How do you make software components argue?

An exploration of argumentative reasoning within autonomous software agents

Daniel Bryant

Submitted for the MPhil to PhD
Transfer from the
University of Surrey

Software Systems Group
Department of Computing
School of Electronics and Physical Sciences
University of Surrey
Guildford, Surrey GU2 7XH, UK

March 2007

©Daniel Bryant 2007

ABSTRACT

Autonomous software components, more commonly referred to as software agents, are often cited as a key enabling technology for the next-generation of online services. However, in these services, consisting of large-scale open multi-agent systems, classical forms of reasoning (such as provided by traditional monotonic logics) will be unsuitable. In this work we discuss our vision for a non-monotonic reasoning component for software agents, and present our current work on a prototype light-weight Java-based reasoning engine. The logical machinery underlying our implementation is based on defeasible argumentation, a relatively new paradigm in logical reasoning based upon sound theoretical concepts from the study of argument. The use of argumentation allows an agent to reason about and provide support for opinions, claims, proposals and ultimately decisions and conclusions in a flexible manner. The primary motivations behind this report are to illustrate that a practical agent-based implementation of argumentation is now viable. However, much work remains to be done, particularly in regard to the inherent computational complexity of the process of argumentation. A fundamental message of our work is that we take seriously the need for open source tools and benchmarking applications, which will allow us to gather sound empirical evidence for the applicability of argumentation within this domain.

TABLE OF CONTENTS

TABLE OF CONTENTS – *Continued*

CHAPTER 1

Introduction

1.1 Overview

Autonomous software components, more commonly referred to as software agents (Franklin and Graesser, 1997; Wooldridge, 2002), are often cited as a key enabling technology for the next-generation of online services. The purpose of these services can be diverse, and includes large-scale electronic-commerce (Tsvetovatyy et al., 1997), information organisation and retrieval (Bryson et al., 2003), recommender systems (Salter and Antonopoulos, 2006), automated decision making (for example, when reasoning over proposals for action (Girle et al., 2004) or assisting in multi-party negotiation (Parsons et al., 1998)) and facilitating the emerging distributed system paradigm known as Service-Oriented Computing (Papazoglou, 2003). In order to be effective in these services a software agent must act rationally (Wooldridge, 2000) and be capable of expressing claims and judgements, with the ultimate goal of reaching a decision, a conclusion, or informing, convincing or negotiating with human users or other agents (Amgoud et al., 2004).

However, in these services, consisting of large-scale open multi-agent systems, classical forms of reasoning (such as provided by traditional monotonic logics) will be unsuitable. Pertinent information may be insufficient or in contrast there may be too much relevant but partially coherent or contradictory information (Amgoud et al., 2004), and in the case of multi-agent interaction, conflicts of interest are inevitable (Wooldridge, 2002). In everyday life, decision involving this type of information is often based on arguments and counter-arguments. The decisions made in this way have a basis that can be more easily referred to for explanation purposes. Not only is a best choice suggested, but also the reasons of this recommendation can be provided in a format that is easy to grasp. When compared with traditional automated rule-based decision-making this approach is more acute with the way humans often deliberate and finally make a choice (Amgoud and Prade, 2004a). Accordingly, agents may benefit from the use of 'defeasible argumentation', a relatively new paradigm in logical reasoning based upon sound theoretical concepts from the study of argument, in order to support opinions, claims, proposals and ultimately decisions and conclusions. In particular argumentation is a promising model for reasoning with inconsistent knowledge. It follows a three step process: constructing arguments and counter-arguments, then selecting the most acceptable of them and finally concluding.

1.2 Why can't agents argue? Motivations for our work

In our recent work (Bryant et al., 2006; Bryant and Krause, 2006) we have argued that for software agents to be assisted by argumentation, they will require access to a general purpose non-monotonic reasoning component (an "argumentation engine"). However, creating a concrete implementation of an argumentation-based reasoning process typically

requires many complex design-decisions to be hard-coded into the application (i.e. forcing a particular formalism or argument judgement process on the user) and also presents several computational complexity issues. When these facts are combined with the (typically resource-bounded) constraints placed upon the execution environment of a software agent, this makes the design and implementation of such a reasoning component a non-trivial task.

There are several successful prototypes of standalone non-monotonic and argumentation-based reasoning applications documented in academic literature, but many of the implementations are proprietary, and the code (and hence verifiability) is deliberately hidden from view, preventing other researchers from exploring the inner workings of the system and making modification. Other prototypes are often built using esoteric or inefficient languages not suitable for large-scale (or agent-based) deployment, are not flexible with regard to input and output of data, and offer limited configuration options to the user. It is our belief that such an application should be constructed in an appropriate (and efficient) language and should also be open-source, allowing (and in fact encouraging) other researchers to build on the previous work, such as enabling support for new formalisms, adding new configuration options and conducting additional experimentation.

Research in argumentation has also inherited the tendency from the nonmonotonic reasoning community to focus the majority of attention on the theoretical issues, ignoring the practical details of algorithm implementation. There are many examples pervading the argumentation literature where increasingly complex theoretical scenarios are created in order to 'test' the success of argumentation systems, without thorough analysis as to the practicality of these scenarios. In addition, when analysis of systems does occur, it frequently tends to consider only theoretical worse-case scenarios (for example, (Dunne and Bench-Capon, 2002), (Dimopoulos et al., 2000)) which may not reveal the real world performance of the algorithms (Vreeswijk, 2006). The proprietary implementations of algorithms as mentioned previously has also not helped this situation, making the establishment of a common testing platform difficult, and although several of the implementations include tools to automatically generate test cases, there is sometimes suspicions that these tools have been engineered to produce test cases that best suite the authors implementation. We believe that the argumentation research community would benefit from a common repository of realistic test data that is publicly available (ideally large-scale), and a common methodology or platform on which to conduct empirical evaluation.

In summary, the primary motivations behind this report are to illustrate that a practical agent-based implementation of argumentation is now viable, but much work remains to be done. A fundamental message of this report, and a common theme throughout our work, is that we take seriously the need for open source tools and benchmarking applications that will allow us to gather sound empirical evidence for the applicability of argumentation.

1.3 Our vision

In this report we discuss our vision for a non-monotonic reasoning component for software agents and our current work on a prototype light-weight Java-based argumentation engine. The core engine has been built using tuProlog (Denti et al., 2005, 2001), an existing

open-source Prolog engine, as its foundation, which followed the same design principles that we require for our intended domain of application. Although our goal is to create a general purpose argumentation engine that can be configured to conform to one of a range of semantics, the current version of the engine implements the argumentation-based framework presented in (Amgoud et al., 2005). This allows our engine to generate arguments and counter arguments over an inconsistent knowledge base, determine the acceptability of arguments and construct proofs using an argument game approach to constructing proofs of acceptance in the style of (Jakobovits and Vermeir, 1999a). As the engine is built on top of an existing Prolog implementation it naturally supports standard Prolog inference which allows us to prototype a variety of meta-interpreters that support other forms of argumentation.)

This report also explores how we might approach our ultimate goal of creating a toolkit to support empirical experimentation with argumentation system implementations and facilitate potential use for our work, recommender systems. This is achieved by examining issues such as the implementation of an experimental testing strategy for argumentation systems, the generation of simple realistic arguments to use in empirical evaluations and exploring how numerical values of uncertainty may be incorporated into practical implementations.

1.4 Report Structure

Chapter 2 discusses key concepts within software agents and argumentation theory. A brief history of each topic is given, including the trends that lead to the emergence of the two paradigms in their respective fields. The chapter concludes by discussing and motivating our interest in combining the two topics.

The challenges that are presented when trying to exploit argumentation within software agents are discussed in Chapter 3. The chapter begins with a review of the challenges presented within each individual topic,discusses how these may affect our work and explores way in which these challenges may be overcome.

Chapter 4 discusses the limited amount of previous work on combining the software agents and argumentation, and examines how (if at all) the identified challenges have been overcome. This line of research is extended in Chapter 5, with the presentation of an in-depth review of existing defeasible and argumentation-based reasoning implementations. This review focuses mainly on the technical details of the implementations, and identifies the intended use of the implementation, the underlying technology utilised, the interface presented, configuration options offered, complexity issues, experimental evaluation and comparison with other implementations.

Chapter 6 discusses our current work on a lightweight Java-based prototype argumentation engine for software agents. Here we identify our design methodology, discuss implementation details and examine several of the problem that were identified when demonstrating the application at a recent conference.

Our future plans for both the current engine implementation and also other associated projects, including how we might approach our ultimate goal of creating a toolkit to support empirical experimentation with argumentation system implementations, are presented in Chapter 7. The final chapter concludes the work we have presented.

CHAPTER 2

Key Concepts

2.1 Overview

This chapter introduces key concepts and terminology that is used throughout this report. The brief histories that lead to the emergence of both software agents within software engineering and argumentation within defeasible logics is given in order to provide background information and motivate our interest in both topics.

2.2 Software Agents

2.2.1 The emergence of agents

As computer systems have become increasingly ubiquitous they have evolved from being isolated entities into large distributed interconnected systems. As a result, software technology has undergone a transition from monolithic systems, designed to run on a single platform, to ad hoc ensembles of semi-autonomous, heterogeneous and independently designed sub-systems (Wooldridge, 2002). These trends have led to the emergence of a new type of software application classified as autonomous software agents. In the classical sense an agent is defined as "One who does the actual work of anything, as distinguished from the instigator or employer; hence, one who acts for another" (OUP, 2007). It therefore should come as no surprise that a software agent is a software component that acts on behalf of a user (either a human-user or additional software component) within the environment it is situated in, and is often imbued with some form of intelligence allowing the agent to autonomously work, plan, reason and react to changes in its environment.

A precise definition of the term "software agent" has become somewhat of a holy grail within the computer science community and there are several widely accepted definitions. However, this report will use the definition by Franklin and Graessar (Franklin and Graesser, 1997), which captures the core essence of an agent as a situated and embodied problem solver.

> An autonomous software agent is a system situated within an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect[sic] what is senses in the future. (Franklin and Graesser, 1997, p. 4)

In more specific terms, two distinct views of agents can be identified. The weak notion of an agent states four properties necessary and sufficient for agenthood; autonomy, reactiveness, proactiveness and social ability. The strong or intentional notion of agents requires agents to be based around control architectures comprised of mental components such as beliefs, desires and motivations (intentions).

An additional interesting and important property of agents, although often cited as orthogonal (Franklin and Graesser, 1997), is the ability to be mobile network-aware entities which can autonomously change their execution environment, transferring themselves between distributed systems at run-time. As stated by Koukoumpetsos and Antonopoulos (2002), this allows an agent to migrate to remote resources where interaction can take place locally, thereby reducing data traffic and latency, increasing efficiency and making the application more robust. Code mobility is likely to play an increasingly important role in the future as the bandwidth gap between stationary and mobile devices continues to grow and the amount of data required for processing increases (Kotz and Gray, 1999). However, the current research conducted has not explored this feature of agent technology, and therefore it will not be discussed further in this report

2.2.2 The next generation of agents

In order to realise several proposed next-generation computational services, such as large-scale electronic-commerce (Tsvetovatyy et al., 1997), automated decision making (Girle et al., 2004) and Service-Oriented Computing (Papazoglou, 2003), the use of autonomous software agents will be essential due to the large volumes of data and distribution of responsibilities amongst applications, systems and organisations. These agents must also be capable of interacting with other agents, and as stated by Wooldridge, "not simply by exchanging data, but by engaging in analogues of the kind of social activity that we all engage in every day of our lives: cooperation, coordination, negotiation, and the like" (Wooldridge, 2002, p. 1). These multi-agent systems will be composed of multiple individual agents and seem a natural metaphor for building a wide range of "artificial social systems" (Wooldridge, 2002, p. 1). In these systems a software agent must act rationally (Wooldridge, 2000) (i.e. make good decisions in its own best interests given the beliefs it has about the world) and be capable of expressing claims and judgements, with the ultimate goal of reaching a decision, a conclusion, or informing, convincing or negotiating with other agents (Amgoud et al., 2004).

In order to be capable of making and expressing good decisions an agent will require a well-defined internal reasoning process. First attempts at representing and manipulating an agent's environment used classical forms of logic (Wooldridge, 2002). Logic was chosen because by fixing on a structured, well-defined artificial language (as opposed to an ill-defined natural language), it is possible to investigate the question of what can be expressed in a rigorous, mathematical way. However, representing realistic environments and conducting 'common sense reasoning' (as performed by humans when presented with contradictory or incoherent information) was not possible using classical logic, due to factors such as the monotonic nature of the logic and inherent computational complexity issues (Garey and Johnson, 1979). Accordingly, a series of nonmonotonic logics were created, and although many of these systems proved useful they often came with their own problems (Cadoli and Schaerf, 1993). Recently, a relatively new paradigm in reasoning logic referred to as argumentation, a process based on the exchange and valuation of interacting arguments, has been promoted as an ideal system to overcome these problems and meet the requirements of a flexible reasoning process.

2.3 Argumentation

2.3.1 The rise of nonmonotonic logics

The area of nonmonotonic logics originated in the late 1970's in an effort to build effective knowledge representation formalisms and deal with the inherent challenge of modelling commonsense reasoning which almost always occurs in the face of incomplete and potentially inconsistent information. A logical model of commonsense reasoning demands the formalization of principles and criteria that characterise valid patterns of inference. In this respect, classical logic proved to be inadequate since it behaves monotonically (Chesnevar et al., 2000a). Since then solid theoretical foundations of nonmonotonic logics have been established (the interested reader is referred to (Antoniou, 1997) for further information). Although this report is not focused on nonmonotonic logics in general, it is still useful to give a brief overview of the topic as these logics could be considered the precursor of argumentation and share many characteristics.

Several styles of nonmonotonic logics exist. Most of them takes as the basic 'non-standard' unit, the notion of a default, or defeasible condition or rule: this is a conditional that can be qualified with phrases like 'typically' 'normally' or unless shown otherwise' (Prakken and Vresswijk, 2002). Defaults do not guarantee that their consequent holds whenever their antecedent holds, instead they allow us in such cases to defeasible derive their consequent i.e. if nothing is known about exceptional circumstances. Most non-monotonic logics aim to formalise this phenomenon of 'default reasoning', but they do so in different ways. The efforts of the past two decades culminated in several noteworthy nonmonotonic systems: default logic, logic programming with negation as failure, defeasible logic and plausible logic. Several of these systems will be briefly reviewed later in the implementation review section, and the interested reader is referred to (Antoniou, 1997) (Prakken and Vresswijk, 2002) for a comprehensive overview.

2.3.2 Introducing Argumentation Systems

Argumentation systems are yet another way to formalise nonmonotonic reasoning, using the construction and comparison of arguments for and against certain conclusions (the interested reader is referred to three excellent surveys in this field (Prakken and Vresswijk, 2002) (Chesnevar et al., 2000a) (Besnard and Hunter, 2006)). The next two sections of the report aim to give a theoretical overview necessary for understanding the practical implementation issues discussed later, and as such sometimes follows closely the discussion of argumentation systems given by Prakken and Vreeswijk in their seminal survey paper (Prakken and Vresswijk, 2002). It should be noted that at this stage of the research we are only interested in symbolic approaches of inference under uncertainty, and therefore discussion of systems exhibiting nonmonotonicty using numerical theories in combination with logics (such as probabilistic argumentation systems (Haenni, 2001)) is deliberately kept to a minimum.

In argumentation systems the basic notion is not that of a defeasible conditional, but that of a defeasible argument. The idea is that the construction of arguments is monotonic, i.e. an argument stays an argument if more premises are added. Nonmonotonicity, or defeasibility is not explained in terms of the interpretation of a defeasible conditional,

but in terms of the interactions between conflicting arguments. In argumentation systems nonmonotonicty arises from the fact that new premises may give rise to stronger counter-arguments, which defeat the original argument. When we chain defeasible reasons to reach a conclusion, we have arguments, instead of proofs. It makes sense to require defeasible reasons for argumentation. Arguments may compete, rebutting against each other, so a process of argumentation is a natural result of the search for arguments. Adjudication of competing arguments must be performed, comparing arguments in order to determine what beliefs are justified.

Argumentation systems can be applied to any form of reasoning with contradictory information, whether the contradictions have to do with rules and exceptions or not. For instance, the contradictions may arise from reasoning with several sources of information, or they may be caused by disagreement about beliefs or about moral, ethical or political claims.

2.3.3 The key ingredients

According to Prakken and Vresswijk (2002) argumentation systems generally comprise of the following five elements (although sometimes implicitly): an underlying logical language, definition of an argument, of conflicts between arguments and of defeat among arguments and, finally, a definition of the status of arguments, which can be used to define a notion of defeasible logical consequence. The first two elements still fit with the standard picture of what a logical systems is, as argumentation systems are built around an underlying logical language and an associated notion of logical consequence, defining the notion of an argument. Some argumentation systems assume a particular logic, while other systems leave the logic partly or wholly unspecified. The notion of an argument corresponds to a proof (or the existence of a proof) in the underlying logic.

The remaining three elements are what make an argumentation system a framework for defeasible argumentation. The first is the notion of conflict between arguments. Three types of conflict are discussed in the literature, rebutting, assumption attacks and undercutting (see (Prakken and Vresswijk, 2002) for further explanation.) The notion of defeat is a binary relation on the set of arguments. It is important to note that this relation does not yet tell us with what arguments a dispute can be won; it only tells us something about the relative strength on two individual conflicting arguments. Therefore what is also needed is a definition of the status or arguments on the basis of all the ways in which they interact. It is this definition of the status of arguments that produces the output of an argumentation system: it typically divides arguments in at least two classes: arguments with which a dispute can be 'won' and arguments with which a dispute should be 'lost'. Sometimes a third intermediate category is also distinguished containing arguments that leave the dispute undecided.

These notions of status can be defined both in a 'declarative' and in a 'procedural' form. The declarative form, usually with a fixed point definition, just declares certain sets of arguments as acceptable, (given a set of premises and evaluation criteria) without defining a procedure for testing whether an argument is a member of this set: the procedural form amounts to defining just such a procedure. Thus the declarative form of an argumentation system can be regarded as its (argumentation-theoretic) semantics, and the procedural

form as its proof theory.

2.3.4 Accepting arguments

We can say that argumentation systems are not concerned with truth of propositions, but with justification of accepting a proposition as true, therefore arguments are either justified or not justified. However, how do we determine which arguments are justified? In the literature, two approaches to the solution of this problem can be found. The first approach is by providing a definition such that there is always precisely one possible way to assign a status to arguments, the so called 'unique status assignment' approach. The second approach instead regards the existence of multiple status assignments not as a problem, but as a feature: it allows for multiple assignments (each representing a valid point of view) and defines an argument as 'genuinely' justified if and only if it receives this status in all possible assignments.

The problem to decide which arguments may be accepted has two aspects. The first aspect, the theory, is concerned with questions such as which notions of acceptability there exists and how different notions of acceptability relate to each other. Beginning with Dung's seminal work (Dung, 1995) there have been many proposals for such a notion of acceptability including grounded, admissible, preferred and stable. When dealing with argument systems, questions often boil down to the following two fundamental problems: Should this argument be accepted in all possible worlds? i.e. should everyone accept this argument? Is there a possible world in which this argument must be accepted? i.e. can anyone defend this argument? According to Vreeswijk this part is relatively well understood (Vreeswijk, 2006). The second aspect of deciding which arguments may be accepted is involved with the design and analysis of algorithms that decide on acceptability. Here the analysis is divided into two approaches. The first approach is interested in the complexity of specific acceptability problems in worst-cases. This direction is well covered in, for example (Dunne and Bench-Capon, 2002) (Dimopoulos et al., 2002). The second approach is interested in the design of algorithms with the intention to actually use them in practise, and this is the area most related to our desire of the practical application of argumentation.

2.3.5 Argumentation Algorithms

Algorithms to decide on argument acceptability can be further divided into two subtypes, namely query based algorithms and total algorithms. Query-based algorithms compute answers for one particular argument, whether such answers are yes/no answers, defence sets or full extensions. Total algorithms compute answers for all arguments, defence sets etc. The decision as to which of the two types is most appropriate depends on the reasoning scenario and Chapter 3 will discuss such scenarios in more detail. However, before providing pointers to algorithms documented in the academic literature it is worth taking a step back to look at the broader categories of argumentation. Within the argumentation literature some authors focus their studies on a logic based theory of deductive arguments, or the problem of the acceptability and comparison of arguments, and many authors consider argumentation as a dialectic process during disputation.

The first category is well covered in work by Besnard and Hunter, and they have presented several query-based algorithms for deductive argumentation using classical logic (a comprehensive discussion of the algorithms, complexity issues and other interesting issues can be found in (Besnard and Hunter, 2006)). The second category has been explored be several researchers, including Pollock, who over the past two decades has worked on a comprehensive framework to support the operation of a rational agent, including comprehensive defeasible reasoning algorithms (Pollock, 1992) (which have been successfully implemented in the OSCAR software suite (Pollock, 1995)). Cayrol and colleagues have also presented several algorithms for determining argument acceptability based on comparison of arguments, one example of which includes decision algorithms for total computation of credulous preferred acceptance and sceptical preferred acceptance in coherent argumentation systems (Cayrol et al., 2003) (an argumentation system is coherent iff preferred and stable extensions coincide). Blurring the lines between the second and third categories of is Dung, Kowalski and Toni, who have presented a family of dialectic proof procedures for the admissibility semantics of assumption based argumentation (Dung et al., 2006). However, the third category has received the most attention in recent literature and also been implemented the most in practical applications.

Although there have been numerous contributions to the study of argumentation as a dialectic practice, we are in particular interested in contributions by two groups. The first group consists of Guillermo Simari, Alejandro Garcia, Carlos Chesnevar and colleagues from the Universidad Nacional del Sur in Argentina, and the second consists of Gerard Vreeswijk from the Universiteit Utrecht in The Netherlands. The first group began with Simari (in combination with Loui) being one of the initial proponents of argumentation-based algorithms for defeasible reasoning (outlined in his Mathematical Treatment of Defeasible Reasoning framework (Simari and Loui, 1982)). Later work conducted by Garcia and Simari has resulted in several algorithms and a very interesting practical implementation of what they refer to as "Defeasible Logic Programming (DeLP)". Although this work has perhaps not received the attention it deserves from the academic community, Chesnevar and colleagues have worked extensively with both the algorithms and implementation, producing several papers outlining computational issues and how they may be overcome, and also producing several practical small-scale demonstrators for realistic scenarios.

Gerard Vreeswijk has been a constant contributor to the argumentation-based algorithm community over the past two decades and a strong proponent of practical implementations of argumentation. Vreeswijk implemented algorithms from his PhD thesis in the early nineties resulting in IACAS (Vreeswijk, 1993a), a practical implementation of dialogue based argumentation (discussed in Chapter 5). Later work from Vreeswijk includes collaboration with partners of the Argumentation Service Platform with Integrated Components (ASPIC) EU project resulting in dialectic-based query algorithms and another practical implementation to determine the acceptance of arguments based on the semantics of credulously preferred sets (Amgoud et al., 2005). The latest work of Vreeswijk is presented in (Vreeswijk, 2006) where he outlines an algorithm that computes grounded and admissible defence sets based on credulous preferred acceptance in one pass for single arguments.

This brief tour of argumentation algorithms has attempted to illustrate that there are many diverse algorithms available and opportunities for them to be adapted for use within an agent-based reasoning component, avoiding the need to create a new algorithm which

would be a demanding task. However, a final word of caution should be made regarding the algorithms discussed, and this is that many lack a complexity analysis and therefore it remains unclear how well these algorithms perform in practise.

2.3.6 Existing uses of Argumentation

Although the previous section may have ended on a pessimistic note it is worth stating that we feel strongly that argumentation is worth studying within the context of 'common sense reasoning'. Argumentation has already proved useful in many areas, notably the medical and legal domains (where it is a natural metaphor for the type of reasoning that occurs). Notable successful practical uses of argumentation include a tool for diagramming dialectic argumentation presented in text (Reed and Rowe, 2004) which allows argument structures to be identified and exported in a common and open language, and argument assistants that guide the users production and structure of arguments (specifically for the application of law). Recently an argumentation based reasoning engine was created that, for example, can facilitate web searches by allowing users to increase relevant documents returned from a query (Chesnevar et al., 2006b). This is achieved by allowing users to specify their personal preferences using defeasible rules in addition to a query. However, although these applications are proving useful to human users, perhaps with the exception of Garcia, Chesnevar and colleagues work (Garcia et al., 2000) (Capobianco et al., 2004) (which will be discussed later), there has been little implementation of argumentation within software agents.

2.4 In summary

This chapter has shown how trends in software engineering over the past decade have lead to the emergence of autonomous software agents. Although existing agents utilising classical logics have been useful, the next generation of agent technology will require the use of a more powerful reasoning logic. The chapter has also introduced such a group of logics, referred to as logics for defeasible argumentation, and attempted to provide a brief theoretical background to motivate our interest in practical implementation work discussed in later chapters. Finally, we have identified that there is a diverse range of argumentation-based algorithms available and several successful existing uses of argumentation in practical software applications. However, we are left questioning why existing agent technology hasn't already taken advantage of such a flexible reasoning paradigm.

CHAPTER 3

Arguing with Agents - The Challenges

3.1 Overview

The purpose of this chapter is to discuss the challenges of incorporating argumentation-based reasoning within software agents. The chapter begins by separating the two paradigms and analysing the individual challenges presented, identifying the inherent problems associated with the use of both software agents and argumentation-based reasoning. A discussion of how these challenges may affect our work and potential solutions are also included. The chapter concludes with an overview of the key issues and analyses the impact of combining the two paradigms.

3.2 Challenges with Software Agents

3.2.1 Why aren't agents everywhere?

The software agent concept was introduced into academic literature in the 1980's, with the discussion becoming prevalent during the 1990's. Increasingly software agent technology was promoted as offering unparalleled benefits when compared with traditional software-based approaches, and it was anticipated that agent technology would become highly utilised (Kotz and Gray, 1999). However, software agents have not become as pervasive as some researchers may have envisioned. One of the primary reasons for this may be the absence of acceptance within the commercial sector. When Kotz and Gray examined mobile agent technology in 1999 (Kotz and Gray, 1999) they identified that there was a lack of a 'killer application', and this problem can easily be related to the general concept of agent technology. The software agent paradigm is in many respects a new and powerful programming paradigm, and its use can lead to efficient applications. However, most particular applications can be implemented just as cleanly and efficiently with a traditional technique, and therefore the advantages of agents are modest when any particular application is considered in isolation. Kotz and Gray argued that instead researchers must present a set of applications and convince accordingly that the entire set can be implemented with much less effort (and with that effort spread across many different programming groups).

There are also many technical hurdles to overcome with agent technology. Current agent systems are often written in relatively slow, esoteric or interpreted languages for portability and security reasons. Portability is very important in the design of agent technology as frequently agents will travel over a network to be deployed onto remote hosts which may be utilising heterogeneous technologies. Security challenges are a natural consequence of this behaviour - how can we trust our agent will not be tampered with or given restricted access to resources, and how can the host trust that our agent will not commit malicious acts, intentionally or otherwise? In addition to these problems, the existing communication

protocols utilised in distributed systems may not be sufficiently expressive to be used effectively by software agents. As stated by Wooldridge agents will need to interact "not simply by exchanging data, but by engaging in analogues of the kind of social activity that we all engage in every day of our lives: cooperation, coordination, negotiation, and the like"(Wooldridge, 2002, p. 1). Although we feel that it is worth drawing attention to these problems we cannot expect to solve them all in a project of this scope, and therefore we will not discuss security or communication challenges further as we believe these are issues for agent platform designers. However, recently Denti and colleagues (Denti et al., 2005) have argued that the subject of portability directly relates to the construction of a reasoning component intended for agent-based deployment, and therefore we believe this topic is very important to our work.

The challenges presented by the portability of agent code are twofold. Firstly, as agent technology is designed to be portable, so must any component. The component must be capable of being transmitted across the network with the agent, and in order to ensure the component is robust it should preferably operate as a standalone module with limited reliance on other components. In addition a clear interface or Application Programmers Interface (API) should be presented to allow agent developers to easily take advantage of the functionality offered. Secondly, the portability of agent code can mean that frequently the operating requirements of the agent can change. For example, agents may be forced to operate in resource-bound conditions and therefore the design of agent components should be as flexible and light-weight as possible. Denti and colleagues argue that the call for minimality on a component means that a high degree of configurability is its necessary counterpart (Denti et al., 2005). In particular, configurability should be dynamic, so as to face the openness and rapid change of most application environments such as the Internet. Agents may also be required to reason under real-time constraints (Capobianco et al., 2005) and so this puts further emphasis on the need for flexible operation and high efficiency. In order to create agent components that meet these challenges, particularly components capable of argumentation-based reasoning which is known to be computationally complex, several non-trivial design decisions need to be made.

An additional limitation for agent development is the absence of well-defined toolkits for agent construction that are capable of dealing with some of the technical challenges. There are several efforts with implementing these toolkits, but the implementations are frequently proprietary or the services offered fail to meet all the requirements of a project. Examples of popular agent toolkits include JADE (Bellifemine et al., 2001), Jackdaw (Ltd., 2006) and Jinni (Tarau, 1999). Typically these toolkits provide an API for agent and platform development, specify components that should be implemented within the platform and offer various services. For example, JADE offers support for FIPA's extensive agent messaging service, Jackdaw offers good support for code mobility and large-scale deployment, and Jinni allows Prolog code for knowledge representation to be run seamlessly alongside agent code. However, each implementation tends to specialize on what the authors believe is most important and often neglects other important aspects which may prevent any single platform from becoming the de facto standard.

3.2.2 What can be done?

Providing a good demonstration application for our reasoning engine would be useful, and even though we cannot expect to come up with a 'killer application', a sound demonstration of the combination of software agents and argumentation technology would provide motivation for the continuation of this work. The problems associated with the technical challenges of agent code are improving as Sun Microsystem's Java platform is fast becoming a de facto standard within the agent community (particularly within academic work). Java was built from the ground up as a network aware platform, and as such supports the foundations of many of the necessary features for agent technology, such as dynamic downloading and execution of portable code, control of access to and use of resources by executing code in a virtual machine (JVM) and the safe execution of untrusted code through the use of restricted "sandbox" environments. Although traditionally the execution efficiency of the Java platform did not compare favourably to other contemporary platforms, incremental releases of Java have focused much attention on this issue and performance has improved greatly. An additional reason for the Java platform becoming the de facto standard is the increased support for execution of additional language being added to the JVM (Tolksdorf, 2007). The remaining challenges associated with the design of our reasoning engine component, such as portability and the use of agent toolkits will be addressed further in Chapter 6, which is the discussion of our current work.

3.3 Challenges with argumentation

3.3.1 How difficult can it be to argue?

There are many challenges to be faced when attempting to implement argumentation-based reasoning. We have already hinted that computational complexity is potentially a big issue, and is something that may be preventing this type of logical reasoning formalism from being exploited in existing applications. However, before we discuss this subject and associated challenges in more detail there are several other important issues that directly impact on complexity. The first challenge when designing an argumentation system is determining the most appropriate underlying logical machinery to use. This includes choosing the type of logical language to reason with and deciding how arguments are accepted resulting from some judging process, or in other words which semantics to reason with. The literature reveals two broad classes of logics used within argumentation systems - highly expressive but complex first order logic, which has been argued as essential for real-world reasoning by the likes of Pollock (1992), and Besnard and Hunter (2006), or a less expressive, but more tractable defeasible logic, proponents of which include Nute (1994), Maher et al. (2001) and Garcia and Simari (2004). We have already briefly discussed the various judging processes but for a more detailed overview related to classical logic the interested reader is referred to Besnard and Hunter's discussion (Besnard and Hunter, 2006), and for an overview of argumentation-theoretic semantics Prakken and Vreeswijk's discussion (Prakken and Vresswijk, 2002) of the topic and associated challenges is recommended.

When the logical language and semantics have been decided, the next challenges faced are connected with the design of algorithms used to manipulate the underlying components.

Disregarding the associated complexity issues for the moment, the first challenge presented is determining the motivation for the algorithm - do we want our agent to compute all arguments, extensions etc or do we want our agent to determine answers to individual queries made in a more ad hoc manner? The difference between total and query answering algorithms has been mentioned previously in this report, and both approaches have there pros and cons. A total algorithm would especially be relevant to an agent deployed into a small-scale and relatively static environment. The agent may desire to identify all the arguments that are available resulting from both his internal knowledge and beliefs about the external environment in order to make the best decision. Traditional agent-based approaches to this type of reasoning utilise elements that are typically of a propositional nature and enforce that the number of elements constructed remains within reasonable bounds. In such situations it would be reasonable to expect that all argument elements could receive a status-assignment. On the other hand, an agent may be operating in a highly dynamic and complex environment which will require the use of argument systems that are based on first-order languages or equally expressive languages. Arguments in first order systems are constructed dynamically and therefore cannot be known in advance. Vreeswijk (2006) argues that these systems can only rely on query answering algorithms. An additional advantage of a query answering algorithm is that it can in principle take on the task of a total algorithm, simply by enumerating all arguments and querying each argument as it is enumerated. In fact work of Dimopoulos et al. (2000) suggest that in terms of complexity such brute force methods are perhaps the best we can achieve.

As we have hinted throughout the discussion in this section, the major challenged faced when utilising argumentation-based reasoning is that in general it is inherently computationally complex. With first order languages even finding the basic units of argumentation is computationally challenging (Besnard and Hunter, 2006). Typically a minimal consistent subset of the available knowledge base is presented in support of a argument, but as reminded by Besnard and Hunter (2006) determining whether a set of formulae in propositional calculus is classically consistent is an NP-complete decision problem (Garey and Johnson, 1979) and worse still determining whether a set of first order formulae is classically consistent is an undecidable decision problem (Boolos et al., 2002). In addition to determining whether the formulae are consistent, deciding whether a set of propositional classical formulae entails a given formula is a co-NP-complete decision problem (Garey and Johnson, 1979). Although less expressive languages may allow the units of argumentation to be found easier, complexity problems are encountered when trying to design algorithms to determine whether an argument is acceptable with regard to a certain semantics (Dunne and Bench-Capon, 2002). For example, deciding whether an argument is acceptable using the popular stable semantics is an NP-complete decision problem, and utilising the preferred semantics may be even harder (Dimopoulos et al., 2002). However, all these results are based on theoretical worse-case analysis of algorithms, and several members of the argumentation community have recently discussed that empirical evaluation may be necessary to determine whether these theoretical findings are cause for valid concern.

Despite the importance of experimental studies to the area of argumentation, there has been little reported in the literature. While several algorithms have been published and some implementations described, the results are far from conclusive. This state of affairs can be attributed to the lack of systematic experimentation with implemented systems

(with the exception of work conducted by Maher and colleagues (Maher et al., 2001), which will be discussed later). To test and experiment with software systems we require easily generated, realistic and meaningful test instances. One possible approach frequently used in experimental research, as utilised by Maher and colleagues in evaluating defeasible reasoning systems is to generate data randomly. This method offers an unlimited number of test cases and often the user has control over at least some parameters of data generated. However, according to Cholewinski and colleagues (Cholewinski et al., 1999) resorting to randomly generated programs and theories, a solution often used in other areas such as graph algorithms or satisfiability testing, may also not be an ideal approach. First it is difficult to argue that randomly generated data have any correlation with cases that are encountered in practical situations. Second, only a very careful selection of parameters makes randomly generated instances difficult to solve and hence useful for benchmarking purposes.

Another approach, as discussed by Cholewinski and colleagues (Cholewinski et al., 1999), is to produce a collection of real-life problems. Such benchmarks are now used in several areas of experimental research in computer science. The benefits of this approach are evident - the problems are real and thus meaningful. In addition, they are easily disseminated. However, there are also drawbacks. The data often does not provide enough flexibility to allow fully-fledged testing. In particular, a comprehensive study of performance scalability cannot be easily be conducted as databases of benchmarks rarely contain families of test cases of similar structure and growing sizes that would allow good extrapolation of running time. Accordingly, another key challenge associated with the acceptance of argumentation-based implementations is the need to develop experimental testing methodologies and to make these and any associated large-scale data sets publicly available to allow verification of applications.

The discussion of the production of large-scale argumentation-based datasets raises other important challenges, perhaps with knowledge representation in general. Although these issues will be outside of the scope of a project of this size, it is interesting nonetheless to be aware of them. An example of such a challenge is how data, such as knowledge and argument structures, should be represented so as to be accessible. Current work on the Argument Interchange Format (Chesnevar et al., 2006a) may go some way towards this problem. A more fundamental question includes how the data is to be interpreted by disparate agents i.e. the problem of ensuring that agents are sharing a common understanding or meaning of symbols used (Ogden and Richards, 1956). Work on defining and creating ontologies to specify domain specific and general purpose machine readable concepts may be the answer to this problem, but much work remains, especially with regard to inferring new knowledge using such formalisms (Antoniou and Bikakis, 2007).

3.3.2 What can be done?

Designing an algorithm to conduct argumentation-based reasoning is clearly not a trivial task. Accordingly, as our primary goal is to incorporate argumentation into software agents the best decision may be to focus on implementation issues rather than the design of a novel algorithm itself. Much progress has been made on developing algorithms for argumentation. For example, Cayrol et al. (2003), Jakobovits and Vermeir (1999b), Vreeswijk (2006) and

Amgoud et al. (2005) have all published algorithms that could be implemented. As part of our current work (see Chapter 6) we have evaluated several of these algorithms, considering the underlying logical language and semantics and also the reasoning requirements of an agent in order to determine which is most appropriate for inclusion in a prototype of our engine. The previous section of this chapter clearly indicated that dealing with computational complexity issues may be a big part of our final work, and although the initial stages of the research will most likely focus on creating a framework for incorporating argumentation algorithms into an agent-based reasoning component, it is worth taking the time to review existing approaches to reduce the inherent complexity. Although there has been much discussion in the literature about the computational complexity of argumentation there has been comparatively little progress made in developing techniques for overcoming the computational challenges of constructing arguments. Of the theoretical work that does exist the techniques can be broadly divided into four categories - knowledge compilation, techniques for intelligently querying an inference engine, approximating arguments and argument tree pruning.

There are several approaches to knowledge compilation in the literature. The first technique, presented by Besnard and Hunter (2006), is referred to as argument compilation (their discussion relates to algorithms working in first order logic, but they argue that the techniques could be generalised to other formalisms). Argument compilation is a process that improves inefficiency by finding all the minimal inconsistent subsets of the knowledgebase and then forming a hypergraph from this collection of subsets. Whilst the process of argument compilation is expensive, once the knowledgebase has been compiled, it can be queried relatively cheaply and the algorithm provided in (Besnard and Hunter, 2006) can be used to efficiently construct undercuts for an argument, and by recursion undercuts to undercuts. A similar technique is presented by Capobianco and colleagues, (Capobianco et al., 2004). Their mechanism essentially consists of adding a repository of precompiled knowledge, referred to as a dialectic database, which can be queried more cheaply than creating an entire argumentation tree for each query. The dialectic database contains the set of all potential arguments that can be built from the knowledgebase as well as the defeat relationship among them, and can be understood in more general terms as a graph from which all possible dialectical trees computable can be obtained. In this way the use of precompiled knowledge can improve the performance of argument-based systems in the same way Truth Maintenance Systems assist general problem solvers.

An additional compilation technique for a knowledgebase is stratification, which was highly utilised in making early default reasoning implementations computationally efficient (for example, DeReS (Cholewinski et al., 1999)). Essentially, stratification was used to decompose the set of defaults into sets (strata) in such a way that extensions can be computed in a modular way. Within the DeReS system the strata had to be explicitly stated by the programmer, with the system performing significantly better when a default theory possessed a fine stratification. However, there is little work on the use of stratification within argumentation-based reasoning. In summary, although the first two compilation techniques are initially computational expensive, it could be envisioned that with a relatively static knowledgebase the compilation process could be conducted when appropriate, for example, overnight. The same 'offline' technique may also apply to stratifying a knowledgebase. However, the majority of agent-based environments are unlikely to be static, and therefore

the cost of constantly maintaining or rebuilding the compilation in a dynamic environment is likely to be prohibitively high (although work by Capobianco and colleagues (Capobianco et al., 2004) does attempt to circumvent this problem by restricting compilation to only knowledge that is indicated to be static).

The second category for improving efficiency is based on techniques for intelligently querying an inference engine. The majority of these techniques are similar in principle to knowledge compilation except that the compilations are built as needed 'on the fly'. In classical logic and other formalisms if arguments are sought for a particular claim, queries are posted to an inference engine or Automated Theorem Prover (ATP) to ensure that a particular set of premises entail the claim, that the set of premises is minimal for this, and that it is consistent. Besnard and Hunter present a technique known as contouring (Besnard and Hunter, 2006) which is a principled means for intelligently querying an ATP in order to search a knowledgebase for arguments. They discuss that during the course of building an argument tree there will be repeated attempts to ask the same query, and there will be repeated attempts to ask a query with the same support set. Each time a particular subset is tested for a particular claim more information is gained about the knowledgebase. So instead of taking the naive approach of always throwing the result of querying away, this information can be collated to help guide the search for further arguments and counterarguments. Besnard and Hunter state that even in the worse case, by maintaining appropriate contours the number of calls to the ATP is always decreased, offering an improvement over nave searching for arguments and counterarguments (Besnard and Hunter, 2006). The downside to maintaining contours, or even partial contours, is the amount of information that must to be kept about the knowledge base over time. The authors suggest that when a contour appears to be too large to be manageable it may be better to erase it, and construct the elements of the contour as and when required. Capobianco and Chesnevar (1999) also offer a similar scheme to argument contouring for Simari and Loui's early work on the Mathematical Treatment of Defeasible Reasoning framework (Simari and Loui, 1982). They propose creating a repository of previously computed justifications (containing dialectic trees obtained as answers to previous queries) which they refer to as an Argument-based Justification Maintenance System or a dialectic base. This work was a precursor to the previously mentioned dialectical database, but here only grounded justification trees are stored. When a query is posted to the inference engine the engine begins by trying to solve the query according to the information stored in the dialectic base. If it cannot be solved, then the usual justification process will be started.

As intelligent agents must be able to operate within a dynamic environment an argumentation-based reasoning system must be capable of incorporating new information into its knowledgebase. The key problem encountered when using a technique to intelligently query an inference engine is deciding which elements of the repository of data stored are affected after adding new information, and how appropriate changes should be managed. When a new fact is added to the non-defeasible knowledge the contours or justifications obtained in the past, which are stored by the system, may no longer be valid. To overcome this problem a revision process should be performed every time a new fact is added. However, this is not a trivial task as several interacting features, such as argument consistency and minimality, and the acceptability of argumentation lines in the stored dialectic trees all have to be checked. Although Besnard and Hunter imply that their con-

tours are for use with a static knowledgebase, Capobianco and Chesnevar state that they are interested in using their dialectic bases in a dynamic environment, but do not provide a comprehensive discussion of the associated cost of the maintenance process. We suspect in a similar manner to knowledgebase compilation techniques that if the knowledgebase is changing frequently this cost could be prohibitively high.

The third category of efficiency improving techniques is based on generating approximate arguments. Pollock (1995) was one of the first proponents of using this technique to reduce the inefficiency of defeasible reasoning. His proposal included allowing a defeasible reasoner to draw conclusions tentatively, sometimes retracting them later, and perhaps reinstating them still later and so on. Pollock argued that human reasoning is defeasible in one sense by allowing a conclusion to be retracted as a result of further reasoning without any new input. Therefore, an argument may be justified in one stage of reasoning and unjustified later without any additional input. An argument is "warranted" when the reasoner reaches a stage where for any new stage of reasoning the argument remains undefeated. However, in theory the reasoner can be stopped at any time and asked for the current justified conclusions, which would be an approximate answer. Besnard and Hunter also present a framework for approximate arguments (Besnard and Hunter, 2006) that can be utilised for producing useful intermediate results when undertaking argumentation using an ATP. Rather than throwing away an argument that is found by a call to the ATP not to be minimal or consistent, it can be treated as an intermediate finding, and used as part of an "approximate argument tree". This approximate tree can be built with fewer calls to the ATP than building a complete argument tree, and can be refined, as required, with the aim of getting closer to a complete argument tree. Therefore, finding approximate arguments requires fewer calls to an ATP, but it involves compromising the correctness of arguments. Techniques based on approximation are also used in other approaches to reasoning under uncertainty, and although this report does not focus on probabilistic argumentation systems it is interesting to note that for reducing the inefficiencies of this type of reasoning a strategy for computing approximated solutions is often employed (Haenni, 2001). Typically these strategies concentrate on reducing dramatically the number of arguments computed by using a cost function (such as the number of literals in an argument, or the probability of the negated conjunction) to generate important arguments only.

As many argumentation systems are built using dialectical searches, the final category of efficiency improving techniques is based on tree-pruning. One example of such a technique is presented for Garcia and colleagues Defeasible Logic Programming system (Garcia and Simari, 2004). According to their definition of justification, a dialectical tree is built depth-first and resembles an AND-OR tree, and even though an argument may have many possible defeaters, it suffices to find just one acceptable defeater in order to consider the original argument defeated. Therefore, when analysing the acceptance of a given argument not every node in the dialectical tree has to be expended in order to determine the acceptance (referred to as the label) of the root - α-β pruning can be applied to speed up the labelling procedure. It is well known that whenever α-β pruning can be applied, the ordering according to which nodes are expanded affect the size of the search space. Chesnevar and colleagues (Chesnevar et al., 2000b) further extended the original work by proposing a technique to create an evaluation ordering based on determining the acceptable (or feasible) defeaters which can be efficiently computed according to consistency constraints (avoiding

fallacious argumentation). Essentially, given two alternative defeaters for an argument the one which shares as many ground literal as possible with argument being attacked should be preferred (on average this can dynamically obtain the shortest argumentation line). This goal-oriented way of characterising attack helps to dramatically prune dialectical trees.

A number of authors have also looked at using extrinsic factors to facilitate the process of argumentation, such as the social value of arguments (Bench-Capon, 2003) or how an argument resonates with an audience, based on empathy or antipathy (Besnard and Hunter, 2006). In the same way as people naturally focus on the most relevant arguments, it is often possible to measure the relevance of supporting and refuting arguments by the use of a corresponding cost or utility function. Although there has been limited work conducted, it is believed that a utility function such as this could be used to selectively prune dialect trees or reduce the number of arguments generated (Haenni, 2001). However, the computation of the utility function can be seen as a separate decision problem (Anthony Hunter, personal communication), and may add to the computational complexity of the argumentation process, potentially offsetting any improvements. Further investigation into this technique is needed.

As discussed in the previous section of this chapter, the final challenge associated with argumentation is the absence of empirical analysis of algorithms and associated implementations. There are several approaches to overcome this problem. The first is based on the fact that problems in logic are well-known to be hard to solve in the theoretical worst case, but these worse case scenarios may not always be representative of the practical problem. Vreeswijk (2006) discusses that in (Dimopoulos and Torres, 1996) it was proven that the preferred membership problem - and hence the admissible membership problem - is NP-complete. From this it could be concluded that the admissible membership problem has been "solved". However, he feels that this is a non-productive viewpoint. Vreeswijk argues that many argumentation tools are in need of an algorithm to compute grounded or admissible defences, and it may well be that in spite of the results from a worse case analysis there exist algorithms that perform acceptably in average or typical cases (Vreeswijk, 2006). He continues by discussing that a possible line of research that is not currently being explored is to empirically test an algorithm's complexity. An empirical analysis basically amounts to running the algorithm over multiple cases and measuring the amount of elementary computation steps the algorithm has executed on average. Nudelman (2005) describes in detail how to conduct such experiments. Vreeswijk argues that the implementation details and average case complexity analysis included in his recent work should provide enough material to define these tests for his proposed algorithm. He concludes his discussion by stating that although he did not conduct empirical analysis is his work he is in favour of such a technique. However, he strongly believes the presentation of an algorithm must be accompanied by a conventional complexity analysis first, before it can be subject to practical tests.

Additional approaches proposed to empirically evaluate nonmonotonic reasoning system may also be applicable to argumentation-based implementations. Maher and colleagues have created a tool named DTScale (Maher et al., 2001) which is capable of generating parameterised problems to experimentally evaluate the defeasible reasoning system they have implemented. The authors admit they have not yet been able to create realistic random problems, but the test theories that can be generated include a fairly comprehensive variety

of scenarios including theories containing undisputed inferences, circular inferences (which are unsolvable) and inferences involving various widths and depths of rule trees. The use of this tool is discussed further in Chapter 5, the review of existing reasoning engines. Cholewinski and colleagues also propose a similar tool called TheoryBase (Cholewinski et al., 1999) which is used to evaluate DeReS, their implementation of default logic. TheoryBase is based on the Stanford GraphBase graph generating system which is utilised to empirically test graph solving algorithms, and is capable of generating parameterised families of default theories of similar structure and properties, and of sizes controlled by a numeric parameter. GraphBase, and consequentially TheoryBase, provide two additional advantages when compared with existing random problem generators. Firstly, the algorithms within the application root the graphs (and hence default theories) they generate in objects such as maps and dictionaries in an effort to ensure some correlation of the graphs generated to real-life problems. Secondly, every default theory generated gets a unique label (or identifier) which allows easy reconstruction of test cases generated. Cholewinski et al. (1999) argue that this overcomes two of the fundamental problems with generating random problems, and we are curious whether such a technique could be applied to generate a random argumentative knowledgebase utilising existing real world problems. This process of practical experimentation is definitely a challenge that we would like to investigate further, and more discussion can be found in Chapter 7, an overview of our planned future work.

3.4 In summary - Combining the technologies

This chapter has attempted to identify the inherent challenges with both software agents and argumentation-based reasoning. The incorporation of argumentation within software agents may add much needed flexibility to an agent's reasoning abilities, increasing the chances of successful deployment into large-scale dynamic environments. However, our discussion has illustrated that some of the requirements and restrictions of agent-based computing may impact on the design and functionality of argumentation algorithms, and in a similar fashion the inherent computational complexity associated with argumentation may amplify some of the existing challenges related to the design of agent components. The discussion presented in this chapter has also illustrated that although complexity issues may be a big part of our work, our first focus should be on establishing a technical framework to allow incorporation of an argumentation algorithm into an agent-based reasoning component. Clearly, any reasoning component will need to be flexible due to the different reasoning requirements of an agent. One potential solution to this problem would be to offer support for a multitude of logical reasoning formalisms and associated semantics.

Our review of theoretical techniques to increase the efficiency and performance of argumentation systems has been very interesting, and we are keen to see if any of these theories can be incorporated into our implementation. Our discussion has illustrated that some of these techniques would appear highly suitable for use within agent-based technologies. For example, the use of approximating arguments would appear to be a beneficial technique when the available computing resources are not known before the reasoning process begins. The agent could attempt to build an approximate argument to support a query, and continually refine the argument until computational resources are exhausted. The use of

tree pruning techniques would also appear to improve efficiency, by dramatically reducing the number of arguments generated when performing a dialectical-style analysis. However, the use of additional techniques that at first glance appear to be highly beneficial, such as knowledgebase compilation and intelligent querying of the inference engine, may be problematic within an agent-based deployment. The environments into which agents will be deployed will frequently be highly dynamic, and therefore the cost of maintaining useful and accurate compilations or repositories of data may be prohibitively high. Although much has been learned by examining the theoretical work, a review of existing implementations that combine the two technologies is essential. Work in this area is limited, but it will be interesting to identify potential domains of application, determine logical machinery and algorithms utilised, study technical implementation details, examine how the essential challenges associated with complexity have been handled and analyse any empirical evaluation conducted.

CHAPTER 4

Review of Current Argumentation in Agents

4.1 Overview

With the previous chapter presenting an overview of the associated challenges of incorporating argumentation-based reasoning within software agents, this chapter discusses previous work on combining the two paradigms and examines how (if at all) the identified challenges have been overcome. It should be stated that this review only consists of practical work on argumentation-based reasoning within agent implementations (i.e. the use of a reasoning engine), and does not focus on other work related to argumentation that would be irrelevant, such as the implementation of argumentation-based dialogue protocols within multi-agent systems.

4.2 Agents that reason and negotiate by arguing

In Parsons and Jennings (Parsons et al., 1998) a formal and implementable model of argumentation-based reasoning and negotiation for autonomous agents is presented. Their goal (and proposed 'killer application') is to facilitate the process by which a group of agents communicate with one another to try and come to a mutually acceptable agreement on some matter. The model indicates how agents capable of flexible and sophisticated argumentation can be specified both in general terms and in terms of an agent implemented using the Belief-Desire-Intention framework. Parsons and Jennings show in an abstract manner how agents can construct arguments to justify their proposals, critique proposals and exchange arguments to help guide problem solving behaviour. The specification of this functionality is given in terms of what is referred to as the multi-context approach. This approach provides a framework which allows distinct theoretical components to be defined and interrelated. Different contexts within the system represent different components of an agent's architecture, such as plan execution or situation awareness, and interaction between the components is specified in terms of bridge rules. Parsons and Jennings argue that this makes it possible to move directly from the logical specification of the architecture to a formal description. Additionally, since each context contains a set of statements in a logic along with axioms of that logic, it is possible to move directly to a concrete implementation in which the various context are implemented as concurrent theorem provers which exchange information. This division of responsibility within the architecture means that efficiency can be improved in comparison to a monolithic approach as each context and associated theorem prover will only be manipulating a limited amount of elements,

Parsons and Jennings present no concrete realisation of their framework, and hence there is no discussion of technical implementation details. There is a brief discussion on efficiency issues, such as the previously mentioned approach to utilising multiple contexts to divide the workload on the inference engines. The computational complexity of the argumentation

process is clearly dependent upon the language in which the arguments are built (which is not specified in this paper). If the language is full first order logic then Parsons and Jennings state that the problem of building arguments is semi-decidable. Similarly if the language is full propositional logic, the problem is decidable but NP-complete. They recommend that the language is restricted to the language of propositional Horn clauses, which means that the problem of building an argument is not only decidable but also may be achieved n time proportional to the number of propositions in the language. In summary this work provides an appealing overview of how a complete framework for negotiating agents utilising argumentation-based inference may be specified, but it leaves the concrete implementation details of incorporating this type of reasoning, which we are interested in, for future work.

4.3 Argumentation framework for negotiating autonomous agents

In (Schroeder, 1999) Schroeder outlines a framework to allow cooperation and negotiation within a multi-agent system in order to facilitate modelling of business processes. Schroeder begins by illustrating a declarative argumentation framework for a single agent which is then adapted into a multi-agent system. The main objective of his work is to build an operational and efficient implementation and therefore a top-down and goal-directed proof procedure to compute the argumentation process is defined. The argumentation framework provided is closely related to the semantics of logic programs, and therefore to compute the argument acceptance process in a goal-directed manner Schroeder takes advantage of a proof procedure developed elsewhere based on the well-founded semantics (which is stated as equivalent to the argumentation semantics utilised). Limited information is provided about the architecture of agents within this system except that they are defined as a tuple consisting of arguments, domain information, a flag indicating whether the agent sceptical or credulous, and a list of argumentation and cooperation partners. How an agent could assign preference to various arguments is also discussed briefly, and ultimately the computationally simple approach of using a fixed order of partial arguments at design time is chosen. Schroeder specifies one of the current limitations with the implementation is that agents are not capable of performing updates and observations of the environment, but this could be addressed by future work.

The implementation is described briefly, but includes no details regarding the technology utilised or, with the exception a simple demonstration and screenshots, no results of empirical evaluation. Complete details of the inference engine and implementation of the argument judgement process are not given and unfortunately the website specified as containing the implementation is no longer available. In conclusion the paper does provide an interesting overview of incorporating computationally efficient argumentation-based reasoning into agent technology, in particular the effective use of a query answering algorithm, but in our opinion lacks sufficient detail necessary to convince us of the feasibility and practicality of this particular approach.

4.4 Deliberative Stock Market Agents

Garcia and colleagues present a framework for implementing a deliberative multi-agent system in (Garcia et al., 2000)which can be used as a proactive tool for expressing and putting

to work high level stock trading strategies. The agents described in this work are able to monitor the stock market, extract information, and using the domain knowledge provided in the form of defeasible rules perform defeasible reasoning in order to achieve the established goals. The project combines Jinni (Java INference engine and Networked Interactor) (Tarau, 1999), a toolkit and platform for building intelligent autonomous agents, and an implementation of Defeasible Logic Programming (DeLP) (Garcia and Simari, 2004), which provides the agents with the capability of reasoning using defeasible rules in a dynamically changing environment. Jinni is essentially a multi-threaded logic programming language (consisting of lightweight Prolog interpreters), intended to be used as a flexible scripting tool for gluing together knowledge processing components and Java objects in distributed applications. As such, the framework allows agents to coordinate and cooperate though the use of an asynchronous blackboard. The agents have a reasoning module, based on DeLP, where knowledge can be expressed in the form of defeasible rules, and the inference engine is capable of formulating arguments and counterarguments in order to decide whether to perform an action or not.

The agents responsible for committing user actions consist of domain knowledge in the form of a DeLP program, a defeasible reasoning module implemented as a DeLP interpreter, and the specification of goals and actions to be performed. Agents are implemented as a Prolog program, and an appropriate user interface provides a way of specifying the agent behaviour without knowing how to program in Prolog. The strict and defeasible rules are also specified using a graphical interface. The DeLP interpreter is accessed by an agent though the use of custom Prolog predicates provided by the Jinni API, and although reuse of the engine in this format would require the use of the Jinni framework, we can also conclude that in principle this implementation of DeLP is suitable for deployment as a standalone reasoning component. The ability to make real-time decisions on certain rules as the changes occur would appear to be an important aspect of working with stocks, but the paper does not offer any concrete discussion of performance or indicate results of empirical evaluation. However, the standard version of DeLP does implement tree-pruning techniques to improve performance, and anecdotal evidence provided by Garcia and colleagues (Garcia et al., 2000) states that the implementation performs reasonably when deployed in a small-scale demonstration environment.

4.5 An argument-based framework to model an agent's beliefs in a dynamic environment (ODeLP)

Observation-based Defeasible Logic Programming (ODeLP) (Capobianco et al., 2005) is an argumentation-based formalism for agents reasoning in dynamic environments. ODeLP builds on the foundation provided by DeLP (Garcia and Simari, 2004), which is intended to model the behaviour of an agent in a static scenario. As with DeLP, the ODeLP formalism uses a knowledge representation language in the style of logic programming and an argumentation-based dialectic style of inference. Capobianco and colleagues discuss that the majority of approaches to agent-based reasoning that utilise logic programming rely on Extended Logic Programming (ELP) as an underlying formalism. Thus, the agent's knowledge is codified in terms of an ELP program and the well-founded semantics of the program represents the agent's beliefs. Although ELP is expressive enough to capture

different kinds of negation (strict and default) it has limitations for modelling incomplete and potentially contradictory information. In a multi-agent system it is common that agents require such capabilities, as they interact with the environment and among themselves, processing new inputs and changing dynamically their beliefs and intentions. Clearly, in such a setting, the argumentation formalism underlying such systems should be able to effectively incorporate new information into the knowledgebase of the agent and reason accordingly. To provide the agents with the ability to sense the changes in the world and effectively integrate them into its existing beliefs, ODeLP adapts the DeLP knowledge representation system to handle dynamic perceptions.

ODeLP models the beliefs of an agent in a simple way. Given a program P representing an agent's knowledge, a literal Q is believed by the agent iff Q is warranted. Consistency is a basic property for agent's beliefs, in the sense that it is not possible to believe simultaneously in a literal Q and its compliment. In ODeLP, the mechanism for maintaining consistency and updating the knowledgebase of an agent is simple but effective. Capobianco and colleagues assume the perception is carried out by devices that detect changes in the world and report them as new facts (literals). The actual devices used will depend on the particular application domain, and details are not specified within their work. The authors also make the assumption that the perception mechanism is flawless, and new perceptions always supersede old ones. Capobianco and colleagues continue the discussion by stating that real time issues play an important role when modelling agent interaction, and in an argument-based multi-agent system setting, a timely interaction is particularly hard to achieve as the inference process involved is complex and computationally expensive. To solve this issue, they looked at truth maintenance systems and how the use of precompiled knowledge helped to improve the performance of problem solvers. Accordingly, they have enhanced the behaviour of ODeLP by incorporating dialectical databases which are data structures for storing precompiled knowledge. These structures can be used to speed up the inference process when answering future queries (discussed previously).

Although several demonstrations are provided in (Capobianco et al., 2005) the concrete implementation details of ODeLP are not described, and therefore we can only speculate that this implementation would be similar to the standard DeLP application (which has been discussed in the previous review). It would be very interesting to explore empirical results for this implementation, due to the computationally demanding task of updating data and also the use of dialectical databases to overcome this challenge. However, in common with much of the work with DeLP no experimental results are provided.

4.6 In summary

This chapter has attempted to provide an overview of existing work that successfully combines argumentation-based reasoning within software agents. Firstly, it is interesting to note the variety within the intended domains of application. Equally as interesting is that all of the concrete implementations utilise the same underlying framework for the foundation of their inference process, which is Logic Programming. All of the implementations also utilise a query-based algorithm and only support the use of one type of argumentation semantics. The choice of and limitation in utilising one type of semantics are primarily to be based on computational complexity issues. However, as we have commented in the

previous chapter this may not offer the required flexibility for agent-based reasoning.

Parsons and Jennings work provided an interesting overview of how we may approach designing and implementing a complete argumentation-based reasoning framework, but lacked concrete implementation details. However, their discussion of the common problem within agent-based technology moving from the logical specification of a component to formal description and then implementation (a common problem when attempting to implement agent technology) is thought-provoking. Schroeder's work also provides an interesting discussion, again providing an overview of a complete framework for incorporating computationally efficient argumentation-based reasoning. However, Schroeder's description of the implementation is brief, relying on an existing translation process to transform defeasible rules into Logic Programming, and does not provide motivation for the use of argumentation or convince us as to the feasibility of his approach with regard to practical deployment. In contrast, work by Garcia and colleagues illustrate a scenario where the use of argumentative reasoning provides many benefits (and may in fact be necessary) and they also provide a convincing description of the underlying technology used to implement argumentation. Their use of an existing agent toolkit is interesting and appears to act as a catalyst, combining and glueing together various functional components within their system. However, the fact that the DeLP reasoning engine is implemented as a component within the Jinni framework means that is not reusable outside of this platform. Work by Capobianco and colleagues on ODeLP presents an interesting approach to overcoming the inherent complexity issues. Although in the previous chapter we hypothesised that knowledgebase compilation techniques would not be suitable for agent-based implementations their approach to restricting the compilation to include only static knowledge is novel. We would be very interested to explore empirical evaluations of this technique, and indeed we would extend this desire to all of the implementations reviewed, but apart from anecdotal evidence and simple examples none of the systems examined provide comprehensive experimental results.

This review of existing work, especially the DeLP-based agent reasoning engine, has provided us with much inspiration for our implementation. However, the absence of technical implementation details and discussion of experimental evaluation lead to the conclusion that an additional review of reasoning engine implementations should be performed. In order to provide a comprehensive review we have decided to examine both defeasible and argumentation-based reasoning engines, and also due to the limited amount of work in this domain, we have not exclusively focused on implementations intended for agent-based deployment.

CHAPTER 5

Review of Existing Reasoning Engines

5.1 Overview

This chapter of the report examines previous implementations of both defeasible and argumentation-based reasoning engines and associated literature. This review focuses mainly on the technical merits of the implementation, and as such, references are given to allow the interested reader to explore the underlying theoretical details. For a complete overview and comparison of associated theories either (Prakken and Vresswijk, 2002) (Chesnevar et al., 2000a) or (Besnard and Hunter, 2006, Chap 10) should be consulted. Currently we have examined only systems that rely on symbolic approaches to dealing with inference under uncertainty. Several of the systems reviewed do also offer numerical support, but it is not their primary focus, and as such discussion of these features is minimal. We are not claming that this review is a comprehensive list of nonmonotonic reasoning implementations (and in particular we have not reviewed the many default logic based inference engines), but the we have aimed to include a selection of different inference styles (bottom-up and top-down), algorithm types (query answering and total) and implementation technologies in order facilitate the design and development of our engine.

5.2 Logic programming - the father of automated reasoning

Computational logic arose from the work begun by logicians in the 1950's on the automation of logical deduction, and was fostered in the 1970's by Colmerauer and collegues (Colmerauer et al., 1973) and Kowalski (Kowalski, 1979) as Logic Programming (LP). It introduced to computer science the important concept of declarative - as opposed to procedural - programming. The Prolog language (Colmerauer et al., 1973) became the implementation approximating this ideal. Many defeasible and argumentation-based reasoning systems use LP as a logical foundation when designing algorithms (or are built as "metainterpreters" within an implementation of LP), and as such a brief introduction to this technology will be beneficial to understand the framework provided. For a comprehensive overview of the use of Logic Programming within the context of automated reasoning the interested reader is recommended to consult (Alferes and Pereira, 1996), although it's coverage of argumentation is somewhat limited.

When introduced to the knowledge engineering community Logic Programming quickly became a candidate for knowledge representation due to its declarative nature and the so called "logical approach to knowledge representation" (Alferes and Pereira, 1996). This approach rests on the idea of providing machines with a logical specification of the knowledge they possess, thus making it independent of any particular implementation, context-free, and easy to manipulate and reason about. Consequently, a precise meaning (or semantics)

must be associated with any logic programming in order to provide its declarative specification. Declarative semantics provide a mathematically precise definition of the meaning of a program, which is independent of its procedural executions, and is easy to manipulate and reason about. When reasoning, these semantics typically provide the foundation for total algorithms. In contrast, procedural semantics is usually defined as a procedural mechanism that is capable of providing answers to queries. The correspondence of such a mechanism is evaluated by comparing its behaviour with the specification provided by the declarative semantics. Finding a suitable declarative semantics for logic programs has been acknowledged as one of the most important and difficult research areas of Logic Programming. This challenge has been inherited in the creation of nonmonotonic logics and argumentation.

There were many early attempts at bridging the gap between logic programming and argumentation - the work of Phan Minh Dung (Dung, 1995) is particularly relevant. Within first order logic Dung present a theory for argumentation whose central notion is the acceptability of arguments. He proves that argumentation can be viewed as a special form of logic programming with negation as failure, introducing this as a general logic programming based method for generating meta-interpreters for argumentation systems.

5.2.1 Prolog (1973)

Prolog is an implementation of Logic Programming, and is essentially Horn clause programming augmented with the NOT operator under the Selected Literal Definite with Negation as Failure (SLDNF) derivation procedure (Loyd, 1984). Prolog does not allow the use of classical negation to specify negative conclusions, and therefore these are only drawn by default (or implicitly), when the corresponding positive conclusion is not forthcoming in a finite number of steps. This is the specific form of closed world assumption of the original completion semantics given to such programs. However, there were several fundamental problems with the use of the completion semantics, and to deal with the issue of non-terminating computations even for finite programs and other problems, a spate of semantic proposals were set forth from the late 1980's onwards, of which the well-founded semantics of (Gelder et al., 1991) was an outcome. These semantics deal with non-terminating computations by assigning such computations the truth value "false" or "undefined" and thereby give semantics to every program.

The well-founded semantics deals with normal programs i.e. those with only negation by default, and thus it provides no mechanism for explicitly declaring the falsity of literals. This is what is wanted in some cases. However, this can be a serious limitation in other cases, and explicit negative information plays an important role in natural discourse and commonsense reasoning. In fact, several authors have stressed and shown the importance of including a second kind of negation "not" in logic programming for use in knowledge representation and non-monotonic reasoning (Alferes and Pereira, 1996). These arguments lead to the introduction of classical negation in combination with default negation, in what was coined as "Extended Logic Programming (ELP)" (Gelfond and Lifschitz, 1991). Recently the Well-Founded Semantics with eXplicit negation (WFSX) (Alferes and Pereira, 1996) incorporates into the language of logic programs an explicit form of negation in addition to the previous implicit negation, relates the two, and extends to this richer

language the well-founded semantics.

There are currently many different implementations of the Prolog language, and although many offer comparable performance (over the years the inference process has become relatively fast), the main differences are connected with the deployment platform or designed functionality. Typically rules and facts are imported into the Prolog implementation in a batch-style manner by loading and parsing a text file containing the "rulebase". Prolog is often used as a framework in which other logical systems that are designed to provide additional functionality are built. This is achieved by creating something referred to as a metainterpreter, which is essentially another layer of Prolog code that modifies the standard functionality or augments the inference procedure provided by Prolog. The next section of this chapter will discuss several such implementations.

5.3 Defeasible Logics - Spanning the ages

A development closely related to defeasible argumentation is so-called 'defeasible logic' (Nute, 1994). In both fields the notion of defeat is central - in argumentation defeat is among arguments, but in defeasible logic defeat is typically among rules. As stated previously, although this report is focused on defeasible argumentation, it is useful to examine implementations based on this different approach as they deal with similar logical mechanisms (indeed, they could be seen as a precursor to argumentation systems) and explore issues such as the efficient manipulation of the knowledgebase and querying, and also offer insight into problems such as dealing with the inherent complexity issues. Interestingly, some of the implementations reviewed in this section are described as defeasible reasoning systems, but use an argumentation-based approach to reasoning.

The implementations are reviewed in approximate chronological order, and although this list is by no means exhaustive we have attempted to present the systems that are most interesting with regard to our work.

5.3.1 Nathan (1992)

Nathan (Loui and Simari, 1994) resulted out of early work on logics for defeasible reasoning by Simari and Loui (specifically their Mathematical Treatment for Defeasible Reasoning framework (Simari and Loui, 1982)), and was based on combining ideas from systems created by both Poole and Pollock. At the time the paper was published Simari and Loui felt that Poole's system treated specificity, i.e. the comparative measure of the relevance of information, in an "elegant and usable way, but d[id] not describe adequately when to apply his specificity comparator to interaction among arguments." (Simari and Loui, 1982). They continued by discussing that Pollock's system was "too general for AI's use" (Simari and Loui, 1982) and although he treated the interaction among arguments properly he rejected specificity, which they saw as essential. A series of applications were created over the years by Simari and Loui and their students, each demonstrating the incremental theoretical work that was being conducted. However, this discussion will focus on Nathan, the system that drew most attention. In essence Nathan was an early implementation of a defeasible reasoner utilising specificity to prioritize between generated arguments. Arguments were justified using Pollock's inductive definition, where by applying a series of incremental steps

an argument is classified as "in" or "out" in a series of levels. This bottom-up approach to reasoning determined that an argument is warranted when it is "in" in all remaining levels.

The implementation was written as a metainterpreter within Prolog, and as such provides a very simplistic command line driven interface to the user. The knowledgebase is defined in an identical manner to a Prolog rulebase and contains a finite set of definite clauses and defeasible clauses, possibly containing atoms affect by the neg relation. The neg relation allows the presentation of negative facts in the system and is not related in any way to negation as failure. Issuing the analyse command invokes the query justifier by and starts the process of testing whether there is an undefeated argument which supports the query (full details of the associated proof can be found in (Simari and Loui, 1982)). If the search finds such a justification, one of the argument structures and all possible defeaters that were considered will be displayed. All the justifiers can be obtained by rejecting the answer and forcing the system to keep searching. If the answer is negative, the system will have two possible answers - the query has no supporting argument or even though arguments can be constructed to support it, all of them were defeated. In the latter case, the system will return all the potential justifiers, already defeated with its associated defeaters. Although rather simplistic, this method of representing the argument structure associated with a justification "trace" allowed a user to understand the reasoning process that had been conducted and has been replicated in many other implementations.

No empirical evaluation results were presented for the system, but as was the trend in the literature at the time of publication, the solution to several simple benchmark examples are demonstrated. Simari and Loui identify some limitations in the implementation due to the limits of its underlying resolution-refutation linear-input set-of-support theorem-prover for FOL, but these problems mainly affect the authors approach to planning with the system. The implementation offers limited configuration or extra features, but does include a simple Perl preprocessor for extracting rules from well structured example cases (however, discussion of this feature is limited). Complexity issues are not discussed in great detail (as was typical at the time of publication). However, Simari and Loui did identify that inherent complexity issues with first order logic are overcome by restricting the language used to represent the knowledgebase to Horn clauses, a subset of first-order logic. Although this implementation may be considered simplistic when compared with current applications, Nathan and associated work (specifically the MTDR framework (Simari and Loui, 1982)) created a foundation on which much other work was conducted, in particular work on DeLP (Garcia and Simari, 2004) and also early work on utilizing implementation techniques to overcome the inherent computational complexity issues (Capobianco and Chesnevar, 1999)(both of which are discussed later).

5.3.2 d-Prolog (1993)

d-Prolog (Nute, 1993) is a nonmonotonic extension of the Prolog programming language based on Nute's defeasible logic (Nute, 1994) (which in turn was based on Nute's earlier work on Logic for Defeasible Reasoning (LDR) (Nute, 1988). d-Prolog is implemented as a metainterpreter within Prolog and as such the interface provided is command line driven and offers no API for external applications to utilise. Nute's aim was to develop a logic that is efficiently implementable and therefore he kept the language as simple as possible

(Prakken and Vresswijk, 2002). One unary function neg, a sound negation operator which Nute distinguishes from the built in negation-by-failure operator not, and two binary infix functors representing defeasible rules and defeaters respectively, are added to the basic language of Prolog. To derive conclusions defeasibly Nute introduce a unary functors @ to invoke the defeasible inference engine, and a clause Goal is defeasibly derivable just in case the query $? - @Goal$ succeeds. The implementation can also be instructed to perform exhaustive investigation of queries. In response to the query $? - @@Goal$ d-Prolog will test all possible lines of inference and give an appropriate answer as to the assigned status of the query, 'definitely yes', 'definitely no', 'presumably yes', 'presumably no', 'can't tell'. These two query types essentially provide credulous and sceptical acceptability respectively. Nute uses specificity to decide superiority of rules and to adjudicate conflicts between defeasible rules (essentially a more specific rule is considered superior, and these are determined by looking at the bodies of the two conflicting rules to see if either can be derived from the other).

In Nute's original paper (Nute, 1993) there is no discussion of efficiency issues, apart from the use of Horn clause style rules dictated by Prolog and the obligatory demonstration of benchmark examples (a correct solution to the Yale Shooting Problem is provided). However, recent work by other authors has discussed these issues in relative depth. In work on plausible logic conducted by Rock (Rock and Billington, 2000) (reviewed later), he discusses the polynomial time complexity associated with d-Prolog. In addition, both Maher and colleagues (Maher et al., 2001), and Antonio and Bikakis (Antoniou and Bikakis, 2007) have used the d-Prolog implementation as the standard benchmark for comparison to their defeasible reasoning systems implementations. Frequently when used as a comparison implementation d-Prolog is substantially more efficient when there are no disputing rules, due to the fact that these rules are interpreted directly by the underlying Prolog system (of which modern interpreters are efficient). However, when disputing rules are common d-Prolog performs badly, with time growing exponentially in the problem size (Maher et al., 2001). Work by Antoniou and Bikakis demonstrates that d-Prolog is able to handle a relatively large knowledgebase, with the maximum number of rules limited by the underlying Prolog implementation (standard settings in SWI-Prolog typically allows around 20000 rules (Antoniou and Bikakis, 2007)). However, in results from tests conducted by Maher et al. (2001) d-Prolog demonstrates incompleteness inherited from Prolog when it loops on circular lines of inference rules. However, the test suites are automatically generated, and the authors do not discuss whether these rules could have been coded in an alternatively satisfiable way.

The implementation of d-Prolog offers several configuration options, for example, the use of specificity when solving goals can be turned off, and a whynot predicate is introduced that can provide a "proof-trace" illustrating how a statement is derivable. Nute also introduces the incompatible predicate to indicate clauses that are incompatible. This is added to prevent knowledge engineers from using what might be an intuitive way of representing incompatibility which would have caused infinite looping in d-Prolog (as the d-Prolog engine does not process cuts(!), disjunctions(;) and the built-in negation by failure(not) properly). However, d-Prolog does support a limited form of negation-as-failure by allowing the programmer to "specify for which predicates the 'closed world assumption' is to be made." (Nute, 1993). The relatively flexibility and robustness of this early implementation

of defeasible logic in demonstrated in several practical systems and has also been exploited commercially e.g. for controlling expert systems recommendations (Nute et al., 1990) and logical control of an elevator system (Covington, 2000).

5.3.3 EVID (1994)

The EVID (Causey, 2003) (Causey, 1994) reasoning implementation is mainly intended for practical applications in decision support systems. EVID is written as a shell application (a Prolog metainterpreter) that is used to interpret other programs written in Prolog syntax, and is capable of performing defeasible reasoning. Interaction with the application is conducted through the standard Prolog-style command line interface, and a knowledge engineer must write an application program (the knowledgebase in standard nomenclature) to use together with the EVID shell. A knowledgebase consists of definite (non-defeasible) and defeasible rules, which are written using specific predicates provided as part of the shell. This process does require technical knowledge for rule construction and assumes that the user understands negation by failure extensively. An application knowledgebase also typically includes some relative and absolute defeater rules. The user runs both EVID and the knowledgebase in the Prolog compiler and typically states particular data (atomic sentences) into this system and queries it about consequences of these data.

Interestingly, Causey states that at the time the paper was written there is no consensus regarding what should be the adequacy requirements for a defeasible reasoning system. We conjecture that this is still an open problem today. Besides the issues about formalizations, he also discusses that there are other important questions about the nature of practical defeasible reasoning and how it can be represented in a computational logic system. Causey then continues to specify some desirable functional requirements of an ideal interactive defeasible reasoning system, such as having access to a detailed explanation of justifications and allowing the user to over-ride any conclusions made by the application, which he then attempts to implement in EVID. Although Causey does not conduct any empirical evaluations of EVID (or deliberate complexity issues in much detail), his discussion of adequacy does appear to support our view regarding testing implementations of defeasible reasoning "Unfortunately, much of the current literature, both formalistic and computational is guided by a body of simple and eclectic examples about birds and penguins etc. These test cases are useful, but I do not believe that they provide very good understanding of the typical uses of defeasible reasoning in practical contexts." (Causey, 1994).

EVID offers detailed configuration of explanations of justifications using the "why" predicate, which generates a "proof-trace" for a justification of a conclusion (citing all of the supporting evidence or reporting defeats to the user). Causey states that in some applications there will be times when the user wants to know what additional user-input data would enable the system to infer a conclusion which it currently cannot, including what additional data would server as either relative or absolute defeaters. Accordingly, he has included other predicates in the EVID shell such as "howdefeatit", which shows how a currently justified conclusion can be defeated and "howgetit" to determine how a currently defeated conclusion could be obtained. EVID also allows a user to defeat any conclusion (promoting Causey's theory of the user being able to always "have the last word" (Causey, 1994)), either relative to particular supporting evidence, or absolutely, providing that the

user does not contradict himself in so doing. Causey states "We do not want the user to perform epistemologically irrational actions. Obviously we would not want the user to add not p and not p and EVID prevents this" (Causey, 1994). However, in the system we are developing we would have to question this restriction in an automated reasoning application. Frequently real world knowledge an agent would be manipulating could be inconsistent. Additional limitations with the EVID implementation include the inability to represent classical negation and also the fact that the systems cannot infer new defeasible rules from the supplied knowledgebase. Causey states these issues are the goal of future work.

5.3.4 OSCAR (1995)

Most of John Pollock's influential work on defeasible reasoning in the last two decades has been devoted to investigating the processes to be performed by an intelligent agent, so that its conclusions and decisions can be considered as rational. Most of his designs have been embedded into OSCAR (Pollock, 1995), a fully implemented architecture for rational agents written in LISP and is based upon a general purpose defeasible reasoner. The principal contribution of OSCAR's defeasible reasoner is that it provides the inference-engine for autonomous rational agents capable of sophisticated reasoning about perception, time, causation and planning, etc. In OSCAR, reasoning consists of the construction of arguments, where reasons are the atomic links in arguments. Defeasibility arises from the fact that some reasons are subject to defeat. The implementation makes use of natural deduction, and arguments are encoded into a global inference graph. The nodes of an inference-graph represent premises and conclusions, and the links between the nodes represent dependency relations. It is possible to conceive of reasoning conducted within OSCAR as a dialectical justification process, although Pollock is often not explicit on this point (**?**). Obtaining the results from reasoning and other interaction with the system is conducted through the LISP shell, and although Pollock states that the defeasible reasoner could be incorporated into agent applications, no explicit API is specified.

Defeasible reasoning in OSCAR consists of two parts, constructing arguments for conclusions and deciding what to believe given a set of interacting arguments, some of which support defeaters for others. The latter is done by computing defeat statuses and degrees of justification given the set of arguments constructed (although outside our current discussion, in OSCAR simple probabilities are assigned to premises and calculated for a conclusion using the weakest link principle). This defeat status computation proceeds in terms of the agent's inference-graph, which is a data structure recording the set of arguments constructed at the current point of reasoning. OSCAR is very flexible with regard to configuration, and although Pollock rejects specificity for argument comparison, he provides "reasons schemas" written in a macro language to allow configuration of reasoning processes for every domain being modelled. The current reasoning schemas are written to support inference in first order languages.

Throughout his work Pollock has also extensively discussed the computational issues associated with defeasible reasoning. In (Pollock, 1995) Pollock argues that perhaps the greatest problem facing the designers of automated defeasible reasoners is that the set of warranted conclusions resulting from a set of premises and reason-schemas is not in general

recursively enumerable. The solution embodied in OSCAR is to allow a defeasible reasoner to draw conclusions tentatively, sometimes retracting them later, and perhaps reinstating them still later and so on. Pollock argues that human reasoning is defeasible in two different senses. He distinguishes between "synchronically defeasible" (a conclusion may be unwarranted relative to a larger set of inputs) and "diachronically defeasible" (a conclusion may be retracted as a result of further reasoning without any new input). Accordingly, in OSCAR an argument may be justified in one stage of reasoning and unjustified later without any additional input. However an argument is "warranted" when the reasoner reaches a stage where for any new stage of reasoning the argument remains undefeated. This is useful if dealing with limited resources. In theory the agent can be stopped at any time and asked for an approximate answer. Although empirical evaluations for OSCAR are believed to be available [Chris Reed, personal communication], unfortunately we were unable to locate them.

5.3.5 Deimos and Delores (2001)

Maher and colleagues (Maher et al., 2001) presents two implemented systems based on defeasible reasoning: one which implements a query answering algorithm (for single query evaluation) and one that implements a total answering algorithm (for computing all conclusions of a given theory). The authors treat defeasible logic as a "sceptical nonmonotonic reasoning system based on rules and a priority relation between rules, [which] is used to resolve conflicts among rules" (Maher et al., 2001). They chose to reject the commonly implemented specificity to resolve conflicts between rules claiming instead that in many domains knowledge is often encoded using priorities. Both systems accept theories in a Prolog-like syntax and allow queries to be solved for the definite and defeasible levels. It should be noted that currently both implementations are limited to only the use of propositional formulae.

The query answering system, Deimos (Rock, 2006a), consists of a suite of tools that supports the author's ongoing research into defeasible logic. The main component of the system is the prover, which implements a backward chaining theorem prover for defeasible logic based directly on the inference rules. The system is implemented in Haskell and much of this code along with the design strategy is common to the Phobos query answering system for plausible logic (reviewed later) which was developed in parallel with Deimos. Significant effort has been expended to overcome inherent complexity issues and make the prover efficient. The two primary mechanisms for this purpose are memoization, which allows the system to recognise that a conclusion has already been proved (or disproved) and saves repeated computation, and also loop checking, which detects when a conclusion occurs twice in a branch of the search tree. Both methods are implemented using a balanced binary tree of data. Loop-checking is necessary for the depth-first search to be complete, whereas memoization is purely a matter of efficiency. Deimos is accessible through both a command line interface and a CGI interface (available at $http://www.cit.gu.edu.au/\ arock/defeasible/Defeasible.cgi$.) The web interface provides full accessibility to the application in a fairly intuitive manner. However, there is limited support provided in the way of an API to allow software developers to manipulate the application as a standalone component.

The system that computes all conclusions, Delores (Maher et al., 2001), is implemented in about 4,000 lines of C and is accessed using a command line interface. Delores is based on forward chaining, but can only produce a proof for positive conclusions. The negative conclusions are derived by the use of a pre-processing transformation that eliminates all uses of defeaters and superiority relations, and then applies the algorithm to the transformed theory. The transformation was designed to provide incremental transformation of defeasible theories, and systematically uses new atoms and new defeasible rules to simulate the eliminated features. (A full treatment of the transformation, including proofs of correctness and other properties can be found in (Antoniou et al., 2001)). Maher and colleagues state that the transformation can increase the size of the theory by at most a factor of 12 and the time taken to produce the transformed theory is linear in the size of the input theory. The algorithm for positive conclusions is similar to the bottom-up linear algorithm for determining satisfiability of Horn clauses of Dowling and Gallier (Dowling and Gallier, 1984). The one key difference is in the data structures: the Dowling-Gallier algorithm keeps a count of the number of atoms in the body of a rule involved in the justification process, whereas Delores keeps track of the complete body. Maher and colleague state "the latter results in greater memory usage, but allows us to reconstruct the residue of the computation: the simplified rules that remain. This residual is useful in understanding the behaviour of a theory" (Maher et al., 2001). However, they do not state whether this residual is utilised within their algorithm or for offline exploration. We suspect the latter, but if it is used by the implementation this could be similar to the argumentation "contouring" proposed by Besnard and Hunter (Besnard and Hunter, 2006).

Efficiency within both implementations is clearly an important factor for the authors as in (Maher et al., 2001) the results for a large amount of empirical experiments comparing the performance (in terms of CPU time) of d-Prolog, Deimos and Delores are presented. However, we would argue that the experimental platform is less than ideal due to the disparate implementation technologies and configuration options utilised. For example, in the experiments presented d-Prolog was interpreted using Sicstus Prolog 3.7 fastcode using the default memory allocation and the timing was measured by the Sictus Prolog statistics built-in. Deimos was compiled using the Glasgow Haskell Compiler 4.04, with optimisation flags and times are measured using the CPUTime library. Delores is written in C, compiled using gcc without optimisation flags and times are measured using the standard time library. Although the implementations dictate that separate platforms must be used, we have to question whether this setup provides a fair comparison for all implementations. To highlight a few of our concerns the d-Prolog implementation is interpreted whereas the other two implementations are compiled, and the author's choice to include only certain optimisation flags is not adequately explained. Although we acknowledge that creating a common platform to test implementations is a lofty goal, it may be necessary to convince others of the true value of an algorithm's implementation.

Empirical experiments documented in (Maher et al., 2001) utilise automatically generated parameterised problems designed to test different aspects of the implementations. As part of the Deimos suite of tools, Maher and colleagues have created a tool named DTScale which is capable of generating these parameterised problems. The authors admit they have not yet been able to create realistic random problems, but the tests include a fairly comprehensive batch of scenarios, for example, theories containing undisputed infer-

ences, chains of defeasible inferences, circular inferences (which are unsolvable), inferences involving various widths and depths of rule trees, and inferences where every literal involved in the justification is disputed. It should be noted that due to design of DTScale, a proof for any of the queries for a generated theory will require the use of all facts, rules and superiority statements. This is conducted to allow a comparison of the d-Prolog and Deimos query answering to the Delores total algorithms used in the test implementations (because by definition a total algorithm would have to examine all rules when identifying all conclusions). Unfortunately the authors do not offer a more realistic scenario to compare just d-Prolog and Deimos with queries that do not exercise all of the rules.

Comparison of the behaviour of d-Prolog and the two other systems on strict and defeasible versions of the problems in the first batch of tests clearly demonstrates the expected overhead of interpretation with regard to direct execution. Interestingly, d-Prolog is substantially more efficient than Deimos when there are no disputing rules, but when disputing rules are common d-Prolog performs badly, with time growing exponentially in the problem size. Deimos does not suffer from this performance hit by avoiding some duplication of work performed by d-Prolog due to memoization. It is worth noting that for certain problems, such as the chains of inference, the loop-checking and memoization of Deimos has no effect, and for in some cases, such as the rule trees problems consisting of more than 200,000 rules memoization increased time by a factor of about 10. Nevertheless, the authors argue that loop-checking is necessary for completeness and the advantage of memoization over the range of problem type can clearly be seen from experimental results. It is also interesting to note that not all experiments could be conducted when the number of rules in a theory becomes too large, due to the memory handling limitations of the underlying Haskell and Prolog technologies used for Deimos and d-Prolog respectively. Except for the direct execution of strict rules by Prolog, the implementation of Delores was shown clearly to be the fastest of the implementations. This is an encouraging result for the authors, especially in circumstances where it is known in advance that the total set of conclusions must be determined, or that all rules will be utilised in the reasoning process. However, we have to question how frequently in a real world scenario the computation of all conclusions will be required, and the difficulty of knowing a prior that all rules will be required in the reasoning process.

On the discussion of complexity Maher and colleagues conclude that the empirical evaluation illustrates that both of their implementations of defeasible logic result in linear execution times as expected - the time for Deimos grows at O(N log N), with the loop checking contributing the log N factor. It is also apparent from the experiments that the overhead introduced to Delores by the pre-processing transformation varies quite significantly from problem to problem and is sometimes extraordinarily high, well above what would be expected for a transformation that increases the size of the program only by a factor of 12. However, the author's comment that the transformation implemented in full Delores did not behave linearly and since theoretically it is of linear complexity there is clearly an engineering issue to be addressed here. Work is continuing on both systems. For Deimos, Maher and colleagues are re-implementing memoization and loop-checking using mutable arrays, instead of a balanced tree in order to eliminate the O(log N) factor. For Delores, they are addressing the problems of initialisation and the pre-processing transformation that were exposed by the experimental evaluation.

5.3.6 Phobos (2001)

Phobos (Rock, 2006b) is an implementation of propositional plausible logic written by Rock and Billington (Rock and Billington, 2000), (Billington and Rock, 2001). Plausible logic is an extension of Defeasible Logic that overcomes the latter's inability to represent or prove disjunctions. Rock and Billington state that recent work on the modelling of regulations has shown that the ability to accommodate disjunctions is important. Much like defeasible logic, a plausible reasoning situation is defined by a plausible description, which consists of a set of facts, plausible rules and defeasible rules. The set of facts describes the known indisputable truths of the situation. The set of plausible rules describe reasonable assumptions, typical truths, rules of thumb, and default knowledge, which may have a few exceptions, and the set of defeater rules disallow conclusions which are too risky, without supporting the negation of the conclusion. The authors state that currently only propositional formulae may be used for the plausible description and queries.

Phobos is the first complete implementation of plausible logic and is capable of deducing or proving formulae in conjunctive normal form at three different levels of certainty or confidence. In decreasing certainty they are: the definite level, the defeasible level, and the supported level. Phobos is implemented in Haskell and is accessible through both a command line interface and a CGI interface (available at: http://www.cit.gu.edu.au/ arock/plausible/Plausible.cgi). The web interface provides full accessibility to the application in a fairly intuitive manner. However, as with Deimos there is limited support provided in the way of an API to allow software developers to manipulate the application as a component. Several tools are offered with the implementation, including Description2Theory which transform a plausible description into a Plausible Theory making it easier for the knowledge engineer to express rules, and also a tool similar to Deimos' DTScale to automatically generate parameterised problems designed to test different aspects of the implementations. The authors utilise these tools to present a multitude of interesting empirical results. It could be conjectured that the increased expressiveness with plausible logic also increases the complexity. However, performance measurements with a variety of scalable automatically generated plausible theories has shown that the prover can operate with theories consisting of thousands of rules and priorities, and the prover has a time complexity per sub-goal that is close to linear (O(N log N) time complexity) with respect to the number of rules in the theory.

Rock and Billington state that planned future work on this implementation includes refining and optimising the time and space complexity of the propositional implementation. For example, storing rules and priorities in lists which must be sequentially searched is the first and easiest target for optimisation. The authors comment that they would like to extend the implementation by adding support for variables, and exploring the help guide found on the implementation website would indicate that a new version of Phobos has been deployed since the paper was written which appears to support variables. However, there is limited information available and no experimental data could be found.

5.3.7 Defeasible Logic Programming - DeLP (2003)

Garcia and colleagues present an implementation of a formalism that combines results of logic programming and defeasible argumentation, named Defeasible Logic Programming (DeLP) (Garcia and Simari, 2004). Although this implementation effectively straddles the boundary between defeasible reasoning and argumentation we have elected to include the review in this section of the chapter. DeLP allows information to be represented in the form of weak rules in a declarative manner and conclusions to be warranted using a defeasible argumentation inference mechanism. DeLP was originally implemented as a metainterpreter in Prolog, and can be accessed through a web interface (available at http://cs.uns.edu.ar/ajg/DeLP.html). Additionally, an abstract machine named Justification Abstract Machine (JAM) has been designed for the implementation of DeLP as an extension of popular Prolog Warrens abstract machine (WAM). Recent research has shown that DeLP provides a suitable framework for building real-world agent-based applications that deal with incomplete and potentially contradictory information e.g. a web-search engine recommender systems (Chesnevar and Maguitman, 2004), a reasoning framework for agents within dynamic environments (Capobianco et al., 2005), and a framework for stock-market trading agents (Garcia et al., 2000). It should be noted that the standard implementation of DeLP is intended to model the behaviour of a single intelligent agent in a static scenario in comparison with ODeLP (Capobianco et al., 2005) (discussed previously) which contains extra machinery to handle dynamic scenarios. DeLP loads strict and defeasible rules via text file (or web form) and outputs results in much the same manner as standard Prolog. However, recent work outlining the use of DeLP within a deliberative multi-agent system for implementing stock market trading strategies (discussed in section X) discusses how the DeLP engine has been incorporated into a reasoning component within the Jinni framework (Tarau, 1999). This component is accessed using custom Prolog predicates provided as part of the Jinni API, and although re-use of the engine in this format would require the use of the Jinni framework we can also conclude that in principle DeLP is suitable for deployment as a standalone reasoning component.

Looking into the logical machinery of DeLP, arguments are built on the basis of a defeasible derivation computed by backward chaining applying the usual SLD inference procedure used in logic programming. DeLP incorporates an argumentation-based formalism for the treatment of contradictory knowledge, and a dialectic process is used for deciding which information prevails. In DeLP a literal will be warranted (i.e. a query succeeds) if there exists a non-defeated argument structure. In order to establish whether the structure is non-defeated, the set of all defeaters for all arguments used within the structure will be considered. Since each defeater is itself an argument structure defeaters for the initial defeater will in turn be considered and so on. More than one argumentation line could arise, leading to a tree structure that Garcia and colleagues refer to as a dialectic tree. During this dialectical analysis certain constraints are imposed for averting problematic situations such as producing an infinite sequence of defeaters or handling circular argumentation lines (the inference mechanism is capable of answering "yes", "no" and "undecided" in response to the query for support for a literal). DeLP does not require a priority relation among rules to be explicitly given in order to decide between rules with contradictory consequences. Garcia and colleague believe that common sense reasoning should be defeasible in a way

that is not explicitly programmed (Garcia and Simari, 2004). They believe defeat should be the result of a global consideration of the corpus of knowledge contained within the agent performing the inference and therefore DeLP uses what is referred to as generalized specificity which allows discrimination between two conflicting arguments. Intuitively this notion of specificity favours two aspects in an argument: it prefers an argument with greater information content or with less use of rules. In other words an argument will be deemed better than another if it is more precise or concise.

Although there is no empirical experimental results available for DeLP there has been recent work on a complexity analysis by Cecchi, Fillottrani and Simari (Cecchi et al., 2006) who conducted a thorough analysis of the complexity issues of DeLP using game semantics. Their results indicate that the problem "is a set of defeasible rules an argument for a literal under a defeasible logic program?" is P-Complete and the problem "Does there exist an argument for a literal under a defeasible logic program?" is in NP. However, anecdotal evidence presented in much of Garcia and colleagues work, e.g. (Garcia and Simari, 2004) (Garcia et al., 2000) (Capobianco et al., 2005), would indicate that DeLP performs reasonably in tests conducted for small-scale real-world demonstrators. Much effort in DeLP has been put towards making the implementation (and the inherent complexity issues with argumentation) efficient and tractable. These efforts can be divided into primarily three categories; dialectical tree pruning, using precompiled arguments in a dialectical database and adding support for parallelism into the argument justification process. Each of these will now be discussed.

The dialectical tree associated with the warrant procedure can become quite large for non trivial situations. Given a DeLP knowledgebase there could be several argument structures for a literal. However, the warrant procedure will not construct all the possible argument structures for this literal. Instead it will consider each one of them in turn, exploring the associated dialectic tree. This optimisation is similar in spirit to the one found in Pollock's OSCAR. Much of the effort expended in the initial implementation of DeLP was put into the task of performing an efficient search. According to Garcia and colleagues definition of justification, a dialectical tree is built depth-first and resembles an AND-OR tree, and even though an argument may have many possible defeaters, it suffices to find just one acceptable defeater in order to consider the original argument defeated. Therefore, when analysing the acceptance of a given argument not every node in the dialectical tree has to be expended in order to determine the acceptance (referred to as the label) of the root - α-β pruning can be applied to speed up the labelling procedure. It is well known that whenever α-β pruning can be applied, the ordering according to which nodes are expanded affect the size of the search space. Chesnevar and colleagues (Chesnevar et al., 2000b) propose a technique to create an evaluation ordering based on determining the acceptable (or feasible) defeaters which can be efficiently computed according to consistency constraints (avoiding fallacious argumentation). Essentially, given two alternative defeaters for an argument the one which shares as many ground literal as possible with argument being attacked should be preferred (on average this can dynamically obtain the shortest argumentation line). This goal-oriented way of characterising attack helps to dramatically prune dialectical trees.

In addition to the tree pruning strategies described above, Capobianco and colleagues (Capobianco et al., 2004) define a mechanism in their ODeLP formalism to limit the ex-

pensive creation of dialectic trees in a dynamic environment where an agents perceptions may be changing rapidly. This mechanism essentially consists of adding a repository of precompiled knowledge that can be queried more cheaply than creating an entire dialectic tree for each query. As discussed earlier in this report there are different options for integrating precompiled knowledge into a reasoning mechanism. A simple approach would be recording every argument that has been computed so far, as proposed in Capobianco and colleagues earlier work (Capobianco and Chesnevar, 1999). However, a large number of arguments can be obtained from a relatively small program, thus resulting in a large database. Many arguments are obtained using different instances of the same defeasible rules, and therefore recording every generated argument could result in storing many arguments which are structurally identical, only differing in the constant names being used to build the corresponding derivations.

The solution to this problem is to store as precompiled knowledge the set of all potential arguments that can be built from the knowledgebase as well as the defeat relationships among them. Capobianco and colleagues refer to this precompiled knowledgebase as a dialectic database (Capobianco et al., 2004). Given a DeLP program its dialectical database can be understood as a graph from which all possible dialectical trees computable can be obtained. Instead of computing a query for a given ground literal, the ODeLP interpreter will first search for a potential argument (and its associated defeaters) in the dialectical database, speeding up the search. In this way the use of precompiled knowledge can improve the performance of argument-based systems in the same way Truth Maintenance Systems assist general problem solvers. It should be noted that in the ODeLP system the set of arguments that can be built from a program also depends on the agents perceptual observations set. When this observation set is updated with new perceptions, arguments which were previously derivable may no longer be so. If precompiled knowledge depends on the observation set, it must be updated as new perceptions appear. Clearly this is not a trivial task, as new perceptions are frequent in dynamic environments and a consequence, Capobianco and colleagues argue that precompiled knowledge should be managed independently from the set of observations.

The final method explored for improving efficiency within the implementation of DeLP is the use of parallelism presented by Garcia and Simari (Garcia and Simari, 1999). Implicitly exploitable parallelism for Logic Programming has received ample attention, and Garcia and Simari argue that DeLP, which adds additional functionality to existing Logic Programming, is especially apt for this optimizing technique. In Logic Programming, OR-parallelism, AND-parallelism, and also unification parallelism can be implicitly performed due to the considerable freedom (i.e. non-determinism) in selecting which reduction paths to follow in order to solve a query. Garcia and Simari demonstrate that these three existing types of parallelism that can be exploited directly by DeLP. However, there are new sources of parallelism that can be implicitly exploited in a defeasible argumentation formalism. For example, several arguments for a conclusion can be constructed in parallel, once an argument is found defeaters can be searched in parallel, and several argumentation lines of the dialectical tree can be explored in parallel. All these sources of parallelism for defeasible argumentation provide both a form of speeding up the dialectical analysis and a form of distributing the process of argumentation. Although they do not provide empirical evaluations of the performance enhancement provided by incorporating such parallelism

their claims appear valid.

5.3.8 DR-PROLOG (2007)

DR-PROLOG (Antoniou and Bikakis, 2007) is an implementation of a system for defeasible reasoning on the web, specifically aimed at the semantic web technologies allowing reasoning with rules and ontological knowledge written in RDF Schema (RDFS) or OWL. The developers of the system, Antoniou and Bikakis, discuss that the development of the Semantic Web proceeds in layers, and the highest layer that has reached sufficient maturity is the ontology layer in the form of the description logic based languages of DAML+OIL and OWL (Antoniou and Bikakis, 2007). On top of the ontology layer sits the logic and proof layers. The implementation of these two layers will allow the user to state any logical principles and permit the computer to infer new knowledge by applying these principles on the existing data. Most studies have focused on the employment of monotonic logics in the layered development of the semantic web. However, DR-PROLOG looks at using defeasible reasoning. The implementation is based on Prolog, and is designed to answer queries.

Although the implementation architecture of DR-PROLOG is discussed in depth, there is limited details regarding the interface, and no specific API is outlined that would enable developers to utilise the inference engine as a standalone component. Strict and defeasible rules can be imported into the implementation using the standard Prolog mechanism of loading a text file. The core of the reasoning system consists of a well-studied translation of defeasible knowledge into logic programs under the Well-Founded Semantics (for more details on each of the translations discussed, see (Antoniou and Bikakis, 2007)). The translation of a defeasible theory D into a logic program P(D) has the goal of showing that p is defeasibly provable in D is equivalent to p is included in the Well-Founded Model of P(D). The authors state the main reason for the choice of well-founded semantics is it low computational complexity. Once the theory and query have been translated into a Logic Program they are solved using a standard Prolog inference engine (currently utilising XSB Prolog).

The implementation appears to be very flexible. Defeasible rules can be entered into the implementation either using the author's Prolog-like syntax for defeasible logic or in RuleML syntax, the main standardization effort for rules on the semantic web. Priorities on rules may be used to resolve some conflicts. Antoniou and Bikakis state that priority information is often found in practice and constitutes another representational feature of defeasible logics (Antoniou and Bikakis, 2007). Only priorities between conflicting rules are used, as opposed to systems of formal argumentation where often more complex kinds of priorities (e.g. comparing the strength of reasoning chains) are incorporated. The implementation offers several configuration options, the most interesting being the option to select ambiguity blocking or propagating behaviour when reasoning. A literal is ambiguous if there is a chain of reasoning that supports a conclusion that p is true, another that supports that not p is true and the superiority relation does not resolve this conflict. Ambiguity propagation results in fewer conclusions being drawn, which might make it preferable when the cost of an incorrect conclusion is high. For this reason the authors state that an ambiguity propagating variant of defeasible logic is of interest to their intended

domain of application.

Antoniou and Bikakis present results of empirical experiment comparing the (CPU-time) performance of their system with Deimos and d-Prolog. They utilise the DTScale tool of Deimos to automatically generate tests suites for the evaluation, focussing on defeasible inference assuming the ambiguity blocking behaviour of the test theories (as both Deimos and d-Prolog do not support ambiguity propagation.) Utilising the same testing methodology as employed the authors of Deimos unfortunately inherits the same problems regarding the use of disparate platforms and configuration. For example, DR-PROLOG is executed using XSB Prolog, d-Prolog using SWI-Prolog and Deimos using Haskell. However, this does not prevent the results from providing an interesting insight into the DR-PROLOG implementation. The results show that the compilation of test theories adds a significant amount of time to the overall execution time of the experiments for DR-PROLOG and d-Prolog. In addition, both versions of Prolog used could not compile all the test theories, because the default memory allocation was exhausted, and as a result theories with more than 20000 logical rules were not tested. In general the performance of DR-PROLOG is proportional to the size of the problem due to the defeasible theories being translated in logical programs with the same number of rules. In comparison with d-Prolog, DR-PROLOG performs better in the cases of complex theories (theories with a large number of rules and priorities). In comparison with Deimos, DR-PROLOG performs a little worse in most of the cases of theories with undisputed inferences. However, DR-PROLOG is designed to support rules with variables, while Deimos supports only propositional rules, and this additional feature aggravates the performance on the system.

Antoniou and Bikakis conclude their discussion of DR-PROLOG with a concrete example of travel packages brokering on the semantic web, and they outline the input files that are parsed. The authors provide an overview of proposed future work including adding arithmetic capabilities to the rule language and using appropriate constraint solvers in conjunction with logic programs, and investigating the applications of defeasible reasoning for brokering, automated agent negotiation, mobile computing and security policies.

5.4 Argumentation - The emerging technology

5.4.1 IACAS (1993)

IACAS (InterACtive Argumentation System) (Vreeswijk, 1993a) was one of the first prototype implementations of argumentation written by Gerard Vreeswijk to do interactive dialectic-style argumentation on a computer. IACAS is written in LISP and originally meant to demonstrate the theory outlined Vreeswijk's PhD thesis on defeasible argumentation, referred to as 'abstract argumentation systems' (Vreeswijk, 1993b). Using this theory as a basis, he explored dialectical issues when modelling the procedure of justification as a debate between two parties, a proponent and opponent. As the system was written in the early nineties the interface is relatively simplistic with input and output conducted via the command line or file system. IACAS uses a language in which propositions, rules (strict and defeasible) and cases are represented. IACAS allows argument for and against a proposition to be generated, and also a disputation to be carried out (in the style of two-party immediate response games), where the system will attempt to find arguments for

and against a proposition and present an ultimate conclusion taking all of the arguments into account. Of particular interest when the implementation was first published was that Vreeswijk allowed a proposition to be "established", "denied" or "undecided". This third category is useful for avoiding inappropriate conclusions (such as illustrated by enchmark problems) and was not offered by many other implementations at this time, notably Nute's d-Prolog (Nute, 1993) (which could cause non-intuitive results to be concluded (Prakken and Vresswijk, 2002, p. 295)).

Vreeswijk (Vreeswijk, 1993a) identifies several reasons how his system could be distinguished from a number of other systems that were prevalent in the literature at the same time, such as Nute's d-Prolog (Nute, 1993), early work on Pollock's OSCAR (Pollock, 1992)and work on Loui et al's LMNOP (Loui et al., 1993) (which was work later incorporated into NATHAN (Loui and Simari, 1994)). First, Vreeswijk states that IACAS allows the user to interact with the system in many ways, meaning that he may set parameters, accommodate output, and tailor dispute records to personal taste. Other systems, such as Nute's and Loui's, did not allow the same amount of interactivity. Second, Vreeswijk states that his system handles, what he refers to as "combinatorics", correctly. He argues that many argumentation systems do not find the correct number of arguments, for example, if six arguments to support a statement are available most argumentation systems come up with only one argument (d-Prolog is cited as such a system) or with two or a potentially infinite number of arguments. To find and use the correct number of arguments is essential to debate and Vreeswijk claims that "IACAS finds the right arguments and finds them all" (Vreeswijk, 1993a). Thirdly, the most sophisticated feature of IACAS was stated as the possibility to analyse the epistemic status of a proposition according to Chisholm's 'Theory of Knowledge' (Chisholm, 1997). Chisholm has a theory in which propositions can, for instance, be 'certain', 'beyond reasonable doubt' or 'counterbalanced'. When IACAS is requested to analyse the epistemic status of a proposition, it initiates a debate on that proposition and its negation and synthesises the outcomes into an epistemic modality.

No complexity analysis or empirical results were presented by Vreeswijk (as was common at the time of publication), although several examples were discussed. However, this early work can be seen as essential to the practical implementation of argumentation-based systems. IACAS demonstrated that the theoretical idea's advanced in Vreeswijk's thesis really did work, and there were few such demonstrations before this implementation.

5.4.2 ASPIC Prototype (2005)

The Argumentation Services Platform with Integrated Components (ASPIC) consortium is an EU FP6-funded project focused on knowledge-based services for the Information Society utilizing argumentation systems. The goal of the project is to create a suite of software components based on a common and sound theoretical framework and a development platform for integrating these components with knowledge (e.g. semantic web) resources and legacy systems. The ASPIC D2.5 deliverable (Amgoud et al., 2005) briefly describes an argumentation algorithm based on the theory presented and a prototype implementation of that algorithm called AS (Argumentation System) written in Ruby by Gerard Vreeswijk. Fundamentally AS is an argumentation program that accepts formulas in an extended first-order language and returns answers on the basis of the semantics of

credulously preferred sets. Currently the implementation provides a web interface (available at http://www.cs.uu.nl/people/gv/AS/) allowing batch-type input into the application (meaning that all input is processed in one pass), and due to the design could easily be modified to accept command line style input. However, no explicit API is specified to allow other systems to interact with the implementation as a standalone component.

Strict and defeasible rules can be entered into the system and typically exactly one query is supplied. The language accepted by AS can be considered as a conservative extension of the basic language of Prolog, enriched with numbers that quantify rule strength and degree of belief. In fact the application accepts Prolog programs (although experimentation has revealed that not all standard library predicates are available). Although outside the scope of the current discussion two kinds of numerical input play a role in AS, namely the degree of belief (DOB for short) and rule strength (strength for short). The DOB is a number b in (0,1] that indicates the degree of belief, or credibility of a single proposition. This single proposition can be a fact or a proper rule. The strength of a rule is a number s in (0,1] that indicates the strength with which the antecedent implies the consequent. All rules possess a DOB as well as an implicational strength. Vreeswijk states that rule strength and DOB are "provided as a means to experiment with different mechanisms of argument evaluation and they are not intended to express probabilities, values from the theory of possibilistic logics, nor do they represent values from other numeric theories to reason with uncertain or incomplete information" (Amgoud et al., 2005).

The core prover of the implementation attempts to find an argument with a conclusion for the query specified and then tries to construct an admissible set around that argument. On the macro level arguments are constructed as nodes in a diagraph, and AS tries to build an admissible set around an argument for the main claim. The web interface of AS is capable of displaying a simplistic graphical representation of this graph, which we have found very useful for understanding the underlying argumentation process. Vreeswijk discusses that the framework for AS is built with flexibility and configuration as an important function, and although currently a large number of algorithmic options are hard-coded into AS, future work could allow a user to specify these options at run-time. For example, argument strength is computed according to a sieve sum but could equally well be computed otherwise. Due to the implementations design it is relatively simple to extend AS's input syntax with flags or command line options that indicate specific algorithmic choices. Vreeswijk's treatment of the implementation includes a discussion of several examples which demonstrate the system's ability to handle increasingly complex examples (included on the web site where the implementation can be accessed). However, with this algorithm and associated implementation no complexity analysis is given.

5.4.3 Vreeswijk's admissible defence sets (2006)

In the same spirit as the ASPIC prototype Vreeswijk presents another algorithm and implementation (Vreeswijk, 2006) of an argumentation system that computes grounded and admissible defence sets in one pass (i.e. without walking the search tree twice) for single argument. The algorithm has also been used to compute defence sets in a knowledge representation architecture for the construction of stories based on interpretation and evidence. As discussed in this report previously, algorithms to compute grounded and/or

preferred extensions have been proposed e.g. (Cayrol et al., 2003), (Jakobovits and Vermeir, 1999b), but these algorithms address one particular semantics, they do not combine the search for different semantics and they are often meant to compute full extensions rather than minimal lines of defence. The algorithm has been implemented in the object oriented scripting language Ruby, and is currently accessed using a web interface (available at $http://www.cs.uu.nl/\ gv/code/grd_adm/$). Much of the interface and language is common with the previously discussed system, and therefore a repeat of the associated advantages and drawbacks will not be discussed here.

Vreeswijk presents an overview of the testing process, discussing that a benchmark suite of typical argument systems (i.e. a collection of typical di-graphs) was composed and is available on the implementations website mentioned. Besides standard problems, the benchmark suite also contains problems that are known to be computationally difficult or conceptually problematic. As of April 2006, this collection consisted of 47 problems and is constantly increasing. Vreeswijk also presents a comprehensive complexity analysis, detailing that in the worse case the algorithm may behave exponentially on the size of it input. Other cases are presented, illustrating that the complexity drastically decreases when the worse case example is slightly modified, and Vreeswijk also presents a preliminary proposal for a definition of what constitutes to be an average case, but does not conduct an analysis of such a case. As discussed earlier in this report, Vreeswijk also suggests that a possible line of research that was not explored in his current work is to empirically test the algorithm's complexity. An empirical analysis basically amounts to running the algorithm over multiple cases and measuring the amount of elementary computation steps the algorithm has executed on average. Nudelman (Nudelman, 2005) describes in detail how to conduct such tests. Vreeswijk discusses that although he did not conduct an empirical test he believes such evaluation would be highly beneficial. He concludes by stating that the presentation of an algorithm must be accompanied by a conventional complexity analysis first, before it can be subject to practical tests.

5.5 In summary

This chapter has presented an overview of existing work conducted on defeasible and argumentation-based reasoning engines. Our discussion has included a review of various types of formalisms, semantics, algorithms and technical implementation details. Clearly several similarities can be identified between the implementations throughout this review. The first striking similarity is that many of the reasoning engines use an underlying Logic Programming framework, and are frequently created as a metainterpreter within an existing Prolog implementation. Although this reduces the effort required to construct a reasoning engine and avoids duplicating inference processes that are common to both Logic Programming and defeasible reasoning, this type of implementation technique does have some disadvantages. From a technical aspect the Prolog language is interpreted, which can result in a performance hit over compiled implementations (as indicated by Maher and colleagues experimental results (Maher et al., 2001)). It also means that the interface is based on a Prolog implementation which typically offers a limited API, particularly in regard to allowing developers to utilise the engine as a standalone component, and also insists that a knowledgebase is loaded in as a text file containing strict and defeasible rules. However,

some of the more recent engine implementations are attempting to overcome these limitations, such as the previously discussed incorporation of the DeLP reasoning engine within the Jinni agent platform framework and DR-PROLOG's ability to load knowledgebase rules in RuleML format.

From a theoretical aspect we can observer that several of the early implementations inherited problems associated with Prolog, such as looping with certain theories, as evident in the tests conducted by Maher and colleagues (Maher et al., 2001) on d-Prolog. In addition early implementations sometimes computed unintuitive conclusion due to the reliance on Prolog semantics which concludes a statement is either provable or not provable (i.e. a statement cannot be labelled as undecidable). However, later implementations have incorporated techniques to ensure these kinds of problems do not occur. For example Maher and colleague's Deimos uses loop-checking to prevent circular lines of inference, and Garcia and colleagues DeLP utilises consistency constraints when generating dialectical argumentation trees to prevent logical fallacies from occurring.

We have also observed several trends over the timeline of implementation developments. The reviews show that in the early 1990's the only form of testing conducted on a reasoning engine implementation was a demonstration of intuitive results and correct inference using a standard (and somewhat limited) batch of benchmark examples. The testing conducted with the Nathan and d-Prolog implementations is a prime example of this technique. Authors of reasoning implementations created in the mid 1990's, such as Causey (EVID) and Pollock (OSCAR), clearly began to appreciate the need for experimental evaluation, as already discussed Causey argued strongly against the use of "a body of simple and eclectic examples about birds and penguins etc." (Causey, 1994) to test defeasible reasoning systems. However, only around the start of 2001 did authors of defeasible reasoning implementation begin to discuss in depth technical techniques to improve efficiency and include comprehensive experimental results within their work. Examples of this type of work include Maher and Colleagues (Maher et al., 2001), Rock and Billington (Rock and Billington, 2000), and more recently work by Antoniou and Bikakis (Antoniou and Bikakis, 2007).

In this chapter we have already discussed several limitations with the experimental testing conducted, and we may have been critical particularly in regard to the platform and application set-up. However, we would like to make it clear that in our opinion any attempt to include empirical evaluation of defeasible reasoning implementations should be applauded and encouraged. It is interesting to note that one of the most complete implementations of defeasible reasoning, DeLP, which has been utilised in many small-scale demonstrators, has not undergone empirical analysis, and instead the authors have relied on providing anecdotal evidence. The example of the value of such experiments can be seen with the evaluation of the Deimos and Phobos implementations which confirmed the expected theoretical complexity results. Perhaps more interesting is that the empirical analysis of Maher and colleague's (Maher et al., 2001) Delores implementation identified efficiency problems with the implementation which indicated that the algorithm had not been implemented correctly, and prompted the authors to analyse their technical implementation in order to determine the problem. Without empirical evaluation this incorrect translation of the algorithm into implementation may have gone unnoticed. Also worth noting is that the evaluations of all of the previously mentioned system allowed the authors

to identify and target the most appropriate areas within their implementation to improve performance. It would appear from many of the reviewed implementations that the choice of data structures used within the core inference engine is very important. Although our review has clearly indicated that there are a limited number of argumentation-based reasoning engines available for analysis, the main contributor of such work, Gerard Vreeswijk, has clearly advocated the need for empirical evaluation argumentative implementations. In addition to promoting the use of, and providing many very interesting benchmark examples (which are much more comprehensive than used within testing of early implementations) he has also been advocating the need to empirically evaluate an algorithm's performance using, for example, techniques discussed by Nudelman (Nudelman, 2005).

To conclude our summary of this chapter we would like to draw attention to several other interesting aspects of existing work. First, buried within Causey's work on the EVID implementation he states that there appear to be no consensus as to what is required within a defeasible reasoning implementation, and although this is something we have not thought about in much detail, we believe this may still be a problem today. Causey ideas on how to approach creating a desiderata for such implementations are interesting, such as insisting an engine must provide detailed explanation of justifications and allowing the user to over-ride any conclusions made by the application, but may need to be updated taking into account the technical and theoretical advances over the last ten years. The use of "proof traces" to facilitate a users understanding of the reasoning process seem to be a universally accepted requirement of defeasible and argumentative reasoning engines, as they were implemented in even the very first implementations. Increasingly these traces have become more advanced, with Gerard Vreeswijk's latest work generating a very useful graphical representation of the argument relations. The inclusion of functionality within the EVID implemenation such as the "howdefeatit" operator, which shows how a currently justified conclusion can be defeated, and "howgetit" operator, which determines how a currently defeated conclusion could be obtained are very interesting, and would appear to be very useful for facilitating and exploring the reasoning process, especially within decision support systems.

An additional interesting aspect of existing work is that there appears to be indecision within the defeasible reasoning and argumentation communities regarding the process of resolving conflict between rules or arguments. Some authors argue strongly that the use of priorities to resolve conflict should not be necessary. For example, Garcia and colleagues (Garcia and Simari, 2004) believe that common sense reasoning should be defeasible in a way that is not explicitly programmed. However, other authors, such as Maher and colleagues (Maher et al., 2001) and Antoniou and Bikakis (Antoniou and Bikakis, 2007) argue that knowledge in many domains is naturally represented using priority relations. Other authors, such as Pollock and Vreeswijk appear to take middle ground - offering support for numerical values, but not exclusively relying on them in the inference process. Finally, as we have discussed previously we believe that there may be a need for agent-based reasoning engines to support multiple formalisms and semantics to provide flexibility in the inference offered. However, the only author to discuss such support is Pollock. His use of reasoning schemas, written in a macro language, to tailor the reasoning process to the intended domain is very interesting, and is something we would like to investigate further.

CHAPTER 6

Current Work

6.1 Overview

This chapter provides an overview of the development of our prototype implementation of an argumentation-based reasoning component that is suitable for deployment within software agents. We begin by discussing our decision to implement the engine as a standalone component using an existing Prolog engine. We continue by discussing the initial algorithm we have chosen to incorporate within our engine and the technical implementation details of this work. The chapter also includes a discussion of our initial approach to the set-up of experimental evaluation, and concludes with an analysis of the work conducted so far.

6.2 Preparatory work - Introducing tuProlog

Our previous discussion of the challenges presented when implementing software agents lead us to analyse several existing agent development toolkits in which we could potentially implement our reasoning engine component. However, at the current stage of our research, where we are mainly interested in creating an architectural framework for the inference process, the overheads of producing an implementation compliant to a specific platform are too high and also potentially restrictive with regards to reuse of the component. Accordingly, we decided to implement the reasoning engine as a standalone component. Initial experiments will be conducted utilising the standalone reasoning component and only after satisfactory results have been obtained offline would we consider integrating our work into an existing agent framework.

The review of existing reasoning engines indicated that many such implementations were built upon an existing framework or implementation (typically Prolog) and accordingly the next stage of our work examined existing reasoning engine implementations that we could build upon, looking in particular for an agent or Internet-based engine that provided support for Prolog-style inference. Although we identified several defeasible reasoning engines in our existing discussion and also several standalone component-based implementations of Prolog, the most reliable open-source implementation with an active user-base was tuProlog (Denti et al., 2005). tuProlog is a Java-based light-weight Prolog implementation and has been designed from the ground up as a thin and lightweight engine that is easily deployable, dynamically configurable and easily integrated into Internet or agent applications (Denti et al., 2005). Its core is both minimal, taking the form of a small Java object containing only the most essential properties of a Prolog engine, and configurable, due to the ability to dynamically load and unload predicates, functors and operators which are embedded in libraries. tuProlog offers ISO compatible extensions to the standard 'Edinburgh' syntax (although this itself is configurable). The integration between Prolog and Java is as "wide, deep and clean as possible" (Denti et al., 2005), allowing not only Prolog to be called from

Java, but also Java objects to be instantiate and called from Prolog (which is very useful when creating prototype Prolog metainterpretters that require Java functionality, such as Network or File I/O).

There are a number of advantages to using tuProlog as a foundation for our engine. Firstly, the development of tuProlog itself followed the same design principles that we require for our intended domain of application, such as the minimality and flexibility of the core inference engine. Secondly, the engine is implemented in Java, which is becoming the de facto standard within agent technology and offers many benefits as discussed in Chapter 3, such as the support for complete serialization which allows the component to be easily transmitted across a network. Thirdly, we are building on top of a mature code-base so that much of the functionality that is common to both argumentation and Prolog-type inference can be relied on with a high-degree of confidence. Finally, the implementation is open-source which promotes the ability for us and other researchers to easily modify and verify the implementation.

6.3 Algorithms

Our discussion of challenges associated with argumentation clearly indicated that developing our own novel algorithm at the current stage of the research would not be productive (or possible), as designing such an algorithm is not a trivial task. Accordingly, the existing argumentation-based algorithms discussed previously within this report were reviewed. Ultimately we have attempted to implement the algorithm presented by Gerard Vreeswijk in the ASPIC D2.5 deliverable (Amgoud et al., 2005) which computes argument acceptability of the basis of the semantics of credulously preferred sets. It was determined that this algorithm was the most suitable for our requirements as it is query-based, which was identified as the most appropriate type for an agent reasoning within our intended domain of application, it was based upon the framework of Logic Programming, which is supported by our existing reasoning component tuProlog, and access to all components of the algorithm and an existing implementation (kindly supplied by Gerard Vreeswijk) were available. The existing prototype implementation of the algorithm, entitled simply Argument System (AS), was reviewed in Chapter 5 of this report.

As stated in the previous review of AS, the underlying algorithm supports both two numerical values to indicate the degree of belief (DOB) of a fact and also rule strength. Although we are not currently interested in implementing such numerical support we have utilised the values for DOB to indicate the strict or defeasible nature of a fact or rule within our implementation of the algorithm. To indicate facts or strict rules within our system we apply a DOB of 1.0 to the appropriate statement. To indicate presumptions or defeasible rules we apply a DOB of 0.8. We believe this is the easiest method of indicating defeasibility of statements (within the current technical framework provided by tuProlog, discussed next), and allows us to easily integrate numerical support in future releases of the engine if desired.

6.4 The implementation of our engine - 'Argue tuProlog'

As we are attempting to modify the existing functionality provided by tuProlog to support argumentation, we have decided to name our prototype implementation reasoning engine 'Argue tuProlog' (AtuP). As discussed previously, AtuP builds on the architecture foundations of the existing tuProlog codebase, and as such is implemented in Java and presented as a self-contained component that can be executed independently or integrated in a range of applications by utilising a well defined API we have provided. Figure 6.1 presents an overview of the Argue tuProlog application architecture. The API exposes key methods to allow an agent or Internet application developer to access and manipulate the knowledgebase, to construct rules, specify and execute queries (establishing whether a claim can be supported using the knowledgebase) and analyse results (determining the support for a claim and the acceptability of arguments).

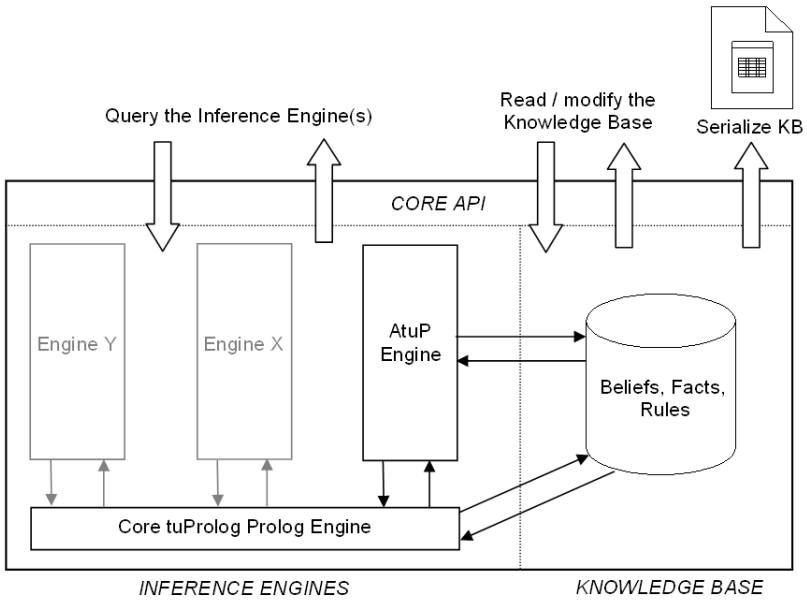

Figure 6.1: Overview of the core engine component architecture. All access to the inference engines and knowledge base is conducted through the core API.

As with the original AS implementation by Gerard Vreeswijk, AtuP accepts formulae in an extended first-order language and returns answers on the basis of the semantics of credulously preferred sets (as defined in (Amgoud et al., 2005)). The language of AtuP is constituted of atoms, terms and rules (see Section 2.7.3 in (Amgoud et al., 2005) for further details) and can be considered as a conservative extension of the basic language of Prolog, enriched with numbers that quantify degree of belief (which we have adapted to indicate defeasibility of statements). As AtuP is built on top of an existing Prolog engine, the engine naturally accepts Prolog programs.

There are two types of rules supported within the implementation, namely those with an empty antecedent (called beliefs) and those with a non-empty antecedent (called rules). Every expression of the form t b. is a rule where t is a term and b indicates

a degree of belief, with 1.0 indicating a strict term and 0.8 a defeasible term. Examples of beliefs include `a 0.8.` `flies(sylvester) 0.8.` Examples of facts are texttt-tflies(sylvester).`raining(London).` Every expression of the form t :- $t_1, ..., t_n$ b. is a rule, provided t, t_i are terms, n ¿ 0, and b denotes the DOB indicating the defeasibility of the rule. Examples of rules include `flies(X) :- bird(X) 0.8.` `a :- c,d.`. A query is an expression of the form ?- $t_1, ...t_n$. where n ¿ 0. It is possible to include more queries in the input, but since we are usually only interested in one goal proposition, this is not typical. For the same reasons, we almost always have that $n = 1$. In the case that $n > 1$, AtuP (like AS) attempts to construct n arguments for t_1,...,t_n that are defendable simultaneously. Internally, this is managed by introducing an additional rule *main* :- t_1,...,t_n after which AtuP tries to find arguments for *main*.

If ?- t. is a query then AtuP's main goal is to try and find an argument with conclusion t and then try to construct an admissible set around that argument. In AtuP every search for arguments for a particular query is encapsulated within another instance of an engine (this engine of course has access to the original engine's configuration and knowledgebase). Using multiple engines allows us to keep track of which participant (PRO or OPP) is conducting the current query an also to pause the "dialogue" at any time for further analysis. Once the first argument, say A, is found, the first engine is suspended and A is returned to AtuP. AtuP then tries to find an attacker of A. Thus for every sub-conclusion s of A, a separate engine is instantiated to search for arguments against s. If one of these remains undefeated (which is defined within (Amgoud et al., 2005)), then A is defeated. Else A remains undefeated.

6.5 Obtaining results

When AtuP has finished determining the support for a claim and the acceptability of associated arguments the engine generates a trace of the argument game dialogue (shown in Figure 6.3), the contents of the admissible set and a simplified graph showing the attack relations between arguments. In addition to providing an API to allow agent developers to utilise our engine to perform these actions, we have also modified the existing tuProlog graphical user interface to facilitate off-line experimentation with the engine (as shown in Figure 6.2).

6.6 Experimental work

Although we have tested our system using several of the benchmark argumentation problems documented within the literature we have currently not conducted a thorough empirical evaluation of our engine, and in fact developing an appropriate experimental methodology is one of the main focuses of our future work. However, we have developed the core engine using Sun Microsystem's NetBeans integrated development environment (Microsystems, 2006b) in which we have installed the latest version of NetBeans Profiler (Microsystems, 2006a), a fully functional application profiling tool. This will allow us to simulate deployment of our engine within a variety of realistic scenarios, and to monitor and analyse such data as the consumption of CPU cycles, memory usage, thread profiling and basic Java Virtual Machine (JVM) behaviour. In addition the profiler can be used to break down

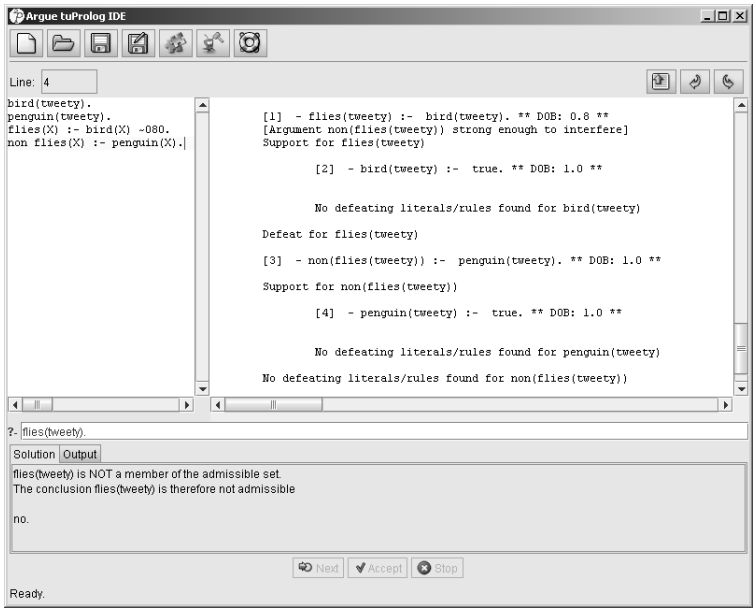

Figure 6.2: Screenshot of "Argue tuProlog" GUI. The left window allows manipulation of the knowledge base. The bottom window allows query entry and displays the results.

the execution of the complete implementation, identifying potential resource intensive operations or problematic data structures (as was common with existing reasoning engine implementations). When we have developed an appropriate evaluation methodology and set up several large-scale knowledge bases the profiling tool will facilitate our ultimate goal of obtaining empirical evaluations of our engine implementation.

6.7 Evaluation

Our initial prototype is a first step in creating an argumentation-based reasoning component. Developing the framework for a flexible agent-based reasoning component has been challenging, and as we expected earlier in our work, this first stage of research has not involved much work on the associated complexity or efficiency issues which will be necessary for final deployment. We have been able to examine performance of the engine utilising the NetBeans profiler, and through simple testing we have already identified several areas of both our code and existing tuProlog code that require improvement to produce an efficient reasoning component. However, during the recent Computational Models of Argument (COMMA) 2006 conference it was identified that our implementation of the algorithm presented in the ASPIC deliverable D2.5 was not completely correct (partly due to our misunderstanding of the algorithm and partly due to the difficulty in representing some parts of the algorithm correctly in the existing tuProlog reasoning architecture), and as a result several counter-intuitive results can be obtained when reasoning. We have determined that correcting these fundamental errors with our implementation of the algorithm would require a fairly major modification to our code that was integrated within the existing tuProlog code. When attempting to correct these issues we learned that a new version

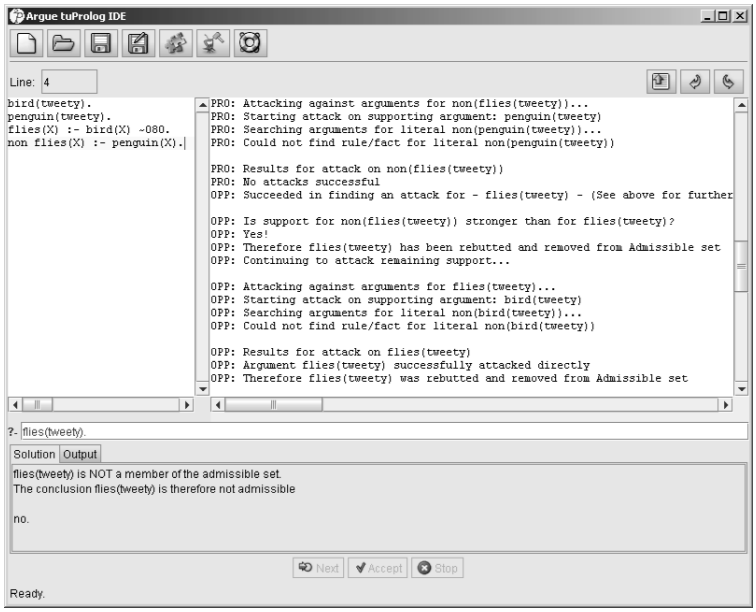

Figure 6.3: Screenshot of "Argue tuProlog" GUI. In this screenshot the right window shows an argument game trace after a query has been executed.

of tuProlog was being released by its creators that promised a more flexible and open inference module to support the style of modification we are conducting. Accordingly, we have decided to stop implementation work until the source code of a stable release of this new version is available (scheduled for Easter 2007), and instead focus on theoretical issues for future work and also completing this report. Although we will most likely discard our existing implementation work on the first version of the engine, the process of implementing the ASPIC D2.5 algorithm has been an excellent learning experience, and will surely help us to create a correct and efficient second version within the new tuProlog framework.

CHAPTER 7

Future Work

7.1 Overview

This chapter of the report discusses our proposed future work for both the agent-based reasoning component and associated tools, and also a potential application for this research. Our research has indicated that the limited number of existing implementations of argumentation within agent technology have been deployed into a variety of domains, such as business process modelling and the implementation of stock market trading strategies. We believe the combination of software agents and argumentation will also be beneficial to automated recommendation systems, particularly in the domain of consumer reviews. To our knowledge there is only one such piece of work that incorporates argumentation related to this topic, which is research conducted by Chesnevar and colleagues (Chesnevar et al., 2006b) using the DeLP reasoning engine implementation (Garcia and Simari, 2004) with the intended domain of application being the efficient searching and recommendation of newspaper articles relevant to user's preference. However, the deployment platform in their work was not agent-based and no large-scale experimental evaluation of the implementation, which we consider as essential, was conducted.

Although we have identified a potential domain of application for our research, our primary focus for future work lies in the essential framework needed to exploit this opportunity, i.e. the completion of the new version of the Argue tuProlog engine implementation and associated experimental platform. We are currently waiting for a stable release of the tuProlog code (on which we have based our engine) to conduct this work, and therefore in the meantime we will explore two additional strands of research to support our work. The first is the automatic extraction of simple arguments, perhaps from consumer reviews, and the second is the use of numerical values within our argumentation framework, a popular way in which humans assign support to statements and use to resolve conflicts between information.

7.2 Adding to the new AtuP

Currently a beta version of the new tuProlog engine has been released which provides a more flexible and open reasoning architecture, and should facilitate our efforts at integrating argumentation-based inference. However, there are currently several outstanding critical bugs with this implementation. The authors of the engine have stated that a stable release of the implementation and associated source code will be released at Easter 2007. As soon as we have access to the new code base we intend to re-implement correctly the existing algorithm presented by Gerard Vreeswijk in the ASPIC D2.5 deliverable (Amgoud et al., 2005). In addition, we intend to add support for more algorithms within our engine and allow an agent utilising the reasoning engine to choose an appropriate formalism at runtime.

Our initial ideas to approach this work included attempting to define a generic or parametric algorithm implementation that could be configured to perform reasoning under a number of different formalisms and algorithms. However, discussion at the recent COMMA conference established that this implementation of this may be too complex, and attempting to define a generic framework to support the multitude of existing argumentation formalisms may result in creating such a fine-grained framework that any benefits would be negated. Clearly further investigation into this topic is required.

We are also very interested with incorporating techniques to improve the efficiency of the inference process that we have identified as appropriate for agent-based reasoning, such as argument tree-pruning and approximating arguments, and believe that the architecture provided by tuProlog is highly suitable for this.

7.3 Experimentation platform

Our previous discussion indicated that although empirical evaluations are beginning to be conducted on defeasible reasoning implementations, work is needed to provide a comprehensive strategy to ensure the testing platforms are comparable. This will be a topic for investigation in our future work. Our first proposed approach consists of implementing several algorithms (discussed above) within our engine and benchmarking the system using a large data set and the NetBeans application profiler, which would naturally allow a fair comparison of algorithmic and implementation performance. An alternative view to benchmarking would be to utilise empirical complexity analysis in order to evaluate the various algorithm implementations. As discussed by Nudelman (Nudelman, 2005), many researchers evaluating algorithm implementations provide only formal complexity analysis to the hard decision problems. He continues to argue that this view causes most work to focus on the aggregate picture of complexity, i.e. complexity of a problem as a whole, and not of particular instances (Nudelman, 2005). In the most classical case this aggregation takes form of the worst case complexity (i.e the max operator). A slightly more informative viewpoint is obtained by adding to the mix some (usually uniform) probability distribution over problem instance or some randomness to the algorithms. This leads to the notion of average-case complexity - still an aggregation, with max replaced by the expectation. In his recent PhD thesis (Nudelman, 2005) Nudelman presents a complementary view of complexity that overcomes some of the shortcomings of formal complexity analysis. He proposes a new approach for understanding the empirical complexity of NP-hard problems. Machine learning techniques are then used to build regression models that predict an algorithm's runtime given a previously unseen problem instance. Nudelman also discusses techniques for interpreting these models to identify where to focus within an algorithm to improve performance and to gain understanding of the characteristics that cause instance to be hard or easy. This would be ideal for our intended domain of application. Not only could we use Nudelman's technique to benchmark our implementation of algorithms and identify modifications to improve efficiency, but we would also have access to very useful information regarding the runtime requirements of an algorithm. For example, the machine learning techniques may allow an agent to efficiently "pre-process" an unseen problem instance and reliably predict resources and time required to complete the reasoning. This would be extremely useful in a resource-bound environment.

7.4 Extracting arguments

An open problem within the argumentation community is the extraction of arguments from unstructured text. We would like to investigate this problem and potentially generate simplistic arguments using movie reviews as our source of text. We believe that arguments presented within such reviews could be used to augment traditional approaches to motivating recommendations to users in a similar way to work conducted by Mozina et al. (2006) on argument based rule learning applied to the domains of law and medicine. Although our initial analysis of this research problem is limited, we have begun by examining work from three authors. The first work is presented by Sergio Alvarado (Alvardo, 1990) in which a theory of how to understand and extract what he refers to as units of argumentation units from unstructured newspaper editorial texts. Secondly, we have began to examine work by Simone Teufel and colleagues (Teufel and Moens, 1999) which discusses the use of a machine learning based annotation scheme to identify and model prototypical academic arguments within unstructured text. Finally, we are interested in the machine learning techniques which have been applied by Bo Pang and colleagues (Pang et al., 2002) to perform sentiment analysis on text.

We believe a combination of these techniques may allow us to develop a machine learning algorithm that can not only determine the sentiment of a movie review, but also extract simple arguments to support our conclusion. For example, a foreign art house movie review containing the word "unwatchable" several time would clearly indicate a negative sentiment from the review author, but if we could learn that this conclusion was frequently supported by the argument "in my opinion, any film that has subtitles is typically unwatchable" we have obtained much more information about the bias of the review author. In addition the arguments identified could be used to augment the recommendation process. For example, an argumentation based system may allow users to express complicated preferences relatively easy. For example, a typical preference rule may be "Typically I do not like movies with non-linear story lines, except when they are directed by Tarantino". This is simply a defeasible rule with an exception which could easily be incorporated into the argumentation process. As a first step towards machien learning techniques for argument extraction we are currently investigating the use of text parsing software to explore whether this technique can provide us with a method of automatically identifying argument structure within text.

7.5 Dealing with Uncertainty

Although this report has focused on one particular kind of inference under uncertainty, many different formalisms for treating such problems have been developed so far. In addition to the symbolic approaches based on nonmonotonic logic we have so far discussed in this report, there are also numerical methods. The most popular numerical approaches are the theory of Bayesian networks, the Dempster-Shafer theory of evidence and possibility theory (Haenni, 2001). Although these are outside the scope of the current work it may be useful to support some kind of formal numerical reasoning in future versions of the engine as knowledge from certain domains is naturally expressed with such values. Future work is planned to evaluate the feasiblility of enhancing arguments with possibilities as discussed in, for example, Krause et al. (1995), Amgoud and Prade (2004b) or Chesnevar

et al. (2004). If possible this would provide us with a computationally efficient model of argumentation with a well-founded numerical semantics.

CHAPTER 8

Final Conclusions

In this report we have discussed our current work on a prototype light-weight argumentation reasoning engine for software agents (Bryant et al., 2006; Bryant and Krause, 2006). The core engine has been built using tuProlog (Denti et al., 2005, 2001), an existing open-source Prolog engine, as its foundation, which followed the same design principles that we require for our intended domain of application. Although our goal is to create a general purpose argumentation engine that can be configured to conform to one of a range of semantics, the current version of the engine attempts to implements the argumentation-based framework presented in (Amgoud et al., 2005), allowing our engine to generate arguments and counter arguments over an inconsistent knowledge base, determine the acceptability of arguments and construct proofs using an argument game approach to constructing proofs of acceptance (Jakobovits and Vermeir, 1999a). The use of an existing Prolog engine also allows our implementation to support standard Prolog inference which will enable us to prototype a variety of meta-interpreters that support other forms of argumentation, with the ultimate goal of integrating these algorithms within the core engine

This report has also explored how we might approach the creation of a toolkit to support empirical experimentation with argumentation-based reasoning implementations. This was achieved by discussing the inherent challenges presented by the use of software agents and argumentation technology, reviewing existing related work and also exploring the technical details of several existing nonmonotonic reasoning engines. Future work will focus on issues such as the improvement of the algorithm implementation within our current prototype, how we may implement an experimental testing strategy for argumentation systems, and also work on extracting simplistic arguments from unstructured text and adding support for numerical values to the argumentation formalism in order to facilitate what we believe is a potential application for our research, recommender systems.

REFERENCES

Alferes, J. J. and L. M. Pereira (1996). *Reasoning with Logic Programming*. Springer-Verlag New York, Inc.

Alvardo, S. J. (1990). *Understanding Editorial Text: A Computer Model of Argument Comprehension*. Kluwer Academic Publishers.

Amgoud, L., M. Caminada, S. Doutre, H. Prakken, and G. Vreeswijk (2005). Final Review and Report on Argumentation System. Technical Report ASPIC Deliverable 2.5.

Amgoud, L., C. Cayrol, and M.-C. Lagasquie-Schiex (2004). On the bipolarity in argumentation frameworks. In Delgrande, J. P. and T. Schaub (eds.) *NMR*, pp. 1–9. ISBN 92-990021-0-X.

Amgoud, L. and H. Prade (2004a). Using arguments for making decisions: a possibilistic logic approach. In *AUAI '04: Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pp. 10–17. AUAI Press, Arlington, Virginia, United States.

Amgoud, L. and H. Prade (2004b). Using arguments for making decisions: a possibilistic logic approach. In *AUAI '04: Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pp. 10–17. AUAI Press, Arlington, Virginia, United States. ISBN 0-9749039-0-6.

Antoniou, G. (1997). *Nonmonotonic Reasoning*. MIT Press.

Antoniou, G. and A. Bikakis (2007). DR-Prolog: A System for Defeasible Reasoning with Rules and Ontologies on the Semantic Web. *Knowledge and Data Engineering, IEEE Transactions on*, **19**(2), pp. 233–245.

Antoniou, G., D. Billington, G. Governatori, and M. J. Maher (2001). Representation results for defeasible logic. *ACM Transactions on Computational Logic*, **2**(2), pp. 255–287.

Bellifemine, F., A. Poggi, and G. Rimassa (2001). JADE a FIPA2000 compliant agent development environment. *Proceedings of the International Conference on Autonomous Agents*, pp. 216 – 217.

Bench-Capon, T. J. M. (2003). Persuasion in Practical Argument Using Value-based Argumentation Frameworks. *J Logic Computation*, **13**(3), pp. 429–448.

Besnard, P. and A. Hunter (2006). *Elements of Argumentation*. In preparation.

Billington, D. and A. Rock (2001). Propositional Plausible Logic: Introduction and Implementation. *Studia Logica*, **67**(2), pp. 243–269.

Boolos, G., J. Burgess, and R. Jeffrey (2002). *Computability and Logic*. Cambridge University Press.

Bryant, D. and P. J. Krause (2006). An Implementation of a Lightweight Argumentation Engine for Agent Applications. In *Proceedings of 10th European Conference on Logics in Artificial Intelligence (JELIA06)*, volume 4160 of *LNAI*, pp. 469–472. Springer.

Bryant, D., P. J. Krause, and G. Vreeswijk (2006). Argue tuProlog: A Lightweight Argumentation Engine for Agent Applications. In *Proceedings of 1st International Conference on Computational Models of Argument (COMMA06)*, pp. 27–32. IOS Press.

Bryson, J., D. Martin, S. A. McIlraith, and L. A. Stein (2003). Agent-Based Composite Services in DAML-S: The Behavior-Oriented Design of an Intelligent Semantic Web. In Zhong, N., J. Liu, and Y. Yao (eds.) *Web Intelligence*. Springer.

Cadoli, M. and M. Schaerf (1993). A Survey of Complexity Results for Nonmonotonic Logics. *Journal of Logic Programming*, **17**(2/3), pp. 127–160.

Capobianco, M., C. Chesnevar, and G. Simari (2004). An argument-based framework to model an agent's beliefs in a dynamic environment. In *Proc. of the First International Workshop on Argumentation in Multiagent Systems. AAMAS 2004*.

Capobianco, M. and C. I. Chesnevar (1999). Introducing Dialectical Bases in Defeasible Argumentation. In *Proceedings of the 6th Workshop on Aspectos Teoricos de la Inteligencia Artificial (ATIA)*, pp. 1–10. San Juan, Argentina.

Capobianco, M., C. I. Chesnevar, and G. R. Simari (2005). Argumentation and the Dynamics of Warranted Beliefs in Changing Environments. *Autonomous Agents and Multi-Agent Systems*, **11**(2), pp. 127–151.

Causey, R. L. (1994). EVID: A system for interactive defeasible reasoning. *Decision Support Sytems*, **11**(2), pp. 103–131.

Causey, R. L. (2003). Computational dialogic defeasible reasoning. *Argumentation*, **17**(4), pp. 421–450.

Cayrol, C., S. Doutre, and J. Mengin (2003). On Decision Problems Related to the Preferred Semantics for Argumentation Frameworks. *J Logic Computation*, **13**(3), pp. 377–403.

Cecchi, L. A., P. R. Fillottrani, and G. R. Simari (2006). On the Complexity of DeLP through Game Semantics. In *Proc. 11th Intl. Workshop on Nonmonotonic Reasoning (NMR 2006). J. Dix and A. Hunter (Eds.)*, pp. 386–384. IfI Technical Report Series, Clausthal University.

Chesnevar, C., J. McGinnis, S. Modgil, I. Rahwan, C. Reed, G.Simari, M. South, G. Vreeswijk, and S. Willmott (2006a). Towards an argument interchange format. *The Knowledge Engineering Review*, **21**, pp. 293–316.

Chesnevar, C. I. and A. G. Maguitman (2004). ArgueNet: an argument-based recommender system for solving Web search queries. In *Proceedings. 2004 2nd International IEEE Conference in Intelligent Systems*, pp. 282–287.

Chesnevar, C. I., A. G. Maguitman, and R. P. Loui (2000a). Logical models of argument. *ACM Comput. Surv.*, **32**(4), pp. 337–383. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/371578.371581.

Chesnevar, C. I., A. G. Maguitman, and G. R. Simari (2006b). Argument-based critics and recommenders: A qualitative perspective on user support systems. *Data and Knowledge Engineering*, **59**(2), pp. 293–316.

Chesnevar, C. I., G. R. Simari, T. Alsinet, and L. Godo (2004). A logic programming framework for possibilistic argumentation with vague knowledge. In *AUAI '04: Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pp. 76–84. AUAI Press, Arlington, Virginia, United States. ISBN 0-9749039-0-6.

Chesnevar, C. I., G. R. Simari, and A. J. Garca (2000b). Pruning Search Space in Defeasible Argumentation. In *Proc. of Workshop on Advances and Trends in Search in Artificial Intelligence*, pp. 40–47.

Chisholm, R. (1997). *Theory of Knowledge*. Prentice-Hall, New Jersey.

Cholewinski, P., V. W. Marek, A. Mikitiuk, and M. Truszczyński (1999). Computing with default logic. *Artif. Intell.*, **112**(1-2), pp. 105–146. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/S0004-3702(99)00053-3.

Colmerauer, A., H. Kanoui, P. Roussel, and R. Pasero (1973). Un systeme de communication homme-machine en francais. Technical report, Groupe de Recherche en Intelligence Artificielle, Universite d'Aix-Marseille II.

Covington, M. A. (2000). Logical control of an elavator with defeasible logic. *IEEE Transactions on Automatic Control*, **45**(7), pp. 1347–1349.

Denti, E., A. Omicini, and A. Ricci (2001). tuProlog: A Light-Weight Prolog for Internet Applications and Infrastructures. In *PADL*, pp. 184–198.

Denti, E., A. Omicini, and A. Ricci (2005). Multi-paradigm Java-Prolog integration in tuProlog. *Sci. Comput. Program.*, **57**(2), pp. 217–250. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2005.02.001.

Dimopoulos, Y., B. Nebel, and F. Toni (2000). Finding Admissible and Preferred Arguments Can Be Very Hard. In Cohn, A. G., F. Giunchiglia, and B. Selman (eds.) *KR2000: Principles of Knowledge Representation and Reasoning*, pp. 53–61. Morgan Kaufmann, San Francisco.

Dimopoulos, Y., B. Nebel, and F. Toni (2002). On the computational complexity of assumption-based argumentation for default reasoning. *Artif. Intell.*, **141**(1), pp. 57–78. ISSN 0004-3702. doi:http://dx.doi.org/10.1016/S0004-3702(02)00245-X.

Dimopoulos, Y. and A. Torres (1996). Graph theoretical structures in logic programs and default theories. *Theor. Comput. Sci.*, **170**(1-2), pp. 209–244. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/S0304-3975(96)00004-7.

Dowling, W. F. and J. H. Gallier (1984). Linear-time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *J. Log. Program.*, **1**, pp. 267–284. doi: http://dx.doi.org/10.1016/0743-1066(89)90009-5.

Dung, P. M. (1995). On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and n-Person Games. *Artificial Intelligence*, **77**(2), pp. 321–358.

Dung, P. M., R. A. Kowalski, and F. Toni (2006). Dialectic proof procedures for assumption based admissible argumentation frameworks. *Artificial Intelligence*, **170**(2), pp. 114–159.

Dunne, P. E. and T. J. M. Bench-Capon (2002). Coherence in finite argument systems. *Artif. Intell.*, **141**(1), pp. 187–203.

Franklin, S. and A. Graesser (1997). Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents. In *ECAI '96: Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages*, pp. 21–35. Springer-Verlag, London, UK. ISBN 3-540-62507-0.

Garcia, A., D. Gollapally, P. Tarau, and G. Simari (2000). Deliberative stock market agents using Jinni and defeasible logic programming. In *In Proc. of the ECAI Workshop on Engineering Societies in the Agents*. Springer Verlag.

Garcia, A. and G. R. Simari (1999). Parallel defeasible argumentation. *Journal of computer science and technology special issue: Artificial intelligence and evolutive computation.*, **1**(2), pp. 45–57.

Garcia, A. J. and G. R. Simari (2004). Defeasible logic programming: an argumentative approach. *Theory Pract. Log. Program.*, **4**(2), pp. 95–138. ISSN 1471-0684. doi: http://dx.doi.org/10.1017/S1471068403001674.

Garey, M. and D. Johnson (1979). *Computers and Intractability*. W H Freeman.

Gelder, A. V., K. A. Ross, and J. Schlipf (1991). The well-founded semantics for general logic programs. *Journal of the ACM*, **38**(3), pp. 620–650.

Gelfond, M. and V. Lifschitz (1991). Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, **9**(3/4), pp. 365–386.

Girle, R., D. Hitchcock, P. McBurney, and B. Verheij (2004). Decision Support for Practical Reasoning. In C. Reed, C. and T. J. Norman (eds.) *Argumentation Machines*, pp. 56–83. Kluwer Academic Publishers, The Netherlands.

Haenni, R. (2001). Cost-bounded argumentation. *International Journal of Approximate Reasoning*, **26**, pp. 101–127(27).

Jakobovits, H. and D. Vermeir (1999a). Dialectic semantics for argumentation frameworks. In *International Conference on Artificial Intelligence and Law*, pp. 53–62.

Jakobovits, H. and D. Vermeir (1999b). Dialectic semantics for argumentation frameworks. In *Proceedings of the 7th Int. Conf. on Artificial Intelligence and Law*, pp. 53–65. ACM Press.

Kotz, D. and R. S. Gray (1999). Mobile Agents and the Future of the Internet. *ACM Operating Systems Review*, **33**(3), pp. 7–13.

Koukoumpetsos, K. and N. Antonopoulos (2002). Mobility Patterns: An Alternative Approach to Mobility Management. In *Proceedings of the 6th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI2002)*. Orlando, USA.

Kowalski, R. (1979). Algorithm = logic + control. *Communications of the ACM*, **22**, pp. 424–436.

Krause, P., S. Ambler, M. Elvang-Goransson, and J. Fox (1995). A logic of argumentation for reasoning under uncertainty. *Computational Intelligence*, **11**, pp. 113–131.

Loui, R. and G. Simari (1994). *NATHAN (Spec13): Argues defeasibly in first-order logic.* http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/reasonng/defeasbl/nathan/0.html. Last accessed: 13th February 2007.

Loui, R. P., J. Norman, J. Olson, and A. Merrill (1993). A design for reasoning with policies, precedents, and rationales. In *ICAIL '93: Proceedings of the 4th international conference on Artificial intelligence and law*, pp. 202–211. ACM Press, New York, NY, USA.

Loyd, J. (1984). *Foundations of Logic Programming*. Springer-Verlag.

Ltd., C. J. (2006). *Jackdaw: A multi-agent platform for mobile services.* http://www.calicojack.co.uk/flyer-jackdaw-v1-0.pdf. Last accessed: 21st February 2007.

Maher, M. J., A. Rock, G. Antoniou, D. Billington, and T. Miller (2001). Efficient Defeasible Reasoning Systems. *International Journal on Artificial Intelligence Tools*, **10**(4), pp. 483–501.

Microsystems, S. (2006a). *NetBeans Profiler*. available from http://profiler.netbeans.org/. Last accessed: 10 May 2006.

Microsystems, S. (2006b). *Welcome to NetBeans.* available from http://www.netbeans.org/. Last accessed: 10 May 2006.

Mozina, M., J. Zabkar, and I. Bratko (2006). Argument Based Rule Learning. In Brewka, G., S. Coradeschi, A. Perini, and P. Traverso (eds.) *ECAI*, pp. 504–508. IOS Press. ISBN 1-58603-642-4.

Nudelman, E. (2005). *Empirical Approaches to the Complexity of Hard Problems.* Ph.D. thesis, Stanford University, Stanford, CA.

Nute, D. (1988). Defeasible reasoning and decision support systems. *Decis. Support Syst.*, **4**(1), pp. 97–110.

Nute, D. (1993). Defeasible Prolog. In *Proceedings of AAAI Fall Symposium on Automated Deduction in Nonstandard Logics, (Technical Report FS-93-01)*, pp. 105–112.

Nute, D. (1994). Defeasible Logic. In Gabbay, D., C. J. Hogger, and J. A. Robinson (eds.) *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 3: Nonmonotonic Reasoning and Uncertain Reasoning*, pp. 353–395. Oxford University Press, Oxford.

Nute, D., R. I. Mann, and B. F. Brewer (1990). Controlling expert system recommendations with defeasible logic. *Decision Support Systems*, **6**(2), pp. 153–164.

Ogden, C. K. and I. A. Richards (1956). *The Meaning of Meaning.* Routledge and Kegan Paul Ltd, 10th edition.

OUP (2007). *Oxford English Dictionary.* http://www.oed.com. Last accessed: 19th November 2006.

Pang, B., L. Lee, and S. Vaithyanathan (2002). Thumbs up? Sentiment Classification using Machine Learning Techniques. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing (EMNLP).*

Papazoglou, M. P. (2003). Service -Oriented Computing: Concepts, Characteristics and Directions. In *WISE '03: Proceedings of the Fourth International Conference on Web Information Systems Engineering*, p. 3. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-1999-7.

Parsons, S., C. Sierra, and N. Jennings (1998). Agents that Reason and Negotiate by Arguing. *Journal of Logic and Computation*, **8**(3), pp. 261–292.

Pollock, J. L. (1992). How to reason defeasibly. *Artif. Intell.*, **57**(1), pp. 1–42. ISSN 0004-3702. doi:http://dx.doi.org/10.1016/0004-3702(92)90103-5.

Pollock, J. L. (1995). *Cognitive Carpentry: A Blueprint for how to build a person.* MIT Press, A Bradford Book, USA.

Prakken, H. and G. Vresswijk (2002). Logics for Defeasible Argumentation. In Gabbay, D. and F. Guenthner (eds.) *Handbook of Philosophical Logic (Second Edition)*, pp. 218–319. Kluwer Academic Publishers, The Netherlands.

Reed, C. and G. Rowe (2004). Araucaria: Software for Argument Analysis, Diagramming and Representation. *International Journal on Artificial Intelligence Tools*, **13**(4), pp. 983–.

Rock, A. (2006a). *Deimos: A Query Answering Defeasible Logic System.* http://www.cit.gu.edu.au/ arock/defeasible/doc/Deimos-short.pdf. Last accessed: 19th November 2006.

Rock, A. (2006b). *Phobos (Version 2): A Query Answering Plausible Logic System.* http://www.cit.gu.edu.au/ arock/plausible/doc/Phobos-short.pdf. Last accessed: 19th November 2006.

Rock, A. and D. Billington (2000). An implementation of propositional plausible logic. In *Proceedings of 23rd Australasian Computer Science Conference, 2000. ACSC 2000.*, pp. 204–210. IEEE Press.

Salter, J. and N. Antonopoulos (2006). CinemaScreen Recommender Agent: Combining Collaborative and Content-Based Filtering. *IEEE Intelligent Systems*, **21**(1), pp. 35–41. ISSN 1541-1672. doi:http://doi.ieeecomputersociety.org/10.1109/MIS.2006.4.

Schroeder, M. (1999). An Efficient Argumentation Framework for Negotiating Autonomous Agents. In *MAAMAW '99: Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, pp. 140–149. Springer-Verlag, London, UK.

Simari, G. and R. Loui (1982). Mathematical treatment of defeasible reasoning and its implementation. *Artificial Intelligence*, **53**(2-3), pp. 125–157.

Tarau, P. (1999). Jinni: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. In *In Proc. of the Fourth INternational Conference on the Practical Application of Intelligent Agnets and Multi-Agents.*, pp. 109–123. London, UK.

Teufel, S. and M. Moens (1999). Discourse-Level Argumentation in Scientific Articles: Human and Automatic Annotation. In Walker, M. (ed.) *Towards Standards and Tools for Discourse Tagging: Proceedings of the Workshop*, pp. 84–93. Association for Computational Linguistics, Somerset, New Jersey.

Tolksdorf, R. (2007). *Languages for the Java VM.* http://www.robert-tolksdorf.de/vmlanguages.html. Last accessed: 13th February 2007.

Tsvetovatyy, M., M. Gini, B. Mobasher, and Z. W. Ski (1997). Magma: An agent based virtual markey for electronic commerce. *Applied Artificial Intelligence*, **11**, pp. 501–523.

Vreeswijk, G. (1993a). *IACAS: an interactive argumentation system - User manual version 1.0.* citeseer.ist.psu.edu/195813.html. Last accessed: 19th November 2006.

Vreeswijk, G. (1993b). *Studies in Defeasible Argumentation.* Ph.D. thesis, Free University of Amsterdam, The Netherlands.

Vreeswijk, G. A. W. (2006). An algorithm to compute minimaly grounded and admissible defence sets. In *Proceedings of 1st International Conference on Computational Models of Argument (COMMA06)*, pp. 109–120. IOS Press.

Wooldridge, M. (2000). *Rational Agents.* The MIT Press, London.

Wooldridge, M. (2002). *An Introduction to MultiAgent Systems.* John Wiley and Sons Ltd.