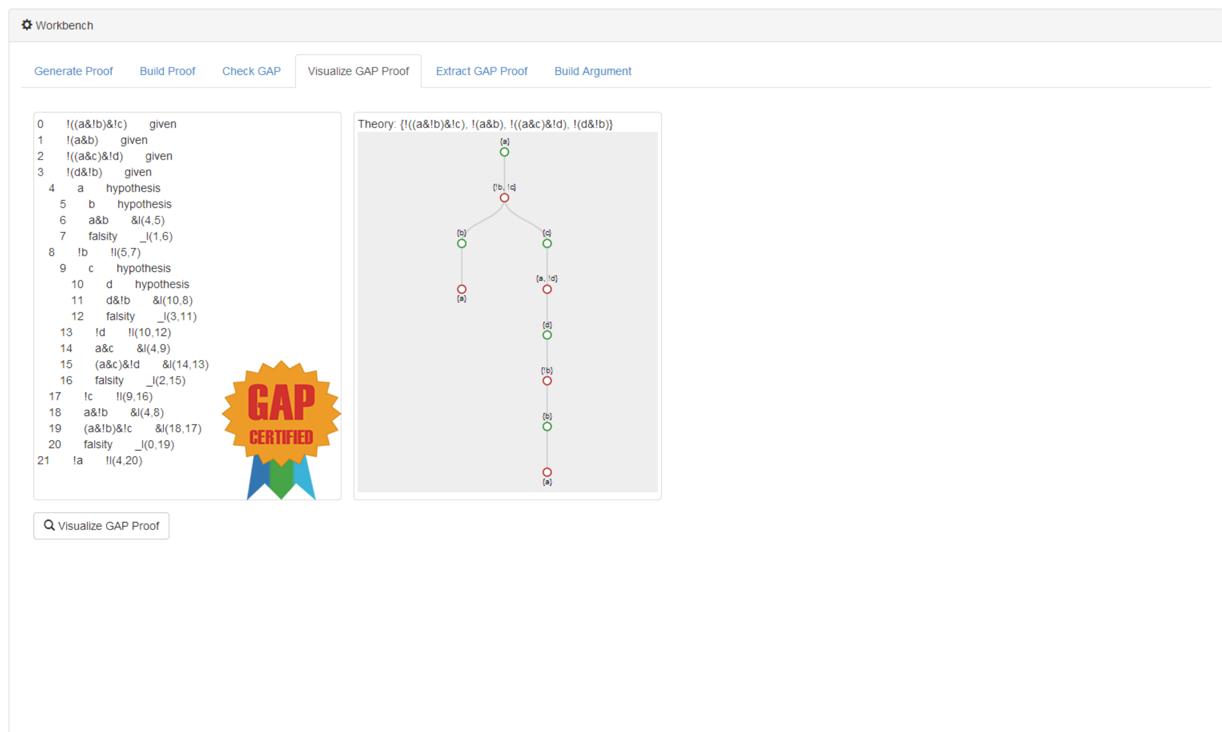

Figure 10.8: The Visualize GAP tab with a given verified proof and its corresponding argument

Extract Proof Tab

This tab is used to transform an argument back to a proof using the extraction algorithm in chapter 8. This tab works much in the same way as the previous tab but with the input and output reversed. The user can drag a visualization (argument) onto the box on the left from the clipboard, and click on the "Extract GAP Proof" button. A proof will then be extracted from the argument and placed onto the right placeholder so that they can be viewed side-by-side. Finally the user can drag the extracted proof onto the clipboard in order to save it. An argument and its corresponding proof can be seen together in Figure 10.9.

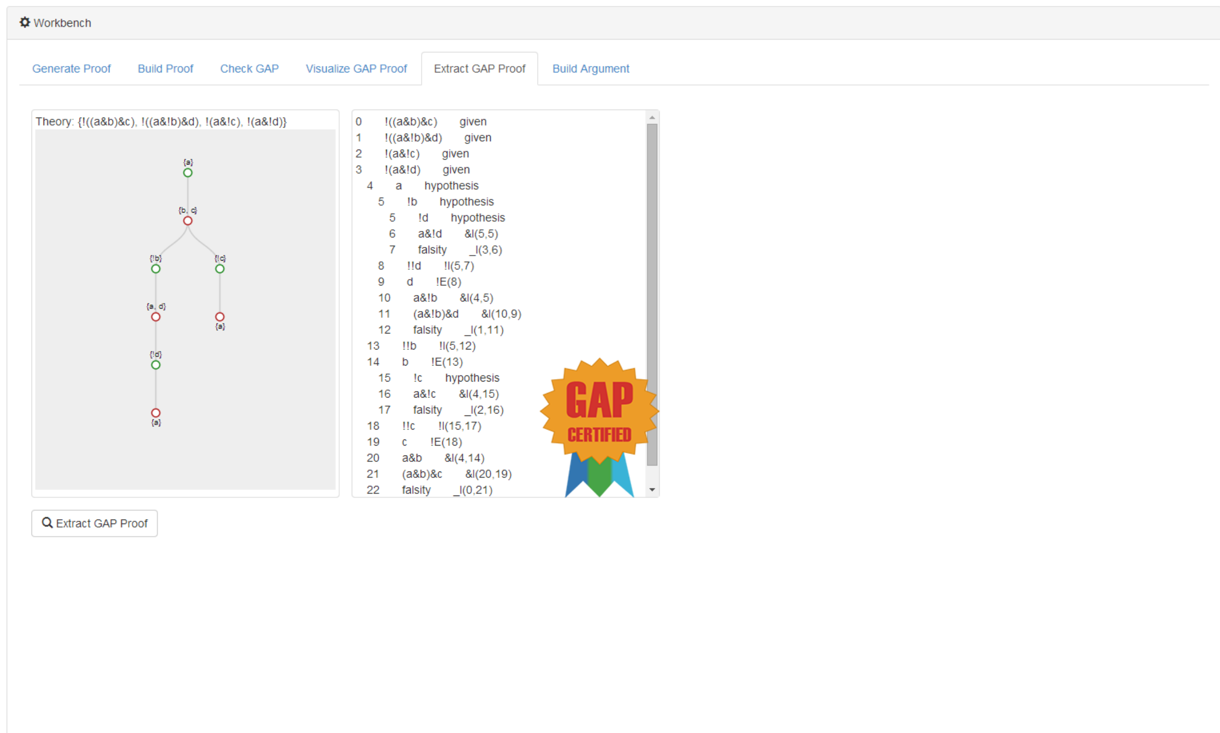

Figure 10.9: The Extract GAP Proof tab with a given argument and its corresponding verified proof

Build Argument Tab

This functionality is only available in the client module. The core module does not have an equivalent function. It is described in its own separate chapter in chapter 12.

10.2.3 Options

Options can be accessed via the top-right dropdown menu above the clipboard as shown in Figure 10.1. A summary of the options is given below:

Option	Description
Import to Clipboard	Used to import exported items back to the Clipboard
Export Clipboard	Exports the current items in the Cliboard
Help	Opens a help page in a new tab with the user manual
About	Displays a pop-up with acknowledgments

The importing-exporting facility is described below.

Exporting from the Clipboard

The data stored in the clipboard can be exported as strings of JSON text and can be saved/distributed/etc. Clicking on the ”Export Clipboard” drop-down menu (or button on the clipboard) will export the entire contents of the clipboard. The data will be opened in a new window. If only one of the items on the clipboard needs to be exported, simply drag that item onto the clipboard's ”Export Clipboard” button. Note that exporting items from the clipboard does not delete them from it.

Importing to the Clipboard

The text exported by the ”Export Clipboard” drop-down menu or button can be imported back into the clipboard by clicking on the ”Import to Clipboard” drop-down menu. A pop-up appears with an input field that allows to paste the JSON text in (see Figure 10.10). Clicking on the ”Import to Clipboard” button will validate the pasted text and if the validation succeeds, the parsed data will be added back to the clipboard. If the validation fails however, the input field will turn red, giving the user the chance to fix the errors and try again.

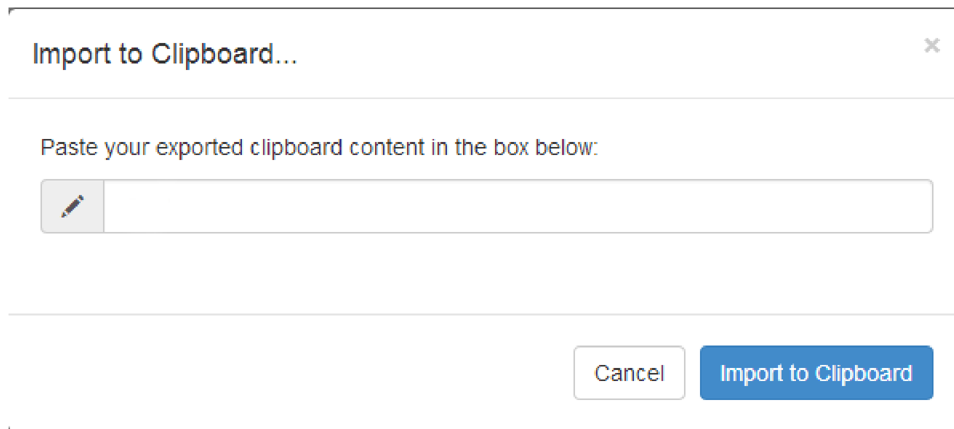

Figure 10.10: A pop-up shows when the user clicks the ”Import to Clipboard” button on the options dropdown menu

10.2.4 Logic Syntax and Natural Deduction in Client Module

This section introduces the syntax used for propositional logic by the client, and how a natural deduction proof is represented.

Symbols for Propositional Logic

In order to make the application appear the same on different devices with different character encoding abilities ASCII characters were used to represent logical connectives. Figure 10.11 summarizes the different symbols used and what they represent.

The same symbols are used by the parser as well.

Entering Propositional Formulas

Single propositional formulas can be entered by typing in the formula using the symbols shown above. This is usually required when entering a goal to be proven. A list of propositional formulas is a comma-separated list of formulas. Space characters are ignored. Lists

Symbol	Meaning
!	Negation
&	Conjunction
|	Disjunction
− >	Implication
_	Falsity (Contradiction)
()	Precedence (Grouping)

Figure 10.11: The logical symbols used by the client application

of formulas are usually required when specifying the theory (which might consist of more than just one formula).

Natural Deduction Format

With the exception of the symbols used to indicate logic constructs, natural deduction is represented in the same way as it is represented in this paper. In section 2.2 a summary of the rules used by the client as well as an example on how the natural deduction proofs look like are provided. The boxes (sub-derivations) are represented with an indentation rather than an actual drawn box. The greater the indentation, the deeper the sub-derivation is.

10.3 Alternatives

There is a wide variety of alternate implementations for the user interface. Among them exist the following:

- Java Swing[2]
- Prolog XPCE[3]
- GTK+[4], QT[5], and other similar application frameworks or widget toolkits

Swing is a Java widget toolkit. It can be used to create platform-independent user interfaces written in Java, much in the same way as HTML, CSS and JavaScript can be used to make cross-platform applications. Java tends to be quite verbose however, leading to more complicated code sometimes. The major difference lies in the fact that Swing is used in Java applications, whereas JavaScript is used in web applications. Swing could have been used to create an interface that is packaged together with the server, merging the two roles in one application (ie there would be no physical separation between the client and server), if a Prolog binding library were used (like JPL). This would make the use of a web browser unnecessary.

Prolog XPCE is a SWI-Prolog library for creating graphical user interfaces. This would be the most preferable option since the interface could have imported the core module directly, rendering the server obsolete. Prolog code tends to be very concise, unlike graphical user interface code in general. However, the look and feel of the interfaces produced by XPCE is a bit dated, and more modern alternatives are out there that could offer a better overall experience.

[2]http://en.wikipedia.org/wiki/Swing_(Java)
[3]http://www.swi-prolog.org/packages/xpce/
[4]http://www.gtk.org/
[5]https://qt-project.org/

GTK+, Qt and other similar widget toolkits have been proven by various well-known applications for their wide variety of interface elements and overall flexibility. These libraries tend to be very fast, since they are written in C or C++. This however, makes it difficult to produce bug-free interfaces because of their complexity and code size. There have been efforts to create bindings to other high-level programming languages such as Python (PyQt[6] and PyGTK[7]) that drastically simplify the making of graphical user interfaces with the aforementioned libraries.

The major advantage of building a web application, and the major disadvantage of desktop applications, is deployment. For web applications, updates do not need to be pushed down to the client, and the client itself does not need to seek updates from a server. That is because a web application is served by the server directly, and so application updates can be hot-swapped whenever they become available. A simple page refresh is all that is required. Thus the major advantage of choosing HTML, JavaScript and CSS is that it allows for the option of hosting the project on a public webserver, and allowing the client to be sent to the user's web browser as a web application. This is not possible with the aforementioned alternatives.

In addition, web development has become very fast, with different toolkits and libraries available that speed up development and significantly increase productivity. Web applications can now run at very high speeds due to sophisticated JavaScript engines incorporated by web browsers. In this respect, the current solution does not lack in terms of performance, productivity or appearance, and has the distinct advantage of quick and easy distribution.

[6]http://www.riverbankcomputing.com/software/pyqt/intro
[7]http://www.pygtk.org/

Chapter 11

Proof Builder

Experience from repeated usage of the theorem prover in order to generate proofs (step 1 and 2) has shown that sometimes the amount of proofs generated using the theorem prover directly can be overwhelming. Most of the time, a particular proof was required for further study, and going through the generated results was found tedious. Thus a need for a proof builder arose (step 1+), so that when a particular proof is required, it can be input directly with the help of the proof builder.

11.1 Motivation for Client Side Implementation

The proof builder is completely client side. There are several reasons for this decision.

Firstly, since this is an aided construction, visual feedback should be frequent and immediate. Requesting validation from the server would be both wasteful in terms of resources and possibly time-consuming as the builder would stall until the response comes back from the server.

Hence by building this feature client side, unnecessary back-and-forth communication is avoided.

Since the proof builder is completely contained in the client, it can be easily integrated into the GUI for a seamless experience that can be used offline as well.

Since this feature involves a gradual construction of a proof with intermediate user input, keeping the current state is essential. Since the proof builder is client side, it does not communicate with the server and therefore the server does not need to hold any session information and can indeed be completely stateless.

11.2 Features

The proof builder features a very simple, easy to use interface. The interface is made of two input fields where the user can input the theory and goal of the proof, a button that signifies the start of a new proof, a placeholder for rendering the current state of the proof, and finally, one more input box for entering commands to guide the construction of the proof.

The proof builder makes use of the client's text input parser. The parser can report incorrect input and validate the user's commands to the proof builder. For example, if the user enters the command $a\&b;\&I(5)$ (which means that a new step is to be added where the derivation is $a \wedge b$ and the reason is $\wedge$-introduction from line 5), the parser will detect that a $\wedge$-introduction rule requires two line references instead of the one provided (one for

each of the two subformulas of the conjunction) and will make the command input box glow red to indicate an error.

Even if the input is syntactically correct, mistakes can still happen. Assuming correct syntactically input that passed the parser stage, the natural deduction rule checker can be used to verify the command. Line references will be checked, and the rule application will be verified as well. For example if the user enters an erroneous command $a\&b;\&I(5,9000)$ and the current step to be added has line number 5, then this is picked up by the rule checker. The rule checker can work out if the derivation was made correctly. For example if the given input is $a\&b;\&I(5,3)$, the rule checker will see that this is supposed to be an application of the $\wedge$-introduction rule. This means that the derivation should be a conjunction of two sub-formulas. Those subformulas (a and b in the example) should have previously been derived on the lines referenced by the command (lines 5 and 3 in the example). If any error occurs, the command input box glows red to signal incorrect input. The effect of this is that proofs generated by the proof builder are sound.

As mentioned above, feedback is given immediately after the user enters a command. Even if the feedback is not very specific, it is adequate for the user to understand what went wrong with just a brief inspection of their input. Since feedback is available immediately after each command entered, and because no progress can be made if the command is not valid, the user can easily identify the mistake.

11.3 Usage

The user starts by navigating to the "Build Proof" tab of the Workbench. There, two input boxes can be found. In the first one, a comma-separated list of formulas can be entered for the theory. In the second input field, a single formula may be entered for the goal of the proof. This is used by the proof builder to identify when the proof is complete. The user clicks on the "Build Proof" button and the proof construction begins if the theory and goal parse correctly. If not, then whichever field contains erroneous input is highlighted. If the parsing succeeds then the command input field is enabled and is ready for use. The theory is already filled in with "given" being the reasons of the steps of the proof. Figure 11.1 shows this state.

The user can type commands in the command input field below the rendered proof under construction and enter them by pressing the Enter key. If the command is accepted, it is executed. Otherwise, the command input field glows red to indicate an error. This is shown in Figure 11.2. There are three supported commands:

- New step: adds a new line at the end of the proof under construction
- Delete last step: deletes the last line of the proof
- Delete last n steps: deletes the last n steps of the proof

The first command is of the format $[formula];[reason]$, where $[formula]$ is any valid formula and $[reason]$ is any of the supported reasons listed in Figure 11.3. These can be compared to subsection 2.2.1. If the step command is a new hypothesis, a box is automatically opened. Boxes are shown by an indentation of the lines of the proof proportional to the level of nesting. If the step command describes a $\neg I$ step, then the current box is closed and the command is appended one level up.

The second command is of the format $--$. This deletes the last line of the proof. If that line was a $\neg I$ or $\to I$ rule then the preceding box is now re-opened and can be closed by issuing another $\neg I$ or $\to I$ step command.

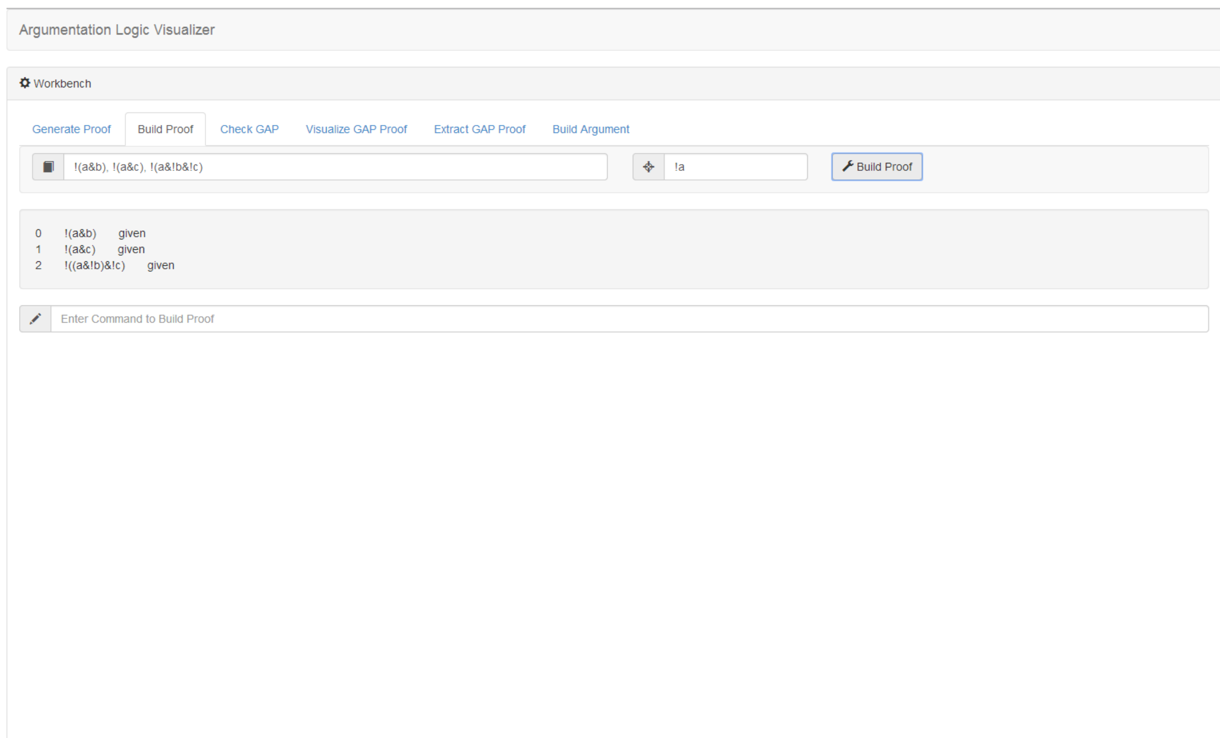

Figure 11.1: The proof builder already fills in the theory as steps for the proof that the user is about to build

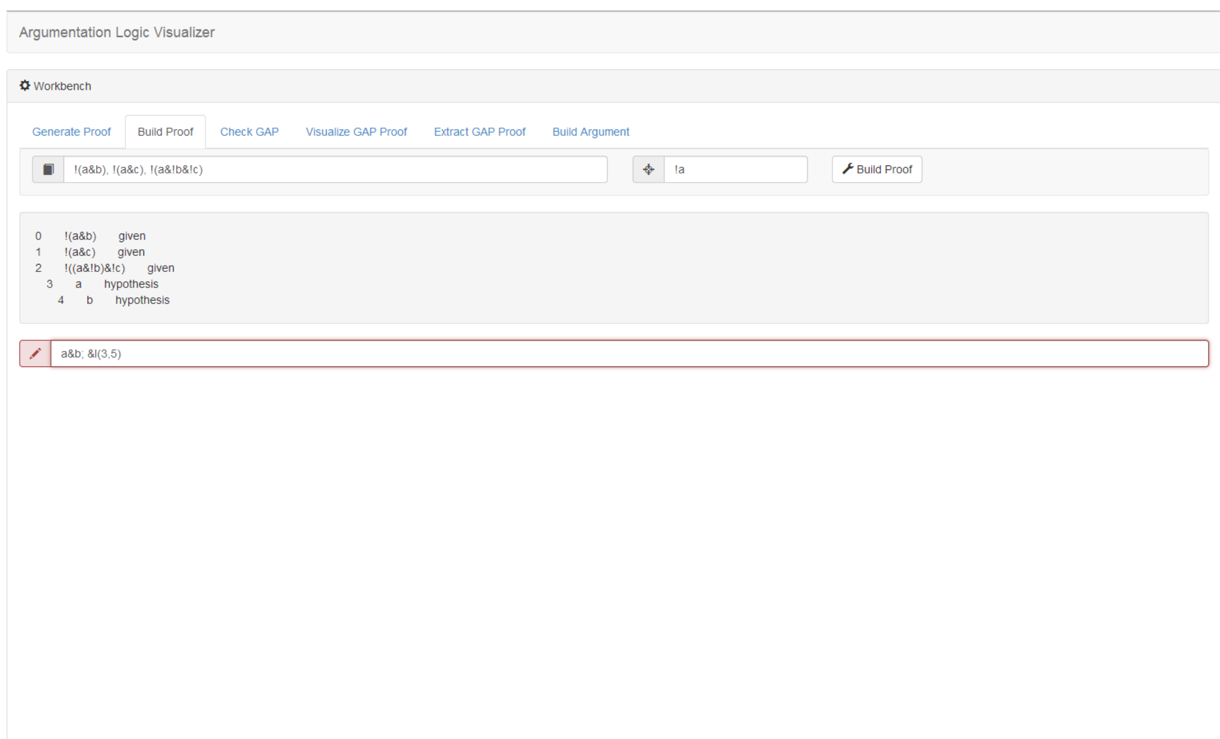

Figure 11.2: The proof builder indicates an error when the user input is incorrect

Rule Format	Rule Name	Description
$\&I(\#, \#)$	$\wedge I$	Conjunction of the formulas indicated by the two referenced lines
$\&E(\#)$	$\wedge E$	The left/right hand side of a conjunction indicated by the referenced line
$- > I(\#, \#)$	$\rightarrow I$	Implication of two formulas at the start or end of preceding box indicated by the referenced lines
$- > E(\#, \#)$	$\rightarrow E$	Right hand side of implication derived by an implication and its left hand side indicated by the referenced lines
$_I(\#, \#)$	$\bot I$	Contradiction introduced by a formula and its negation as indicated by the referenced lines
$_E(\#)$	$\bot E$	Any formula derived from a contradiction indicated by the referenced line
$!I(\#, \#)$	$\neg I$	The negation of a hypothesis and a subsequent contradiction in a box as indicated by the referenced lines
$!E(\#)$	$\neg E$	The derivation of a formula by removing a double-negation from a formula indicated by the referenced line
hypothesis	*hypothesis*	A hypothesis placed at the beginning of a box
$check(\#)$	*reiteration*	Restating a previously derived formula

Figure 11.3: Supported natural deduction rules for the proof builder. A # represents a line number

The last command is of the format $-[number]$, where $[number]$ is any non-negative number. The last n steps are deleted where n is the number specified. This command works by repeating the second command n times so the effects are essentially the same.

The user can enter commands to fill in the proof. When the last step in at the top level (not inside a box) and it matched the goal entered at the very start, then the proof is complete and the command input box is disabled. The rendered proof now becomes draggable and the user can subsequently drag the proof onto the clipboard to save it. Underneath the user interface, the proof has the same exact representation as a proof generated by the theorem prover, and can thus be used to check whether it follows the Genuine Absurdity Property, be visualized (if it does indeed follow the property) and so on.

Chapter 12

Argument Builder

In order to complement the proof builder (chapter 11) an argument builder was created so that specific arguments can be drawn directly, without first having to create the corresponding proof in step 1 (theorem prover) or 1+ (proof builder), then checking it in step 3 (Genuine Absurdity Property check) or 3+ (extended property check), and then visualizing it in step 4 (visualization algorithm).

Much in the same way as the proof builder works, the argument builder provides immediate feedback to the user during the construction of the argument. Unlike the proof builder however, the argument builder is not completely client side. This is due to the fact that the theorem prover needs to be used in order to justify attacks claimed by the user.

The data sent to the server is minimal however, since the only communication between the client and the server when using this builder is a query about whether the supplied theory and given attack claim causes a contradiction with the defense, and does not cause a contradiction without it. The latter is used to check that the actual attack makes sense in that it attacks the defense, and not the theory itself.

Even though the argument builder makes use of the server, all of the state (the theory and the argument under construction) is held locally. This allows the server once more to be stateless. This means that the server does not need to remember any sessions and connections from the clients, greatly simplifying the overall architecture and future maintenance of the project.

The argument builder is step 4++, as discussed in the introduction and section 3.1.

12.1 Features

The argument builder makes use of a simple user interface, using elements from other parts of the overall interface. It consists of two input fields for the theory and the first argument, a thumbnail which shows the argument under construction and an attack input field, where the user can specify their attacks.

The argument builder benefits from the same parser that the proof builder and the rest of the client use, which can check that the theory, starting argument and subsequent attacks are syntactically correct.

At the same time, the argument builder uses a $NACC$-semantics checker, which checks the valid attacks made by the user and automatically ends the attack-defense chain when the attacks form terminal nodes (leaves).

The argument builder features automatic generation of defenses, and termination of the construction of the argument if all branches lead to terminal attacks.

12.2 Usage

The construction of an argument involves a mini-game where the user gives the argument builder the theory and initial argument (the first ”defense”, or the first argument of the ”proponent”). Then it is the job of the user to attack the computer, specifying attack claims. Claims that form good attacks, are accepted and added to the tree. The ”proponent” (the computer) then automatically generates the defenses, since they are very well predictable and specified (section 2.3.2, Defense Against an Attack). When all branches have been closed off (blocked by a terminal attack), the mini-game ends and the user can save the constructed argument.

The user starts off by entering the theory and argument that should be defended by the computer. The user then clicks the Build Argument button in order to begin the construction of the argument. If an error in the input is detected by the parser, the offending input box is highlighted. Upon the beginning of the construction, the first argument of the proponent is visualized. The attack claim input field is then enabled and the program awaits the user’s attack. This state is shown in Figure 12.1. The user can make an attack claim by first specifying what the attack is going to be. In order to do that, the user enters the attack in the attack input box and presses the Enter key on the keyboard. If the attack makes sense (does not contradict the theory itself), then it is accepted, otherwise the input field glows red to indicate a bad attack. This is similar to entering an incorrect command in the command input field of the proof builder.

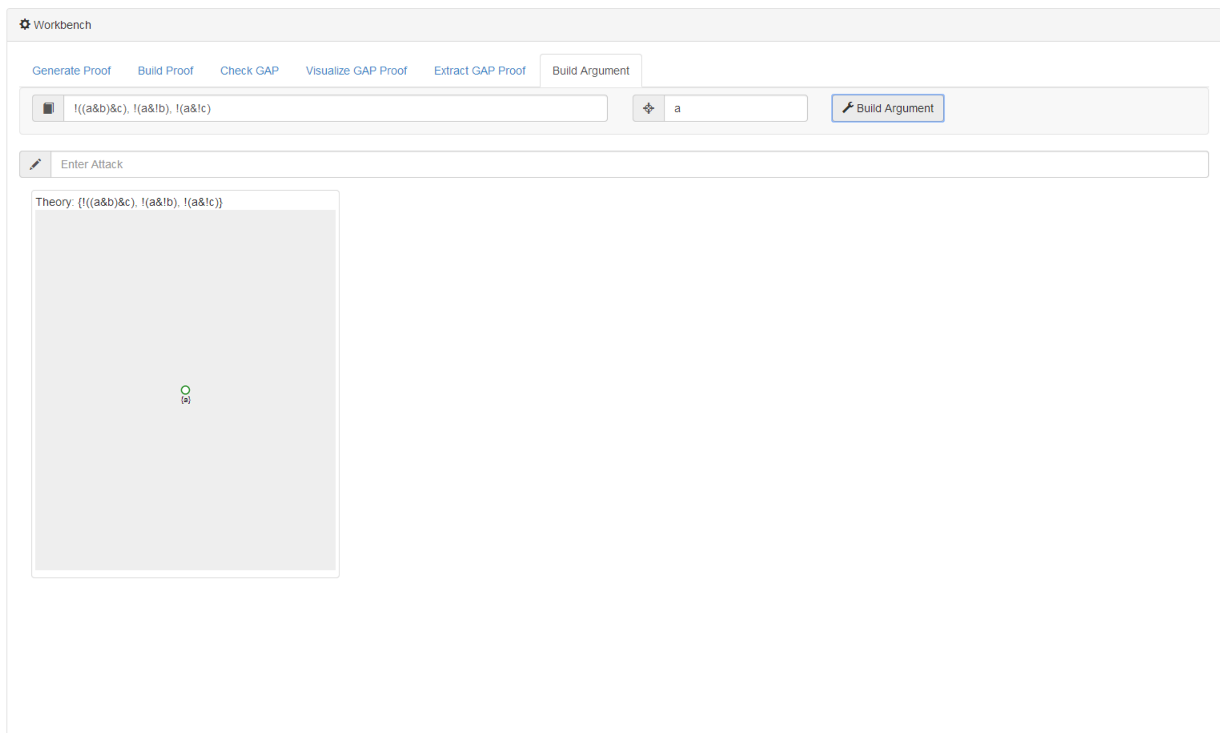

Figure 12.1: The argument builder draws the initial argument and enables the attack input field awaiting for the user’s command

Green nodes represent defenses, and these are generated by the computer. Red nodes indicate attacks, and these are added by the user. If the attack is accepted, then a blue node appears on the bottom left corner of the visualized argument under construction to indicate an attack claim. This state is shown in Figure 12.2. The user can now drag

the attack claim (blue node) onto eligible nodes in the argument tree. Eligible nodes are leaf defenses. As soon as the user starts dragging the blue node, all eligible nodes will be indicated by a faint red drop zone, where the user can drop the blue node. A bright red temporary link will also link the attack claim and defense if the attack claim is hovering over its drop zone. This is shown by Figure 12.3. The user can let go of the blue node to issue the attack. The attack is checked that it reaches a contradiction when combined with the defense it is attacking. If that is the case, the tree is redrawn and the attack claim is "set in stone" as it is drawn underneath the defense as a red node.

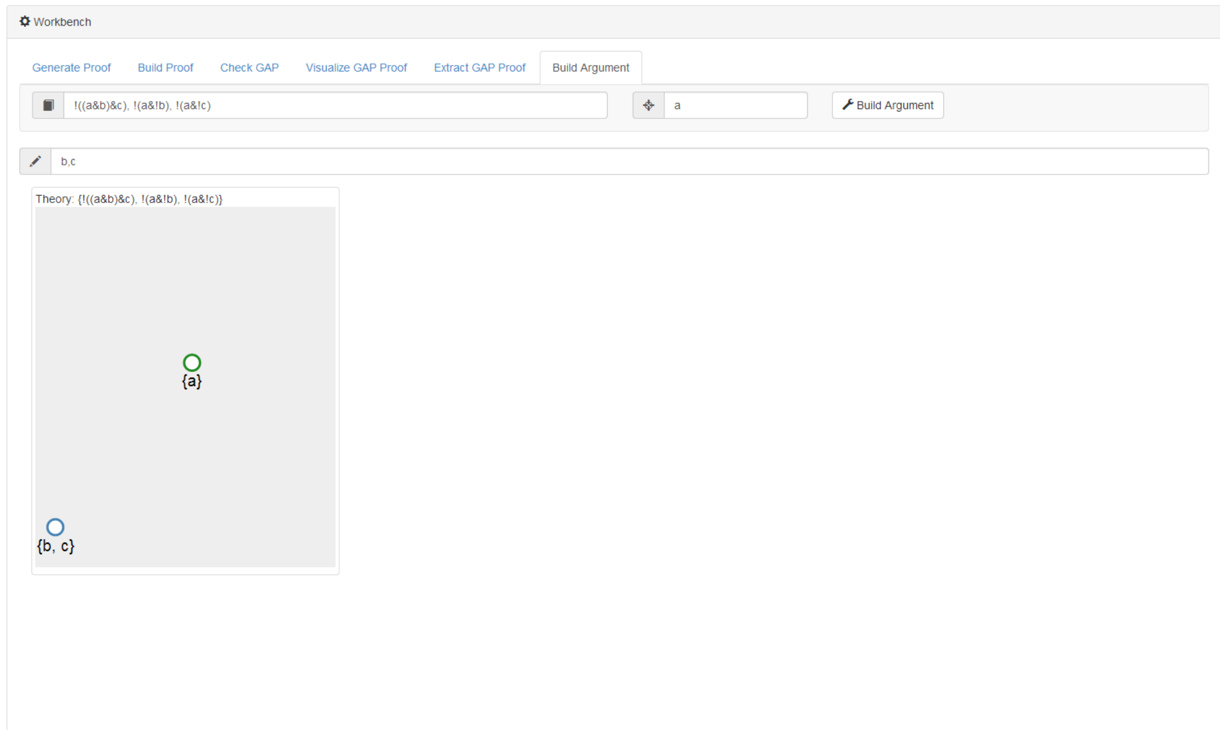

Figure 12.2: The argument builder draws the initial argument and enables the attack input field awaiting for the user's command

When the user makes a successful attack, the computer checks if the attack is a subset of the parent defenses, according to the $NACC$ semantics of Argumentation Logic (section 2.3.3). If it does, it is considered a terminal attack. This means that no defenses are generated to defend against that attack and hence that branch of the tree is effectively closed off (Figure 12.4). If the attack is not a terminal attack, then defenses are generated according to the $NACC$ semantics once more. The defenses are predictable, since the definition of a defense is to take the opposite stance of a component of the attack (section 2.3.2).

The mini-game ends when all branches are closed off and no unattacked defenses remain. The attack claim input field is then disabled and the constructed argument can be dragged to the clipboard in order to be saved. The arguments generated by the argument builder have the exact same format as those generated by the visualization algorithm (chapter 7) and can be used in the same way.

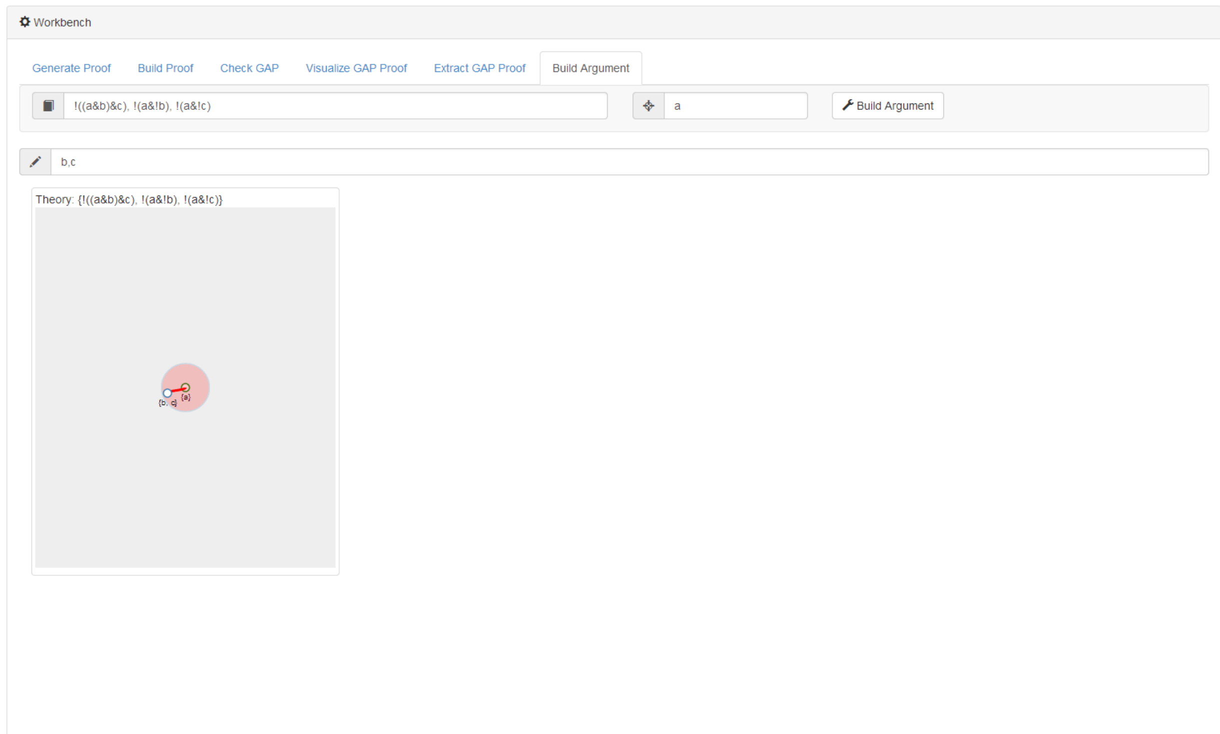

Figure 12.3: The user is about to attack the computer's argument; a red link shows the node the attack will be against if the user drops the attack node

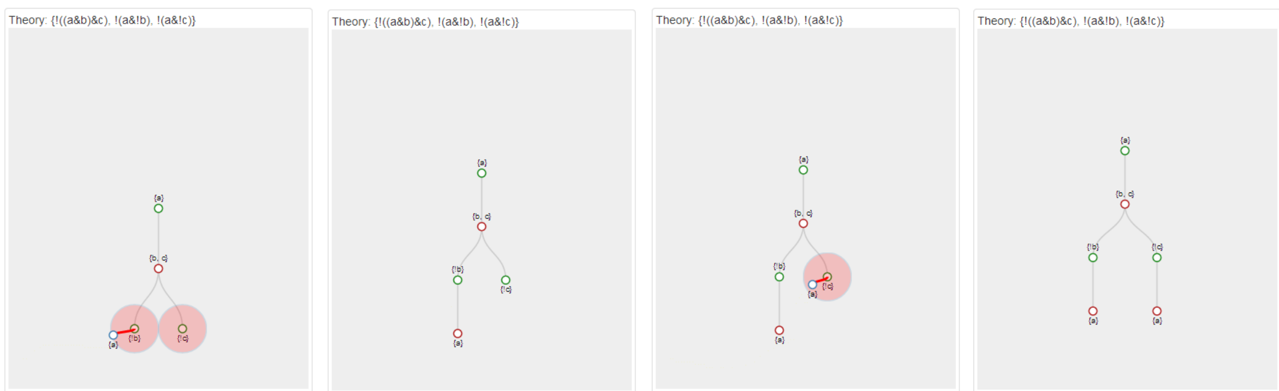

Figure 12.4: The user gives a terminal attack on the left branch, leaving only the right branch open; after a terminal attack on the right branch, the argument is complete

12.3 Generating Arguments Automatically

Originally, an argument generator that works in a similar way to the theorem prover (chapter 4) was envisioned, which could be given a theory T and a starting argument a and generate all arguments proving $NACC^T(\{a\}, \{\})$. The idea was dropped as a lot of computation would be necessary to generate attacks.

In order for an attack to be generated, the argument generator would have to go through the powerset of the language and try each subset in order to test if it forms a valid attack. Defenses, given a valid attack, could be generated easily since they always take the opposite (negation) of the attack components.

Consider a small example with theory $T = \{\neg(\alpha \wedge \neg\gamma), \neg(\alpha \wedge \beta \wedge \gamma), \neg(\neg\beta \wedge \delta), \neg(\alpha \wedge \neg\beta \wedge \neg\delta)\}$. The language for this theory is $L = \{\alpha, \beta, \gamma, \delta, \neg\alpha, \neg\beta, \neg\gamma, \neg\delta\}$ and therefore the powerset of the language $\mathcal{P}$ contains 256 subsets. Almost all of these subsets will need to be processed to check whether they constitute a good attack. Some of these subsets will not constitute good attacks, for example $S_1 = \{\delta, \neg\delta\}$ (it contradicts itself) or $S_2 = \{\alpha, \neg\gamma\}$ (it contradicts the theory itself) and thus further calculations will be necessary to weed them out. This entire procedure needs to be repeated n times for just one argument with n number of attacks.

It is therefore only a viable solution if a good strategy can be devised that can drastically prune the search space of the possible attacks. This, consequently, remains a topic for future work.

Chapter 13

Evaluation

Evaluation of this project cannot be based on user feedback. The implementation is not addressed to a general audience, hence statistics like number of downloads on an online store, or feedback and ratings on a particular website do not apply. At the same time, the implementation is not a piece of software that is meant to run something in a very optimized way, so statistics like frames per second, time intervals, etc are of no use either. It is difficult to evaluate the project since it is based one Argumentation Logic, that is not yet widely known. The project is mostly exploratory. Ideally the evaluation for the project should be written one or two years after its submission.

The overall project will therefore be evaluated on its correctness, stability and on its contributions, as well as on whether it fulfilled the objectives it set out to complete. This evaluation section will start its criticism by evaluating the different components, procedures and algorithms of the project and move up to its aims and objectives in general.

13.1 Theorem Prover

13.1.1 Correctness

In terms of correctness, the theorem prover should be both sound and complete regarding its implemented logic constructs. It uses the Carnegie Mellon Proof Tutor as a starting point which is itself sound and complete. The deviations do not lead to unsound proofs, and completeness should still be maintained. As with any piece of software however, there exists a possibility of errors. No software is perfect, and hence the theorem prover may suffer from bugs. In order to be completely sure that the software works, formal verification methods should be employed. However, such methods take too much time and require a great deal of effort. The theorem prover is critical to the project, however it is not its main focus. The main focus remains the Argumentation Logic. In place of formal verification methods, testing has been applied in order to try and eliminate as many errors as possible and to provide a good starting point for when upgrades or other changes are made to the current version of the prover.

In order to test the implementation plunit[1] was used as the unit test framework of choice. Thankfully, the team behind SWI-Prolog created this unit test framework that works quite well with their Prolog implementation. Several tests were laid out both trying to test that the theorem prover can prove goals that are provable (given the necessary theory) and that it cannot (and should not) prove goals that can't be reached (given again

[1] http://www.swi-prolog.org/pldoc/package/plunit.html

the necessary theory). For the sake of brevity, the test suites will not be included in the report, but are available in the source code of the project.

13.1.2 Performance

The theorem prover works well in general. It does sometimes seem to get slow on complicated theories and proofs that exhibit a very large search space. The purpose of the theorem prover is to be able to provide "all" proofs that prove a goal given a theory. However it is very difficult to exactly specify what "all" means. Obviously there can be infinite proofs for any goal given to the theorem prover (at the very least, the theorem prover could take any atom from the theory and start building an infinite conjunction chain, chaining together that very atom to produce infinite proofs). Instead, the theorem prover tries to produce all (close to) normal proofs. This means that it tries not to take detours by imposing restrictions in its ruleset (see restricted rules in subsection 2.2.4). This ensures that a finite set of proofs can be generated for a particular goal. Still, that set could be very large, as any (allowed) permutation in a Fitch-style natural deduction proof can be considered a completely different proof. The rest of this subsection tries to quantify the performance of the theorem prover.

Execution Time

Consider Figure 13.1, which shows a table with a variety of theories and goals that need to be proven. The theorem prover was tasked to prove the goals using the accompanying theories, and its runtime was measured using the SWI-Prolog profiler[2]. The runtime for each of those executions is small, suggesting that just finding a proof of any kind is not a big job for the theorem prover.

#	Theory	Goal	Runtime
1	$\alpha \wedge \beta \wedge \gamma$	β	0.00s
2	$\neg\neg\neg\neg(\alpha \wedge \neg\neg\beta)$	$\beta \wedge \alpha$	0.00s
3	$\neg\neg\alpha, \alpha \wedge \beta \rightarrow \gamma$	$\beta \rightarrow \gamma$	0.00s
4		$\alpha \rightarrow \beta \rightarrow \alpha \rightarrow \alpha$	0.00s
5		$\neg(\alpha \wedge \neg\alpha)$	0.00s
6	$\alpha \wedge \beta \rightarrow \bot, \neg \rightarrow \bot$	$\alpha \rightarrow \bot$	0.00s
7	$\neg(\alpha \wedge \beta), \neg(\neg\alpha \wedge \gamma), \beta \wedge \gamma$	$\neg\delta$	0.00s
8	$\alpha \wedge \neg\beta \rightarrow \bot, \beta \wedge \gamma \rightarrow \bot, \alpha \wedge \beta \wedge \neg\gamma \rightarrow \bot$	$\alpha \rightarrow \bot$	0.00s
9	$\alpha \wedge \delta \rightarrow \beta, \delta \rightarrow \alpha, \neg(\delta \wedge \gamma)$	$\alpha \wedge \beta \rightarrow \gamma \rightarrow \neg\delta$	0.02s
10	$\neg(\alpha \wedge \neg\beta \wedge \neg \wedge \gamma), \neg(\alpha \wedge \beta), \neg(\alpha \wedge \gamma \wedge \neg\delta), \neg(\delta \wedge \neg\beta)$	$\neg\alpha$	13.47s

Figure 13.1: A list of ten theories and goals proven by the theorem prover along with their execution time. The theorem prover need only find one proof

However, here is the same table again, but this time, the runtime measured is the total runtime to find "all" proofs for the given theory and goal. This table is shown in Figure 13.2. From this and the previous table, it can be inferred that finding one proof is fairly easy, but exhausting the entire search space is an intensive job. The same would apply for when trying to check whether something is non-provable, as required by step 3 (checking for the Genuine Absurdity Property).

[2]http://www.swi-prolog.org/pldoc/man?section=profile

#	Theory	Goal	Total Runtime
1	$\alpha \wedge \beta \wedge \gamma$	β	0.00s
2	$\neg\neg\neg\neg(\alpha \wedge \neg\neg\beta)$	$\beta \wedge \alpha$	5+m
3	$\neg\neg\alpha, \alpha \wedge \beta \to \gamma$	$\beta \to \gamma$	4.52m
4		$\alpha \to \beta \to \alpha \to \alpha$	0.02s
5		$\neg(\alpha \wedge \neg\alpha)$	0.00s
6	$\alpha \wedge \beta \to \bot, \neg \to \bot$	$\alpha \to \bot$	0.08s
7	$\neg(\alpha \wedge \beta), \neg(\neg\alpha \wedge \gamma), \beta \wedge \gamma$	$\neg\delta$	5+m
8	$\alpha \wedge \neg\beta \to \bot, \beta \wedge \gamma \to \bot, \alpha \wedge \beta \wedge \neg\gamma \to \bot$	$\alpha \to \bot$	5+m
9	$\alpha \wedge \delta \to \beta, \delta \to \alpha, \neg(\delta \wedge \gamma)$	$\alpha \wedge \beta \to \gamma \to \neg\delta$	5+m
10	$\neg(\alpha \wedge \neg\beta \wedge \neg \wedge \gamma), \neg(\alpha \wedge \beta), \neg(\alpha \wedge \gamma \wedge \neg\delta), \neg(\delta \wedge \neg\beta)$	$\neg\alpha$	5+m

Figure 13.2: A list of ten theories and goals proven by the theorem prover along with their execution time. The theorem prover finds all proofs for each of the given theory and goal

The two figures, Figure 13.1 and Figure 13.2, imply that each proof by itself may not be too time taxing, however many of them stacked together can impose a huge execution time. This implies that the theorem prover needs to explore a large search space. As a future improvement, the theorem prover can use stricter rules in order to limit the size of the search space it needs to explore.

The theorem prover is quite complex in that every rule has a chance of calling other rules, making very long mutually-recursive chains that are difficult to measure. It is difficult to measure precisely how much time each rule spends working, because it is difficult to define what it means for a particular rule to be working, since the rules will most likely call others. As an example, imagine a proof where the $\bot IE$ rule called the $\wedge I$ rule and that rule took 5 minutes before it returns. Can it be said that the $\wedge I$ took 5 minutes to execute, or is this execution time attributed to the $\bot IE$ who called it? What about the rules that $\wedge I$ called in order to prove its goal? Finding which rule is to blame, if any, is not straight-forward, especially when trying to compare their execution times, but there are some indications as to which is the most expensive.

The following list (Figure 13.3) was taken from a random (but representative, since more tests with similar results were observed) run of the theorem prover that run for around ten minutes. The list shows the execution time spent for each predicate for that run. Note that this execution time is the time spent inside the predicate, but not inside any children (callees) of the predicate. This is partly why the execution time for the rules is low. A huge chunk of the execution time is consumed by the predicate `member_/3`, an internal to Prolog predicate called by Prolog's `member/2` predicate, as seen in Figure 13.4.

Figure 13.4 shows the profile of `member/2`, which is highlighted in yellow. The items below it are the callees, predicates that `member/2` called in order to carry out its task. `member_/3` is the only child of `member/2`, which is why its "self" time is equal to its parent's "children" time. The items above `member/2` are the callers, predicates that called `member/2` as part of their execution. `m3/3` is a predicate that calls `member/2` twice, in order to check for the presence of a particular term inside two lists. Most of the times, this is used by the theorem prover in order to look up a term in both the current and the inherited contexts.

It is easy to see from these figures that `member/2` is used heavily by the theorem prover in since the different rules it implements really are pattern matching rules that match different terms from the context. It makes sense then, for `member/2` to take up a large amount of execution time. However so far the offending rule has still not been revealed. Figure 13.5

lists:member_/3	37.2%
falsitylx/4	7.0%
member/2	4.9%
not/1	4.5%
falsitylE/6	3.4%
append/3	3.1%
impliesE/6	2.3%
notEx/4	2.3%
andEx/4	2.3%
orE/6	2.2%

Figure 13.3: The profiler shows that most of the execution time is attributed to context lookups, as rules really are pattern matching rules

Details -- member/2

Time		Access		Predicate
Self	Children	Call	Redo	
1.8%	5.1%	76587443	0	m2/2
3.1%	32.1%	153513734	0	m3/3
4.9%	37.2%	230101177	0	**member/2**
37.2%	0.0%	228322804	18023275	lists:member_/3

Figure 13.4: Most of the execution time is attributed to `m3/3`, a predicate that calls Prolog's `member/2` twice to look up a term in the current and inherited context

gives a good indication of that.

Details -- m3/3

Time		Access		Predicate
Self	Children	Call	Redo	
0.0%	0.0%	1936	1357	andEx/4
0.0%	0.4%	837147	837070	andI/6
0.0%	0.5%	1216104	1216089	falsityE/6
0.1%	1.1%	2432084	2432084	orE/6
0.1%	1.0%	2432269	2432246	check/6
0.1%	1.2%	2432129	2432129	impliesE/6
0.1%	1.6%	2432083	2432083	falsityIE/6
0.6%	8.4%	20034887	20034716	not/1
1.0%	21.1%	44938694	44938627	falsityIx/4
2.0%	35.2%	76757333	76756401	**m3/3**
3.1%	32.1%	153513734	0	member/2

Figure 13.5: Most of the lookups are called by `falsityIx/4`, the forward rule that tries to find α and $\neg\alpha$ in the context (for any α) and thus derive a contradiction

Figure 13.5 is similar to Figure 13.4, but now `m3/3` is in the spotlight. What is interesting is the usage made by its callers. A great deal of execution time is done by the predicate `falsityIx/4`. This predicate is a helper predicate for the $\bot I$ forward rule (see section 4.1 for the rules used by the theorem prover). This rule tries to find two pairs α and $\neg\alpha$ for some α, and thus add a step $\bot$ indicating contradiction. This is logical, since this rule tries to join each bit of the context with the rest of the context, resulting in an intensive task. This can therefore be remedied in the future, either by removing this rule (if possible, while keeping the current level of expressiveness), or by optimizing this rule, or by integrating this rule in the other rules. This integration will allow each added step to check whether

its negation (or its subformula if it is already a negation) is already in the context, so that not all of the steps are checked against each other each time the rule runs.

Theory and Goal Complexity

The overall execution time of the theorem prover depends on the execution time of the different rules used, thus the overall execution time will be subject to how many times each rule was called and what the context was when it was called. These are dictated by the theory and goal. Thus measuring the theory and goal complexity can yield information as to what the execution time will be, if the given theory and goal are given to the theorem prover to work on.

An accurate description of the complexity may not be possible, since several variables may play a role as to how complex a theory and a goal are. The following list contains different variables that might play a role in grading a theory or goal as to their complexity:

- Language Complexity: this is the size of the language used in either the theory or the goal. For example a theory containing only α, β, γ may not be considered as complex as a theory containing the entire Greek alphabet.

- Structure Complexity: this is the complexity of the different parts of the theory or the goal in terms of how deep and varied their formulas are. For example, a $\alpha \wedge \beta$ appearing in the theory may not be considered as complex as a $\neg(\alpha\wedge\neg(\beta\wedge\epsilon) \rightarrow \gamma\vee\neg\delta)$ appearing in it. Different connectives could have a different complexity, or complexity could vary depending on how those connectives are used to make up a formula. For example, $\alpha \wedge \beta \rightarrow \gamma$ could bear a different complexity than $\alpha \wedge (\beta \rightarrow \gamma)$ because now the principle connective is a conjunction instead of an implication, even if both formulas have the same atoms and are of the same depth.

- Theory Length Complexity: this is the size of the theory. This measures how many elements are in the theory. Of course, this measure can vary. It could be said that $\{\neg(\alpha \wedge \beta), \neg(\gamma \wedge \delta), \neg(\alpha \wedge \delta)\}$ is larger than $\{\neg(\alpha \wedge \beta), \neg(\gamma \wedge \delta)\}$ (note that the previous two complexities are the same between these two theories) and is therefore more complex. However what happens in the case of $\{\alpha \wedge \beta, \gamma\}$ and $\{\alpha, \beta, \gamma\}$ or $\{\alpha \wedge \beta \wedge \gamma\}$? Should those carry different length complexity values? Perhaps this complexity should be counted after conjunctions are broken down to constituent parts.

- Search Space Complexity: this is the size of the search space that needs to be covered by the theorem prover in order to construct a proof. The search space is a consequence of the rules used by the theorem prover. For example if the theorem prover had no rules at all, then the search space would be empty regardless of the theory and goal since the prover can't get anywhere.

- Permutation Complexity: this is the complexity that measures permutations of the same theory. For example, a run with theory $\{\neg(\alpha\wedge\beta), \neg(\alpha\wedge\gamma), \neg(\alpha\wedge\neg\beta\wedge\neg\gamma)\}$ might prove harder than an equivalent run with theory $\{\neg(\alpha\wedge\neg\beta\wedge\neg\gamma), \neg(\alpha\wedge\beta), \neg(\alpha\wedge\gamma)\}$. This can be attributed to the fact that Prolog is a procedural language, and it picks the rules in a certain order. Certain permutations of the theory may lead to a proof faster than others depending on what order the rules are chosen in. This complexity bears no significance when "all" proofs are to be generated, since the entire search space will be covered regardless.

- `member/2` Complexity: this is the number of calls to the `member/2` predicate that the theory and goal would incur, since it is the most time-consuming predicate and is

heavily used by the rules (as seen in Figure 13.3 and Figure 13.4). This is perhaps the most accurate measure of the complexity of a theory and a goal, as it is directly related to execution time.

Other complexities can be defined as well. The above list is just indicative. Care must be taken so that during measurements of one type of complexity all other complexities remain constant. However, even this might not be enough. The different complexities may be inter-dependent, meaning that one change in one complexity can affect how another complexity varies. In order to fully analyze the complexities, a six-dimensional graph must be generated. Complexities that do not seem to affect execution time could be dropped, and correlations between the rest of the complexities should be calculated.

This reaches beyond the scope of this project, as the main focus is Argumentation Logic itself and not the theorem prover's optimal performance alone.

More on Theory and Goal Complexity

For the sake of curiosity mostly, an attempt to further investigate the the relationship between execution time and theory and goal complexity was made.

In order to investigate the effects of language complexity, the other complexities were kept constant as much as possible throughout the experiments. In order not to change the length complexity, the theory was kept at a constant length, hence redundancy was introduced, the nature of which will shortly become apparent. In order not to change the structure complexity, the same structures were used between all experiment runs. Perhaps the least intrusive way to introduce more language into a theory is to just place the new atoms in a conjunction. The experiment involves a conjunction of the letters of the English alphabet as the theory, and different permutations of this conjunction as the goal for each run. The theory was a conjunction of 26 terms. In the first run, all terms were letter a. Next run all terms were letter a except the second which was letter b, introducing b into the language. Third run had all 26 terms a apart from the second and third which were b and c, introducing c into the language, and so on. The length was kept constant at 26 terms, and the structure was constant at a 26-term conjunction. Permutation complexity for the theory was not applicable since the theory kept changing (not permutating). Search space complexity and `member/2` complexity could vary, since a larger language could allow for the same rules to be used, but with more options to choose from. Results suggested near-constant time, since about the same time was needed to complete each run. There was a slight increase in execution time as the language was becoming more varied, since more terms had to be expanded since redundancy was decreasing. However, things changed when "all" proofs were requested to be generated. The search space exploded and the theorem prover could not complete the work as too many proofs had to be generated. For atomic goals, the theorem prover could finish after a couple of minutes when asked to find "all" proofs.

Structure complexity plays an important role in the execution time of the theorem prover through the theory complexity, discussed next. Rules tend to act on the principal connectives of formulas. So to the $\wedge E$ rule, $\alpha \wedge \beta$ and $(\alpha \wedge \beta \rightarrow \neg(\delta \vee \epsilon)) \wedge (\neg(\zeta \rightarrow \eta) \rightarrow \neg\theta)$ are exactly the same in terms of complexity. They both are conjunctions of some sub-formulas. However, the difference is made after the rule acts upon those two formulas. In the former case, only two sub-formulas (atoms α and β) are gathered and the formulas is more or less of no more use. The latter case allows for more expansion, until it is broken down to much smaller parts. This adds more formulas to the theory, hence the length complexity discussed next increases. In other words, there's more things to do with a more structurally complex formula, not at first glance, but when it's broken down.

Theory length complexity seems to greatly affect the performance of the theorem prover, as this tends to greatly affect the search space (complexity). Adding negated formulas ($\neg(...)$) to the theory allows the $\bot IE$ composite rule to be applicable, which tries each of those negations in order to prove the goal. If there are more than one negation in the theory, then this rule can pick any of them and in the next run pick the other, thus the search space includes permutations of the negations in terms of the order they can be picked up and used. The more negations co-exist in the theory, the larger the search space tends to be. A similar effect can be created by implications in the theory. The $\rightarrow E$ rule gathers all implications and tries to prove the first subformula in order to derive and add to the theory the second. If negations and implications are present, then the search space includes combinations of those in terms of the order in which they are used to look for proofs. What adds to the search is the use of $\neg I$, $\rightarrow I$ and proof by contradiction rules. These rules add their hypothesis to the context of the search, essentially providing more options (negations and implications) for the other rules above to explore. This further increases the search space. As an example consider theory $\{\neg(\alpha \wedge \beta), \neg(\alpha \wedge \neg\beta)\}$ and goal $\neg\alpha$. The theorem prover, when asked to find all the proofs proving the goal from the theory, comes back with four proofs and near-instantly. For a theory $\{\neg(\alpha \wedge \beta), \neg(\alpha \wedge \neg\beta), \neg(\alpha \wedge \gamma), \neg(\alpha \wedge \neg\gamma)\}$ and the same goal, the theorem prover takes some time (about a minute) to come back with hundreds of proofs. The language was increased only slightly (introducing γ), and the structure complexity was kept constant (since the introduced terms were also negations of conjunctions of two atoms or one atom and a negated atom).

Permutation complexity can hinder execution time, if the order in which the different parts of the theory is chosen to produce larger and more complicated proofs, with more dead ends in the search space. Early versions of the theorem prover were very susceptible to this type of complexity, but with the introduction of heuristic for the $\bot IE$ composite rule, execution times for permutations of the same theory converge to the same value. The heuristic is explained in subsection 13.1.4. As an example, consider theory $\{\neg(\alpha \wedge \beta \wedge \gamma), \neg(\alpha \wedge \neg\beta \wedge \delta), \neg(\alpha \wedge \neg\gamma), \neg(\alpha \wedge \neg\delta)\}$ and goal $\neg\alpha$. Before the optimization, execution time for permutations of this theory ranged from instant to around one minute. After the heuristic is implemented, any permutation executes instantly. As mentioned in the description of the complexity, permutations do not impact execution time if all proofs are to be found, since the entire search space will be exhausted anyway.

Search space complexity is a measure of how big the search space is. This depends on the applicability of the rules used by the theorem prover, which in turn depends on the theory and goal. It is difficult to define, but is related to the perhaps more accurate `member/2` complexity. This complexity counts the number of calls made to this predicate during execution. Recall that most of the execution time is attributed to the use of this predicate by the rules. In order to estimate execution time from the theory and goal, a function needs to be found with a theory and goal pair as the domain and time as the range. It may be difficult or even impossible to find such a function, as the search space depends on the theory and goal which in turn depend on the rules and vice versa. Different applications of different rules unlock more context, which is not known in advance unless the rules are actually applied in some order. Therefore, it is possible that the execution time cannot by estimated, and that it can only be found by expanding the search space which essentially is running the theorem prover itself (in order to find the number of calls to `member/2`).

Comparison with AProS

AProS[3] is a great theorem prover with a multitude of features that makes it a great environment to work in. The project's history goes back to 1985, and from then algorithms and interfaces have constantly been created and improved. This is a project accomplished by many people over many years.

The current implementation uses Java to implement the theorem prover and a flexible graphical user interface. It sports many features, from importing and exporting proofs to slideshows and graphs of the search space. This implementation seems to be much faster than the implementation of this project's theorem prover, and is closed-source. However the theorem prover used in this project enables two features that are not present in AProS, since the focus is different between their intended usages.

The theorem prover implemented for the exploration of Argumentation Logic allows the ability to turn off certain rules, as required by the check for the Genuine Absurdity Property. AProS allows to modify the order in which rules are used, but not to remove them entirely. In addition, the theorem prover allows for "all" proofs to be found, whereas it seems that AProS stops when the first proof is found.

AProS is a great platform that could be modified (with the approval of its authors) to work for the purposes of Argumentation Logic, however, working on a large existing implementation might have consequences on how much time was left for the rest of the project which is arguably more important. Since AProS is built in Java, at the time of this writing, modifications could prove more time-consuming than building a Prolog application. This is because Java tends to be more verbose (which is the cost of having a great structure in a statically-typed language), whereas Prolog tends to be very concise.

13.1.3 Pruning

The next step could be to prune the search space and drop proofs that are not good. However, ambiguity is faced when it comes to defining "good" proofs. Consider the proofs in Figure 13.6 for defining the measure "good":

1	α	given
2	α	✓

1	α	given
2	$\neg\alpha$	hypothesis
3	$\bot$	$\bot I(1,2)$
4	$\neg\neg\alpha$	$\neg I(2,3)$
5	α	$\neg E(4)$

Figure 13.6: Pruning could potentially trim proofs of potential importance regarding Argumentation Logic

One could argue that the second proof is unnecessary and could be trimmed since it is just a detour from the first proof. However, it is sometimes these scenic routes that are the best routes, and in Argumentation Logic it is no different. The proof on the right happens to follow the Genuine Absurdity Property and can therefore be visualized (see chapter 7). If it were to be pruned, the remaining results would be of little use.

It is therefore necessary to devise pruning algorithms and optimizations that do not reduce the amount of generated proofs that follow the Genuine Absurdity Property, as they are the most valuable proofs. Argumentation Logic-compatible pruning and optimizations remain part of future work.

[3] http://www.phil.cmu.edu/projects/apros/index.php?page=overview

13.1.4 Optimization

One optimization that was implemented that does not change the number of generated proofs is an optimization that grades all of the negated formulas in the current context of a proof, when the theorem prover decides to use the $\bot IE$ rule. This rule gathers all negated formulas and tries to prove their positive sub-formulas in an attempt to prove the goal using a contradiction. The optimization tries to find the number of subformulas of the positive sub-formula that have not been proven yet. Then it rates each negated formula with a number, which is the number of to-be-proven sub-formulas. The theorem prover then orders the negated formulas according to their rating and attempts to use the most promising negated formula first; ie the one that has the fewest sub-formulas remaining to be proven. This is most of the times a great heuristic that guides the search but does not prune it in any way.

13.1.5 Change of Focus

The theorem prover is currently used to perform two tasks.

- The first task is to generate "all" proofs that reach the given goal from the given theory. This places some restrictions on pruning and maybe optimization, since proofs of potential value (in terms of Argumentation Logic) should not be clipped from the search space. This is used by the client to provide a way to make proofs (in addition to the proof builder). This is also used by the proof extractor algorithm.
- The second task is to prove that something cannot be proven, by exhausting the search space and not finding a proof for the given goal and theory. This is currently used by the Genuine Absurdity Property check that involves checking whether a contradiction can be proven without the current hypothesis for each application of the $\neg I$ rule.

Experience from using the tool, during its construction and after, has shown that the theorem prover was rarely necessary for creating a proof. Most of the times, this was the job of the proof builder. Perhaps it is better to shift the focus of the jack-of-all-trades theorem prover to use aggressive pruning and optimization in order to better serve the second task and the proof extraction usage. The "finding all proofs" task may prove to be less popular and perhaps should be dropped in order to speed up the theorem prover. More extensive usage of the application will indicate what the theorem prover should really aim to do.

13.1.6 Summary

The theorem prover strives to be sound and complete over the logic constructs it currently supports. However as with any implementation, bugs may interfere. Formal verification was deemed too taxing in terms of time, so plenty of unit tests were used instead to limit the possibility of errors. The theorem prover works, though sometimes it is slow when the search space becomes too complex. This is because the implementation does not prune any proofs because they may be valuable from an Argumentation Logic point of view. Argumentation Logic-compliant pruning (and other performance enhancements) are among the future improvements. Perhaps it might be better to make use of pruning and optimization in order to make a theorem prover specialized in disproving goals, a procedure required by the Genuine Absurdity Property check, and drop support for providing "all" proofs as this was not a frequently used function.

13.2 Other Algorithms and Procedures

Other algorithms and procedures include the (extended) Genuine Absurdity Property checker, the proof visualization algorithm, the proof extraction algorithm and the proof and argument builders.

Unit testing was once again employed for the Genuine Absurdity Property checkers. This is a very important procedure, so great care was taken in order for this check to be correct. Both the original definition and the extended share a great deal of similarities, and that is reflected in the conciseness and code reuse of different parts of the code.

Manual tests were plenty and thorough for all of the above procedures and algorithms. Performance was very dependent on the theorem prover. Most of the runtime of the code was spent inside the theorem prover than any of the above procedures. The runtime of these procedures was found to be negligible, suggesting very optimal solutions. In order to measure the time spent on the theorem prover and the time spent on the other predicates, SWI-Prolog's profiler[4] was used.

Consider as a (representative) example the proof in Figure 13.7. The profiler was used to measure the time spent on different predicates working to determine whether the proof follows the Genuine Absurdity Property or not.

1	$\neg(\alpha \wedge \neg\beta \wedge \neg\gamma)$	given
2	$\neg(\alpha \wedge \beta)$	given
3	$\neg(\alpha \wedge \gamma \wedge \neg\delta)$	given
4	$\neg(\delta \wedge \neg\beta)$	given
5	| α	hypothesis
6	| | β	hypothesis
7	| | $\alpha \wedge \beta$	$\wedge I(5,6)$
8	| | $\bot$	$\bot I(2,7)$
9	| $\neg\beta$	$\neg I(6,8)$
10	| | γ	hypothesis
11	| | | δ	hypothesis
12	| | | $\delta \wedge \neg\beta$	$\wedge I(11,9)$
13	| | | $\bot$	$\bot I(4,12)$
14	| | $\neg\delta$	$\neg I(11,13)$
15	| | $\alpha \wedge \gamma$	$\wedge I(5,10)$
16	| | $\alpha \wedge \gamma \wedge \neg\delta$	$\wedge I(15,14)$
17	| | $\bot$	$\bot I(3,16)$
18	| $\neg\gamma$	$\neg I(10,17)$
19	| $\alpha \wedge \neg\beta$	$\wedge I(5,9)$
20	| $\alpha \wedge \neg\beta \wedge \neg\gamma$	$\wedge I(19,18)$
21	| $\bot$	$\bot I(1,20)$
22	$\neg\alpha$	$\neg I(5,21)$

Figure 13.7: One of the proofs used to profile execution time of different algorithms versus the theorem prover

Listing 6.1 shows the code for the predicate `checkGAPX/1`, and Figure 13.8 shows the analysis done on this predicate. The code listing shows all the predicates used for checking

[4]http://www.swi-prolog.org/pldoc/man?section=profile

for the Genuine Absurdity Property. The yellow line in the figure represents the "current predicate", which is checkGAPX/1. The lines below that represent the callees. Those are the predicates called by the current predicate. It can be seen that the time spent by any of the Genuine Absurdity Property-specific predicates is negligible (0.0%) apart from checkGAP/7, which accounts for most of the total runtime. Focus should now converge to this predicate.

Listing 5.5 shows the code for the predicate checkGAP/7, which makes use of other Genuine Absurdity Property predicates but also the theorem prover. Line 4 of the code listing in particular, makes use of the prover inside a call of the predicate not/1. Figure 13.9 shows the profiling on checkGAP/7. All of the processing time is attributed to the theorem prover. This predicate, and checkGAPX/1, along with their children have negligible runtime. This is not only compared to the theorem prover, but in terms of absolute time as well as shown by Figure 13.10.

Details -- checkGAPX/1

Time		Access		Predicate
Self	Children	Call	Redo	
0.0%	72.2%	1	0	$c_call_prolog/0
0.0%	72.2%	1	0	**checkGAPX/1**
0.0%	72.2%	1	0	checkGAP/7
0.0%	0.0%	1	0	checkRestrictedTheory/1
0.0%	0.0%	1	0	getTheoryAndRevBox/3
0.0%	0.0%	1	0	checkRestrictedRules/1
0.0%	0.0%	1	0	checkRAND/1
0.0%	0.0%	1	0	reverse/2

Figure 13.8: Most of checkGAPX/1's execution time comes from the predicate checkGAP/7

Details -- checkGAP/7

Time		Access		Predicate
Self	Children	Call	Redo	
		23		<recursive>
0.0%	72.2%	1	0	checkGAPX/1
0.0%	72.2%	1	0	**checkGAP/7**
0.0%	72.2%	4	4	not/1
0.0%	0.0%	26	0	a2/3
0.0%	0.0%	13	39	getUsedHypotheses/6
0.0%	0.0%	4	0	a4/5

Figure 13.9: Most of checkGAP/7's execution time comes from the theorem proverwrapped inside a call to not/1

The proof builder (step 1+), and visualizer (step 4) do not make use of the theorem prover, but the proof extractor (step 4+) and argument builder (step 4++) do use the prover. Runtimes were found to be similar in that algorithm-specific predicates had negligible impact on runtime, and the theorem prover was dominating execution time.

The opportunity to optimize the extended version of the Genuine Absurdity Property checker was taken and the result is that siblings do not need to be rechecked whenever referenced. More details about this optimization can be found in section 6.5.

13.3 Server

The server is a very stable and simple unit of code. It was indirectly tested extensively when testing the client (since the server was always running behind the scenes to support

Details -- checkGAP/7

Time		Access		Predicate
Self	Children	Call	Redo	
		23		<recursive>
0.00 s.	44.71 s.	1	0	checkGAPX/1
0.00 s.	44.71 s.	1	0	checkGAP/7
0.00 s.	44.71 s.	4	4	not/1
0.00 s.	0.00 s.	26	0	a2/3
0.00 s.	0.00 s.	13	39	getUsedHypotheses/6
0.00 s.	0.00 s.	4	0	a4/5

Figure 13.10: The runtime of the Genuine Absurdity Property is negligible not only when compared to that of the theorem prover, but in absolute execution time as well

it). As discussed in section 9.2, it requires virtually no maintenance and a very limited amount of requirements to run.

13.4 Client and User Interface

The client strikes a balance between bells and whistles and minimalistic and intuitive interface. It is easy to use and is quite stable. No unit testing was employed in favor of manual testing, since automated GUI testing is non-trivial and quite time-consuming. In the worst case simply hitting F5 (usual shortcut for a page refresh) on the keyboard solves all problems. It has a variety of features that enhance the client's usability, listed in section 10.1.

13.5 Overall Software Engineering Evaluation

Despite the fact that software engineering was not the main focus of the project, great effort was made to keep the different logical modules separate, independent and extensible. The overall architecture of the project is simple and explained in chapter 3. The core can be used standalone, however the client provides a good wrap-around that simplifies the overall procedure of using the functionality offered by the system. Deployment is made easy because of the client-server infrastructure. Stability is not an issue, as there is a seamless interaction between the client and the server (and core). The only noticeable problem is that sometimes the server is slow to reply to the client after a request was made because the theorem prover takes longer than usual to run.

13.6 Project Aims, Objectives and Contributions

One of the main objectives of the project was to explore Argumentation Logic. Argumentation Logic is a relatively new concept.

Contributions to this concept include the extension of the Genuine Absurdity Property to work with natural deduction proofs that make use of the substitution shortcut that allows sub-derivations to refer to sibling derivations instead of re-proving what they have already proven. The extension explains how an optimization can be achieved by not re-checking sibling sub-derivations for the Genuine Absurdity Property as it does not change with respect to the content they are imported to.

Another contribution was the invention of the visualization algorithm. This algorithm or procedure, provides an exact definition as to how a proof bearing the Genuine Absurdity

Property can be converted into an argument. This algorithm is also consistent with the extension of the property. This algorithm creates a mapping between attacks and defenses with respect to arguments, and contradictions and hypotheses with respect to proofs.

Along the contributions made by the project is a constructive algorithm that can turn an argument back into a proof, and this relies on the visualization algorithm definition, namely on the mapping it defines.

The final contribution of this project is the creation of a software tool that implements the aforementioned concepts and algorithms in order to provide a playground in which learners of Argumentation Logic can practice.

The software tool also provides a proof builder and an argument builder (for Argumentation Logic), and so at a lesser degree, it can be used to learn about propositional logic and maybe argumentation.

Unfortunately, there was not enough time to explore the more challenging concepts of Argumentation Logic, namely its effort to work well in a paraconsistent environment. This topic however is still somewhat under construction, and is therefore volatile and subject to change. Among the future work is to explore this part of Argumentation Logic.

Chapter 14

Conclusions and Future Work

Argumentation Logic is still new, but many lessons were learned by undertaking this project and studying the concept of Argumentation Logic. Among what has been learned is how this new type of logic bridges the gap between traditional propositional logic and argumentation theory, and establishes a connection and an analogy between the two.

Argumentation Logic shows how proofs can be seen as exchanges of relevant arguments, by establishing a check that enforces relevance (that is, the Genuine Absurdity Property, step 3). Subsequently, finding a way to visualize proofs by defining a mapping that allows to convert a natural deduction proof to an argument (step 4) and vice versa (step 4+) was another interesting lesson, something now available for further study as a result of this project.

Argumentation Logic can be extended further to function with more flexible versions of natural deduction that can include substitution and potentially many more shortcuts and derived rules or connectives (step 3+).

Finally, it can be inferred that an implementation of a particular concept is a great way to help in discovering ideas and probing the concept at hand, leading to more improvements and extensions and a much better understanding of what it involves. In fact, many of the implemented procedures (such as the extended Genuine Absurdity Property, the visualization of proofs and the extraction of proofs from arguments) were in a sense byproducts of probing Argumentation Logic during the journey of building this tool. The journey (findings) could arguably be considered more important than the destination (final product).

Future work for the theorem prover includes Argumentation Logic-compliant pruning as well as other performance enhancements that will allow the theorem prover to run faster. Profiling the runtime of the theorem prover has identified where most of the execution time is spent and can allow the optimization to focus on the slow-running parts of the code.

Improvements include the ability to create arguments that distinguish between child derivations and sibling derivations for the visualization algorithm if that is found to be more preferable. In that way, generated arguments will contain the information that certain parts of the attacks and their defenses came from references to siblings instead of children. More information about this can be found in section 7.3. Identifying repeating arguments and creating shorter proofs that reuse one instance of such arguments is a potential upgrade to the current proof extraction algorithm as discussed in section 8.3.

With regards to the client, improvements could include database support for storing proofs and arguments. This would impose a burden on the server however, since user identification and a database setup and connection would be necessary.

Most of the future work however should be pointed towards understanding the paracon-

sistency realm of Argumentation Logic and perhaps upgrade the core and client to provide a playground for playing with paraconsistency. Many of the algorithms and extensions were the result of trying to use Argumentation Logic in order to build a concrete software tool around it. In the same way, by trying to implement the parts of Argumentation Logic that deal with paraconsistency, many doubts, observations, ideas and opportunities for improving Argumentation Logic will surface.

List of Figures

1.1 One of the generated proofs of the theorem prover with theory $T = \{\neg(a \wedge \neg b \wedge \neg c), \neg(a \wedge b), \neg(a \wedge c \wedge \neg d), \neg(d \wedge \neg b)\}$ and goal $\neg a$. "!" represents negation and "&" represents conjunction 10
1.2 The proof in Figure 1.1, now awarded the Genuine Absurdity Property ribbon to indicate that it follows the property. "!" represents negation and "&" represents conjunction 11
1.3 The proof in Figure 1.2, now now vizualised as an argument. Green nodes represent defenses by the proponent, and red nodes represent attacks by the opponent. 11
1.4 The proof builder initialized to build a proof with theory $T = \{\neg(a \wedge b), \neg(a \wedge c), \neg(a \wedge \neg b \wedge \neg c)\}$ and goal $\neg a$. 12
1.5 The argument builder initialized with a complete argument about a, using theory $\neg(a \wedge b \wedge c), \neg(a \wedge \neg b), \neg(a \wedge \neg c)$. 13

2.1 Visualization of abstract argumentation example in subsection 2.1.6 18
2.2 Visualization of the proof of example in section 2.3.4 25
2.3 Proofs showing the use of a shortcut in natural deduction. The proof on the right derives $\neg\beta$ again whereas the proof on the left simply reuses the previous derivation of $\neg\beta$ 26
2.4 Proofs showing the implicit use of copying of ancestor hypotheses: the proof in the middle makes no reference to the ancestor hypothesis α whereas the other two implicitly copy it 27

3.1 The high-level system architecture for the chosen solution 33
3.2 The high-level functional map for the core. White parallelograms represent data, purple boxes represent core predicates and blue boxes represent client functionality 36

4.1 The Prolog constructs accepted and used by the theorem prover 40
4.2 The Prolog constructs accepted and used by the theorem prover 42
4.3 A natural deduction proof and the corresponding output from the theorem prover 43

6.1 Proofs showing that imposing an ordering on the sibling derivations makes the extension definition results dependent on that ordering 54
6.2 Comparisson of different candidate definitions for extending the Genuine Absurdity Property 55
6.3 Referencing of an ancestor's sibling (uncle) derivation 56

7.1 Example proof for the visualization algorithm 61

7.2 Visualization of the proof of example in Figure 7.1 62
7.3 Example another proof that results in a framework like in Figure 7.2. This proof is redundant but correct under the rules of natural deduction nevertheless. 64
7.4 Visualization example of 2-level boxes 67
7.5 Visualization example of empty set attack 68
7.6 Visualization example of ignored successful defense 68
7.7 Visualization example of 3-level boxes 69
7.8 Visualization example of theory attack 69
7.9 Visualization example of 4-level boxes 70

8.1 Proof extraction example 72
8.2 Proof extraction example of two proofs generated from the same argument . 73

10.1 The client GUI showing the workbench on the right, the clipboard on the left and the options on the top right corner 86
10.2 Three thumbnails, from left to right: a proof, a verified Genuine Absurdity Property proof, and an argument 87
10.3 A notification pops up when the user clicks on the "Clear Clipboard" button that asks for confirmation 88
10.4 Incorrect input is highlighted so that the user can revise it 89
10.5 The theorem prover generates all the proofs that arrive to the given goal using the supplied theory 89
10.6 The user can drop an unverified proof onto the placeholder and select between the original and extended definitions for the Genuine Absurdity Property 90
10.7 If the proof follows the Genuine Absurdity Property it is indicated using a ribbon 91
10.8 The Visualize GAP tab with a given verified proof and its corresponding argument 91
10.9 The Extract GAP Proof tab with a given argument and its corresponding verified proof 92
10.10A pop-up shows when the user clicks the "Import to Clipboard" button on the options dropdown menu 93
10.11The logical symbols used by the client application 94

11.1 The proof builder already fills in the theory as steps for the proof that the user is about to build 98
11.2 The proof builder indicates an error when the user input is incorrect 98
11.3 Supported natural deduction rules for the proof builder. A # represents a line number 99

12.1 The argument builder draws the initial argument and enables the attack input field awaiting for the user's command 101
12.2 The argument builder draws the initial argument and enables the attack input field awaiting for the user's command 102
12.3 The user is about to attack the computer's argument; a red link shows the node the attack will be against if the user drops the attack node 103
12.4 The user gives a terminal attack on the left branch, leaving only the right branch open; after a terminal attack on the right branch, the argument is complete 103

13.1 A list of ten theories and goals proven by the theorem prover along with their execution time. The theorem prover need only find one proof 106
13.2 A list of ten theories and goals proven by the theorem prover along with their execution time. The theorem prover finds all proofs for each of the given theory and goal 107
13.3 The profiler shows that most of the execution time is attributed to context lookups, as rules really are pattern matching rules 108
13.4 Most of the execution time is attributed to `m3/3`, a predicate that calls Prolog's `member/2` twice to look up a term in the current and inherited context 108
13.5 Most of the lookups are called by `falsityIx/4`, the forward rule that tries to find α and $\neg\alpha$ in the context (for any α) and thus derive a contradiction . . 108
13.6 Pruning could potentially trim proofs of potential importance regarding Argumentation Logic 112
13.7 One of the proofs used to profile execution time of different algorithms versus the theorem prover 114
13.8 Most of `checkGAPX/1`'s execution time comes from the predicate `checkGAP/7` 115
13.9 Most of `checkGAP/7`'s execution time comes from the theorem proverwrapped inside a call to `not/1` 115
13.10The runtime of the Genuine Absurdity Property is negligible not only when compared to that of the theorem prover, but in absolute execution time as well 116

Listings

5.1 Genuine Absurdity Property top level predicate 45
5.2 Checking whether a proof is a RAND proof 45
5.3 Checking whether a proof uses only conjunction and negation 46
5.4 Checking whether a proof uses any shortcuts 47
5.5 Checking whether a proof follows the Genuine Absurdity Property 47
5.6 Gathering of referenced ancestor derivations for the original Genuine Absurdity Property definition 48
6.1 Checking whether a proof is follows the extended Genuine Absurdity Property 58
6.2 Gathering of referenced sibling derivations for the extended Genuine Absurdity Property definition 58
7.1 First part of the proof visualization algorithm 63
7.2 Second part of the proof visualization algorithm 65
7.3 Third part of the proof visualization algorithm 65
8.1 First part of the proof extraction algorithm 75
8.2 Second part of the proof extraction algorithm 76
8.3 Third part of the proof extraction algorithm 77
9.1 The Prolog code that configures and runs the server 80
9.2 The server configuration file written in Prolog 81
9.3 The server code file that registers handlers that server client queries 81

Bibliography

Terence Anderson, David Schum, and William Twining. *Analysis of Evidence*. Cambridge University Press, Cambridge, 2005.

David Barker-Plummer, Jon Barwise, and John Etchemendy. *Language, Proof, and Logic*. University of Chicago Press, Chicago, 2011.

Trevor Bench-Capon. Value-based argumentation frameworks. In *9th International Workshop on Non-Monotonic Reasoning (NMR 2002)*, Toulouse, France, April 2002.

Phillipe Besnard and Anthony Hunter. A logic-based theory of deductive arguments. *Artificial Intelligence*, 128:203–235, 2001.

Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77: 321–357, September 1995.

Melvin Chris Fitting. *First Order Logic and Automated Theorem Proving*. Springer, November 1995.

Joseph B. Kadane and David A. Schum. *A Probabilistic Analysis of the Sacco and Vanzetti Evidence*. Wiley, New York, 1996.

Antonis Kakas, Francesca Toni, and Paolo Mancarella. Argumentation logic. Technical report, University of Cyprus, Imperial College London, Universita di Pisa, Nicosia, April 2012.

Eric C. W. Krabbe. *Reason Reclaimed*. Vale Press, Virginia, 2007.

Sanjay Modril and Martin Caminada. Proof theories and algorithms for abstract argumentation frameworks. Technical report, King's College London, 2008.

Wilfried Sieg and Richard Scheines. Searching for proofs (in sentential logic). *Philosophy and the Computer*, pages 137–159, March 1992.

Guillermo Simari and Iyad Rahwan. *Argumentation in Artificial Intelligence*. Springer, London, 2009.

Francesca Toni. A tutorial on assumption-based argumentation. *Argument & Computation*, pages 89–117, 2013.

Steven Toulmin. *The Uses of Argument (Updated Edition)*. Cambridge University Press, Cambridge, 2003.