The Knowledge Engineering Review, Vol. 00:0, 1–24. © 2004, Cambridge University Press
DOI: 10.1017/S000000000000000 Printed in the United Kingdom

A review of current defeasible reasoning implementations [1]

Daniel Bryant and Paul Krause

Dept. of Computing, University of Surrey, Guildford, Surrey. GU2 7XH UK
E-mail: {d.bryant, p.krause}@surrey.ac.uk

Abstract

This article surveys existing practical implementations of both defeasible and argumentation-based reasoning engines and associated literature. We aim to summarise the current state of the art in the research area, show that there are many similiarities and connections between the various implementations and also highlight the differences regarding evaluation goals and strategies. An important goal of this paper is to argue for the need for well designed empirical evaluations, as well as formal complexity analysis, in order to justify the practical applicability of a reasoning engine. There are indeed many challenges to be faced in developing implementations of argumentation. Not least of these is the inherent computational complexity of the formal models. We cover some of the ways these challenges have been addressed, and provide pointers for future directions in realizing the goal of practical argumentation.

[1]This work was partially supported by an EPSRC PhD Studentship.

1 Introduction

In everyday life human decision-making is often based on arguments and counter-arguments. Decisions made in this way have a basis that can be easily referred to for explanation purposes as 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. There have been many attempts over the years to automate human decision making which have typically utilised rules within a knowledge-based system. An early example and probably still the most well known of which is MYCIN [69]. However, reasoning in the real world is not easily achieved using such rules as frequently information is incomplete, incoherent or contradictory. It has therefore been proposed that automated decision making systems may benefit from the use of 'defeasible argumentation' [62], 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. When compared with traditional automated rule-based decision-making this approach is more in keeping with the way humans often deliberate and finally make a choice [2] and could prove extremely useful in real world software applications.

The formal foundations of argumentation have been well explored in the academic literature (for surveys of the field see [62], [22] and [7]). In contrast the actual implemented computational models have been less widely explored. There has been limited attempts to create a practical implementation of such models in, for example, a negotiation support application [68], collaborative decision making (using medical transplants as a case study) [73], and a proactive tool for expressing stock market trading strategies [38]. However, the reasoning component is normally tightly integrated within the software and cannot easily be reused within additional applications. In our recent work [12, 11] we have argued that for real world software applications to be assisted by defeasible reasoning, access to a general purpose non-monotonic reasoning component (an "argumentation engine") is required. 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 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 the time is right for such an application to be constructed in an appropriate (and efficient) language and should also be open for modification. This would allow (and in fact encourage) 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, [36], [30]) which may not reveal the real world performance of the algorithms [77]. The proprietary implementations of algorithms as mentioned previously has also not helped this situation, making the establishment of a common testing platform difficult. In addition, although several of the implementations include tools to automatically generate test cases, there can sometimes be suspicions that these tools have been engineered to produce test cases that best suite the authors implementation.

This article examines existing implementations of both defeasible and argumentation-based reasoning engines and associated literature. This review focuses on the technical merits of the implementation, and as such, references are given to allow the interested reader to explore the underlying theoretical details. We have examined only systems that rely on symbolic approaches

to dealing with inference under uncertainty, as it is our belief that the use of numerical values can alter the underlying logical machinery and implementation details significantly, and as such deserves a separate review. Several of the systems reviewed do offer numerical support, but this is not their primary focus, and as such discussion of these features is minimal. We include a selection of different inference styles (bottom-up and top-down), algorithm types (query answering and total) and implementation technologies. We then use this selection to summarise the current state of the art in the research area, show that there are many similiarities and connections between the various implementations and also highlight the differences regarding evaluation goals and strategies. We hope that this review will motivate future research to overcome the existing problems with practical implementation and encourage further empirical experimentation. We strongly believe that the use of this type of evaluation can greatly assist in determining the real-world value offered by argumentation-based reasoning.

The paper begins with an overview of argumentation which briefly discusses the origin of this type of reasoning, summarises the theoretical background and explains the contemporary ideas regarding reasoning algorithms. The challenges of practically implementing argumentation-based reasoning are then discussed in order to motivate the approach we have taken to this survey. The paper continues with a review of 15 defeasible and argumentation-based practical implementations, and concludes with a discussion that summarises key points and attempts to provide a roadmap for future work within this research topic.

2 An Overview of Argumentation-based Reasoning

2.1 The rise of nonmonotonic logics

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 [22]. Since then solid theoretical foundations of nonmonotonic logics have been established (the interested reader is referred to [3] for further information). Although this article is not focused on nonmonotonic logics specifically, it is still useful to give a brief overview of the topic as these logics could be considered the precursor of defeasible argumentation and as such share many characteristics.

Several styles of nonmonotonic logics exist. Most of them take 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' [62]. Defaults do not guarantee that their consequent holds whenever their antecedent holds, instead they allow us in such cases to defeasibly derive their consequent. That is, if nothing is known about exceptional circumstances. Most nonmonotonic 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: autoepistemic logic, circumscription, default logic, logic programming with negation as failure, defeasible logic, multi-valued logics and plausible logic. Several of these systems will be briefly reviewed later in the implementation review section, and the interested reader is referred to [3, 62] for a comprehensive overview.

2.2 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 next two sections of the article 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 survey paper [62]. It should be noted that currently 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 [45]) is deliberately kept to a minimum.

So what exactly is an argument and how does it differ from a logical proof? Following the explanation of this question from [7], we first assume a fixed Δ (a finite set of formulae). Essentially this Δ can be thought of as a large repository of information or a Knowledge Base (KB) from which arguments can be constructed for and against arbitrary claims. Apart from information being understood as declarative statements there is no a priori restriction on the contents. Indeed Δ is not even expected to be consistent. As discussed previously, this survey is only concerned with symbolic approaches to reasoning with uncertainty, and as such no preference ordering is assumed over the contents of Δ. In many frameworks an argument is simply a pair where the first item is a minimal consistent set of formulae that proves the second item. As mentioned before, we are assuming that only deductive arguments are considered. We believe the following widely used definition was first proposed by Simari [71].

Definition 1 *An argument is a pair* $< \Phi, \alpha >$ *such that:*

1. $\Phi \nvdash \bot$
2. $\Phi \vdash \alpha$

3. Φ *is a minimal subset of* Δ *satisfying 2.*

The key thing is that an argument has the form of a logical proof (using a consistent subset of a KB). So once established an argument will always stay valid as additional knowledge is added to the KB. In argumentation systems the basic notion is not that of a defeasible conditional, but that of a defeasible argument. Nonmonotonicity, or defeasibility is not explained in terms of the interpretation of a defeasible conditional (as in default logics), 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 counterarguments, which may defeat the original argument.

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 among 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 The key ingredients

Argumentation systems are generally comprised of the following five elements (although sometimes implicitly) [62]: an underlying logical language, definition of an argument, of conflicts between arguments and of defeat among arguments, and 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 system is, as argumentation systems are built around an underlying logical language and an associated notion of logical consequence which defines 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 commonly discussed in the literature, rebutting, assumption attacks and undercutting (see [62] for further explanation.) The notion of defeat is a binary relation on the set of arguments, but it is important to note that this relation does not indicate with what arguments a dispute can be won; it only indicates something about the relative strength on two individual conflicting arguments. Therefore what is also needed is a definition of the status of 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 into 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. Accordingly, the declarative form of an argumentation system can be regarded as its (argumentation-theoretic) semantics, and the procedural form as its proof theory.

2.4 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 to provide 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 work [34] there have been many proposals for such a notion of acceptability including admissible, preferred and stable semantics. Dung's basic formal notions are as follows:

Definition 2 *An argumentation framework (AF) is a pair (Args, defeat), where Args is a set of arguments, and defeat a binary relation on Args.*

- *A set of arguments is* conflict free *iff no argument in the set is defeated by an argument in the set*
- *An argument A is acceptable with respect to a set S of arguments iff each argument defeating A is defeated by an argument in S.*
- *A conflict-free set of arguments S is admissible iff each argument in S is acceptable with respect to S.*

Definition 3 *In terms of the notions of acceptability and admissibility several notions of "argument assignments" can be defined (which are referred to above as "status assignments").*

- *A conflict-free set S is a stable extension iff every argument that is not in S is defeated by some argument in S*
- *A conflict-free set is a preferred extension iff it is a maximal (with respect to set inclusion) admissible set.*

When dealing with argument systems, questions often boil down to the following two fundamental problems which lead to an additional classification of the type of reasoning performed: Should this argument be accepted in all possible worlds? That is, should everyone accept this argument (which is more commonly referred to as sceptical reasoning)? Is there a possible world in which this argument must be accepted; can someone defend this argument (credulous reasoning)? According to Vreeswijk these problems are relatively well understood [77]. 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 [36] [31]. The second approach is interested in the design of algorithms with the intention to actually use them in practice, and this is the area most related to our desire to survey the field of practical implementations of argumentation.

2.5 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 and defence sets. The decision as to which of the two types is most appropriate depends on the reasoning scenario. 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 [7]). The second category has been explored by 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 [60] (which have been successfully implemented in the OSCAR software suite [59]). 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 [19] (an argumentation system is coherent iff preferred and stable extensions coincide). The third category, argumentation as a dialectical process, has by far received the most attention in the recent literature. An example of blurring the lines between the second and third categories is provided by Dung, Kowalski and Toni, who have presented a family of dialectic proof procedures for the admissibility semantics of assumption based argumentation [35]. There have also been numerous contributions to the study of argumentation solely as a dialectic practice, and in particular for this survey we are interested in further exploring 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 [70]). 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 [74], a practical implementation of dialogue based argumentation (discussed in Section 4.3). Further work from Vreeswijk includes a dialectic-based query algorithm and another practical implementation which can determine the acceptance of arguments based on the semantics of credulously preferred sets [78]. The latest work of Vreeswijk is presented in [77] where he outlines an algorithm that computes grounded and admissible defence sets based on credulous preferred acceptance in one pass for single arguments.

3 Challenges with argumentation

3.1 How difficult can it be to argue?

The brief overview of argumentation provided in the previous section has attempted to illustrate that there are many diverse algorithms available and opportunities for them to be adapted for use within a software-based reasoning component. However, there are many challenges to be faced when attempting to implement argumentation-based reasoning.

We have already highlighted that computational complexity is potentially a big issue, with many algorithms lacking a complexity analysis and therefore it remains unclear how well these algorithms perform in practice. This may be responsible for the absence of uptake of this type of logical reasoning formalism within existing software 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 computationaly complex (intractable) first-order logic, which has been argued as essential for real-world reasoning by the likes of [60], and [7], or a less expressive, but more tractable defeasible logic, proponents of which include [57], [52] and [40]. 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 [7], and for an overview of argumentation-theoretic semantics Prakken and Vreeswijk's discussion [62] of the topic and associated challenges is recommended.

After 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 to compute all arguments, all extensions, or do we want 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 paper, and both approaches have their pros and cons. A total algorithm may be relevant to software applications deployed into a small-scale and relatively static environment, for example, we may desire to identify all the arguments that are available resulting from both current knowledge and beliefs in order to make the best decision. On the other hand, software applications 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. Vreeswijk [77] argues that these systems can only rely on query answering algorithms as arguments in first-order systems are constructed dynamically and therefore cannot be known in advance. 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, the work of Dimopoulos *etal* [30] suggests that in terms of complexity such brute force methods are perhaps the best we can achieve.

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 [7]. Typically a minimal consistent subset of the available knowledge base is presented in support of an argument, but determining whether a set of formulae in propositional calculus is classically consistent is an NP-complete decision problem [42] and worse still determining whether a set of first-order formulae is classically consistent is an undecidable decision problem [10]. 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 [42]. 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

[36]. 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 [31]. On the other hand, it has been observed that 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 defeasible 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 [52], 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 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 [25] resorting to randomly generated programs and theories, a solution often used in other areas such as graph algorithms or satisfiability testing, may 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 [25], 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 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.

3.2 What can be done?

Designing a computationally efficient algorithm to conduct argumentation-based reasoning is clearly not a trivial task. However, over the last decade much progress has been made on this issue, for example, [19], [46], [77] and [78] have all published algorithms that could be implemented, but most have complexity issues. There has also been much discussion in the academic literature about the computational complexity of argumentation, but 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 [7], is referred to as argument compilation (the discussion relates to algorithms working in first order logic, but it is argued that the techniques could be generalised to other formalisms). Argument compilation is a process that improves efficiency by finding all the minimal inconsistent subsets of the knowledge base and then forming a hypergraph from this collection of subsets. Whilst the process of argument compilation is expensive, once the knowledge base has been compiled, it can be queried relatively cheaply and the algorithm provided in [7] 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, [13]. 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 knowledge base as well as the defeat relationship among them, and can be understood in more general terms as a graph from which all possible computable dialectical trees 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 knowledge base is stratification, which was highly utilised in making early default reasoning implementations computationally efficient, for example, DeReS [25] (it should be noted that there is little work on the use of stratification within argumentation-based reasoning). Essentially, stratification was used to decompose the collection 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. As with knowledge compilation, stratification techniques are initially computational expensive, but it could be envisioned that with a relatively static knowledge base the compilation or stratification process could be conducted when appropriate, for example, overnight. However, 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 [13] 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 [7] which is a principled means for intelligently querying an ATP in order to search a knowledge base 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 knowledge base. 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 naive searching for arguments and counterarguments [7]. 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 when required. Capobianco and Chesnevar [14] also offer a similar scheme to argument contouring for Simari and Loui's early work on the Mathematical Treatment of Defeasible Reasoning framework [70]. 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.

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 contours are for use with a static knowledge base, 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 knowledge base compilation techniques that if the knowledge base is changing frequently this cost could be prohibitively high.

The third category of efficiency improving techniques is based on generating approximate arguments. Pollock [59] 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 in [7] 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 paper 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 [45]. Typically these strategies concentrate on dramatically reducing 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 colleague's Defeasible Logic Programming system [40]. 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 in which nodes are expanded greatly affects the size of the search space [67]. Chesnevar and colleagues [23] 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 literals as possible with the 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 [6] or how an argument resonates with an audience, based on empathy or antipathy [7]. 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 dialectical trees or reduce the number of arguments generated [45]. 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 previously in this article, 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 [77] discusses that in [32] 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, Vreeswijk states 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 [77]. 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 ([54] describes in detail how to conduct such experiments). As an example, Vreeswijk argues that the implementation details and average case complexity analysis included in his recent work [77] 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 [52] 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 undisputed inferences, circular inferences (which are unsolvable) and inferences involving various widths and depths of rule trees. Cholewinski and colleagues have also proposed a similar tool called TheoryBase [25] 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. [25] argue that this overcomes two of the fundamental problems with generating random problems, although it remains to be seen whether such a technique could be applied to generate a random argumentative knowledge base utilising existing real world problems.

4 Review of Existing Reasoning Engines

This section of the article examines previous implementations of both defeasible and argumentation-based reasoning engines and associated literature. This review focuses 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 [62] [22] or [7, Chap 10] should be consulted. Currently only systems that rely on symbolic approaches to dealing with inference under uncertainty have been surveyed. 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.

4.1 Logic programming - introducing non-monotonicity

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 [26] and Kowalski [47] as Logic Programming (LP). It introduced to computer science the important concept of declarative - as opposed to procedural - programming. The Prolog language [26] 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 in understanding the framework provided. There are many good text books that discuss Logic Programming within the context of automated reasoning. One of these, such as [1], should be consulted for more detail.

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" [1]. 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, and 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 [34] is particularly relevant. Dung presents a theory for argumentation whose central notion is the acceptability of arguments. He introduced the notion of an *argument framework* in which the attack relation between arguments is represented as a di-graph. In addition, 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.

4.1.1 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 [50]. 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 [43] 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 (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 [1]. These arguments led to the introduction of classical negation in combination with default negation, in what was coined as "Extended Logic Programming (ELP)" [44]. Recently the Well-Founded Semantics with eXplicit negation (WFSX) [1] incorporated into the language of logic programs an explicit form of negation in addition to the previous implicit negation, related the two, and extended 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 "rule base". 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 a metainterpreter, 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.

4.2 Defeasible Logics - Spanning the ages

A development closely related to defeasible argumentation is so-called 'defeasible logic' [57]. 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 previously stated, although this paper 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 knowledge base 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 this survey.

4.2.1 Nathan (1992)

Nathan [48] resulted out of early work on logics for defeasible reasoning by Simari and Loui (specifically their Mathematical Treatment for Defeasible Reasoning framework [70]), 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, 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." [70]. They continued by discussing that Pollock's system was "too general for AI's use" [70] 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 ANSI C, and provides a very simplistic command line driven interface to the user. The knowledge base is defined in an identical manner to a Prolog rule base and contains a finite set of definite clauses and defeasible clauses, possibly containing atoms affected 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 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 [70]). 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 identified 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 knowledge base 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 [70]) created a foundation on which much other work was conducted, in particular work on DeLP [40] and also early work on utilizing implementation techniques to overcome the inherent computational complexity issues [14](both of which are discussed later).

4.2.2 d-Prolog (1993)

d-Prolog [56] is a nonmonotonic extension of the Prolog programming language based on Nute's defeasible logic [57] (which in turn was based on Nute's earlier work on Logic for Defeasible Reasoning (LDR) [55]). 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 [62]. 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 defeasibly derive conclusions 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' or 'can't tell'. These two query types essentially provide credulous and sceptical acceptability respectively. Nute used 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 [56] 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 [66] (reviewed later), polynomial time complexity is associated with d-Prolog. In addition, both Maher and colleagues [52], and Antoniou and Bikakis [4] 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 [52]. Work by Antoniou and Bikakis demonstrates that d-Prolog is able to handle a relatively large knowledge base, with the maximum number of rules limited by the underlying Prolog implementation (standard settings in SWI-Prolog typically allows around 20000 rules [4]). However, in results from tests conducted by Maher *etal* [52] 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 as 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 "[56]. The relatively flexibility and robustness of this early implementation of defeasible logic is demonstrated in several practical systems and has also been exploited commercially, for example, for controlling expert systems recommendations [58] and logical control of an elevator system [27].

4.2.3 EVID (1994)

The EVID [17] [16] 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 knowledge base in standard nomenclature) to use together with the EVID shell. A knowledge base 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 as failure extensively. An application knowledge base also typically includes some relative and absolute defeater rules. The user runs both EVID and the knowledge base in the Prolog compiler and typically enters 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 was 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 override 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 discuss complexity issues in much detail), his discussion of adequacy does appear to support our view regarding the importance of 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 we do not believe that they provide very good understanding of the typical uses of defeasible reasoning in practical contexts." [16].

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 would be times when the user wants to determine what additional user data would enable the system to infer a conclusion which it currently cannot, including what additional data would serve 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" [16]), either relative to particular supporting evidence, or absolutely, providing that the user does not contradict himself in doing so. Causey states "We do not want the user to perform epistemologically irrational actions. Obviously we would not want the user to add p and *not* p and EVID prevents this" [16]. However, we would have to question this restriction in an automated reasoning application. Frequently real world knowledge may 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 knowledge base. Causey states these issues are the goal of future work.

4.2.4 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 [59], a fully implemented architecture for rational agents written in LISP and 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 the reasoning conducted within OSCAR as a dialectical justification process, although Pollock is often not explicit on this point [22]. 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 [59] 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, *personalcommunication*), unfortunately we were unable to locate them.

4.2.5 Deimos and Delores (2001)

Maher and colleagues [52] present 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" [52]. 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 [63], 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 [51]. 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 [52], 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 [5]). Maher and colleagues state that the transformation should 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 [33]. 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" [52]. 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 [7].

Efficiency within both implementations is clearly an important factor for the authors as in [52] 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 [52] 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 inferences, 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 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 types 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 priori 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.

4.2.6 Phobos (2001)

Phobos [65] is an implementation of propositional plausible logic written by Rock and Billington [66], [8]. 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 [64]. However, as with Deimos the web interface provides full accessibility to the application in a fairly intuitive manner. Although 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.

4.2.7 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) [40]. 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 $http://lidia.cs.uns.edu.ar/delp\_client/$). Additionally, an abstract machine named Justification Abstract Machine (JAM) has been designed for the implementation of DeLP as an extension of the 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; for example, a web-search engine recommender system [21], a reasoning framework for agents within dynamic environments [15], and a framework for stock-market trading agents [38]. 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 [15] (discussed later) which contains extra machinery to handle dynamic scenarios. DeLP loads strict and defeasible rules via a 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 discusses how the DeLP engine has been incorporated into a reasoning component within the Jinni framework [72]. 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 (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 colleagues believe that common sense reasoning should be defeasible in a way that is not explicitly programmed [40]. 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 [20] 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 colleague's work, e.g. [40, 38, 15], 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 knowledge base 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 affects the size of the search space. Chesnevar and colleagues [23] proposed 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 [13] define a mechanism in their ODeLP formalism to limit the expensive creation of dialectic trees in a dynamic environment where an agent's 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 paper there are different options for integrating precompiled knowledge into a reasoning mechanism. A simple approach would be to record every argument that has been computed so far, as proposed in Capobianco and colleagues earlier work [14]. 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 knowledge base as well as the defeat relationships among them. Capobianco and colleagues refer to this precompiled knowledge base as a dialectical database [13]. 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 as 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 [39]. 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 (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 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.

4.2.8 DR-PROLOG (2007)

DR-PROLOG [4] 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 [4]. 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 [4]). 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 its 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 interface appears to be very flexible as 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 [4]. Only priorities between conflicting rules are used, as opposed to systems of formal argumentation where often more complex kinds of priorities (for example, 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 test 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.

4.3 Argumentation - The emerging technology

4.3.1 IACAS (1993)

IACAS (InterACtive Argumentation System) [74] 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 in Vreeswijk's PhD thesis on defeasible argumentation, referred to as 'abstract argumentation systems' [75]. 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 benchmark problems) and was not offered by many other implementations at this time, notably Nute's d-Prolog [56] (which could cause non-intuitive results to be concluded [62, p. 295]).

Vreeswijk [74] identifies several reasons regarding 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 [56], early work on Pollock's OSCAR [60] and work on Loui et al's LMNOP [49] (which was later incorporated into NATHAN [48]). 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" [74]. 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' [24]. 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 ideas advanced in Vreeswijk's thesis really did work, and there were few such demonstrations before this implementation.

4.3.2 Argumentation System (AS) (2005)

Vreeswijk's Argumentation System (AS) is an argumentation-based reasoning engine written in the Ruby programming language. The core prover accepts formulae in an extended first-order language and returns answers to queries on the acceptability of arguments using the semantics of credulously preferred sets. The theoretical work behind the engine's algorithm and implementation originates from [46, 18, 19]. Argument games were formalised in [46], where a general framework is proposed which enables argument games to be defined for winning positions in argumentation frameworks. Following this formal approach, but with slightly different defintions a general framework for argument games were proposed in [18, 19]. It is from this work that the two argument games for the credulous acceptance problem under the preferred semantics used within AS are taken. Currently the implementation provides a web interface [78] 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" [78].

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. This is, however, provided in Vreeswijk's subsequent work [77], which we discuss next.

4.3.3 Vreeswijk's admissible defence sets (2006)

In the same spirit as AS, Vreeswijk presents another algorithm and implementation [77] of an argumentation system that computes grounded and admissible defence sets in one pass (that is, 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 ealier in this article, algorithms to compute grounded and/or preferred extensions have been proposed previously, see for example [19, 46], but these algorithms only 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 [79]. 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 (a collection of typical di-graphs) was composed and is available on the aforementioned website. 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 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 paper, 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 [54] 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 in his opinion the presentation of an algorithm must be accompanied by a conventional complexity analysis first, before it can be subject to practical tests.

4.3.4 Argue tuProlog (2006)

Argue tuProlog (AtuP) [12, 11] is a prototype implementation of an argumentation engine based on Vreeswijk's theoretical and practical work from [78]. The authors state that although there are several notable works on the topic of practical implementation of an argumentation system none seem to offer the flexibility that will be required to enable argumentation to be of practical use within contemporary software applications (for example, the authors state that an engine will be required to support multiple types of argumentation formalism). The primary motivations for their work include that they feel it is important to make publicly available their and other "pragmantic" implementation of argumentation together with some large-scale benchmarking knowledge bases. They continue by stating that as far as possible they are aiming towards implementing a general purpose argumentation engine that can be configured to conform to one of a range of semantics, and AtuP is a first step towards this goal. Currently, as with Vreeswijk's system described previously, AtuP accepts formulae in an extended first-order language and returns answers to queries of acceptability on the basis of the semantics of credulously preferred sets. 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. Strict and defeasible rules can be entered into the system and typically exactly one query is supplied. 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 AtuP tries to build an admissible set around an argument for the main claim. AtuP is currently implemented in Java and is presented as a self-contained component that can be integrated into a range of applications (the authors specifically mention agents applications as an intended target) by utilising the well-defined application programming interface (API) provided. The API exposes key methods to allow

a developer to access and manipulate the knowledge base, to construct rules, specify and execute queries, and analyse results. The core engine has been built using tuProlog [29, 28], an existing open-source Prolog engine as its foundation which followed the same design principles of the intended domain of application. tuProlog is a Java-based Prolog engine which has been designed from the ground up as a thin and light -weight engine that is deployable, dynamically configurable and easily integrated into Internet or agent applications. Using the Prolog inference provided by the tuProlog engine a series of meta-interpreters can be implemented for a variety of forms of argumentation. However, the authors state that this way of implementing an argumentation engine has both a serious performance overhead and a less than ideal interface. Accordingly, tuProlog is being reengineered by implementing a series of core argumentation algorithms in Java, effectively pushing the functionality of the algorithm down into the core engine. In addition to providing an API to allow agent developers to utilise the engine the authors have also modified the existing tuProlog graphical user interface to facilitate off-line experimentation with the engine. The core engine has also been developed using Sun Microsystem's Netbeans graphical IDE [53], in which an application profiler has been installed. This allows deployment of the engine within a simulated variety of realistic scenarios, and the monitoring and analysing of such data as CPU usage, memory usage, program loop/branch counting, thread profiling and other basic JVM behaviour. The authors are also in the process of setting up several large-scale knowledge bases, and when this is complete the profiling tool will facilitate the ultimate goal of obtaining empirical evaluations of the performance of a wide range of argumentation models. No examples are provided in [12, 11] and instead the interested reader is referred to Vreeswijk's work for these details. A number of problems connected with the implementation were also discovered after presentation at the JELIA conference [11], which can lead to incorrect conclusions being drawn. Future work intends to correct all of these problems before the engine is released to the general public.

4.4 CaSAPI (2007)

The CaSAPI [37] system is a Prolog implementation for credulous and sceptical argumentation based upon the computation of dispute derivations for assumption-based argumentation frameworks. The system relies upon a generalisation of the original assumption-based argumentation framework and of the computational mechanisms proposed by Bondarenko, Dung, Kowalski and Toni in [9] whereby multiple contraries are allowed. The authors claim this generalisation is useful to widen the applicability of assumption-based argumentation to allow, for example, reasoning about decisions. The underlying theoretical mechanism is defined as "dialogues" between two fictional agents; the proponent and the opponent, trying to establish the acceptability of given beliefs with respect to the chosen semantics. In order to determine whether a belief is to be held, a set of assumptions needs to be identified that would provide an "acceptable" support for the belief, namely a "consistent" set of assumptions including a "core" support as well as assumptions that defend it.

This informal definition of "acceptable" support can be formalised in many ways, using the accepted notion of "attack" amongst sets of assumptions [34]. In [37] the authors outline three such mechanisms (and also identify the corresponding semantics) that affects the level of scepticism in the proponent agent: in GB-dispute derivations the agent is not prepared to take any chances and is completely sceptical in the presence of seemingly equivalent alternatives (implementing the sceptical grounded semantics); in AB-dispute derivations the agent would adopt an alternative that is capable of counter-attacking all attacks without attacking itself (the credulous admissible semantics); in IB-dispute derivations the agent is wary of alternatives, but is prepared to accept common ground between them (the sceptical ideal semantics).

The implementation of the CaSAPI system is invoked using a standard Prolog interpreter. Users are required to load the input assumption-based framework and the beliefs to be proved in the same manner as facts and rules are added to any Prolog program (typically using text files).

Rules are represented as facts of a binary relation $myRule/2$ consisting of a left- and right-hand side. The first argument holds the head of the rule and the second argument a list containing the body of the rule. Assumptions and beliefs to be proved are represented as unary predicates $myAsm/1$ and $toBeProved/1$ respectively, using a list notation for their respective argument. The notion of contrary can also be customised using a binary relation $contrary/2$. The users can then control the kind of dispute derivation they want to employ (GB, AB or IB), the amount of output to the screen (silent, compact or noisy) and the number of supports computed (one or all). They specify their choices as arguments to the command $run/3$. For example, in order to run AB-dispute derivations in silent mode and obtain only one answer the user would specify: $run(ab, s, 1)$.

Currently the authors state that Sicstus Prolog is the implementation language of choice since they intend to employ some of its constraint solving features in future versions of CaSAPI (although version 2.0 of CaSAPI does not make use of any Sicstus specific code and hence it should run on most standard Prolog engines.) An interesting design choice made by the authors is the fact that the argument selection strategies of the agents are not hard coded into CaSAPI. Although different selection strategies do not affect the result of the argumentation process, they can have a significant impact on efficiency. The authors state that these strategies control how the dispute trees are generated and hence can lead to early pruning for certain trees (as discussed in work by Garcia and colleagues [40]). Apart from the above mentioned selection strategies there is little in the way of discussion regarding computational complexity issues, or indeed experimentation. However, a worked example is included in [37], as well as several example applications.

Gaertner and Toni [37] consider and discuss several practical application scenarios; non-monotonic reasoning (using logic programming), legal reasoning (where different regulations need to be applied, taking into account dynamic preferences amongst them), practical reasoning (where decisions need to be made as to which is the appropriate course of action for a given agent), and reasoning to support autonomous agents (about their individual beliefs, desires and intentions, as well as relationships amongst them). Most of these application scenarios require a mapping from appropriate frameworks into assumption-based argumentation, and the authors state that currently two of these application areas (legal and practical reasoning) assume a translation (by-hand) from a given formalism into assumption-based argumentation. However, the authors state that future work includes providing an appropriate front-end to the systems in order to automate the translation, presumably in a similar fashion to the pre-processing of other systems such as Delores [52] and DR-PROLOG [4]. Additional future work includes formalising a number of extensions to theoretical assumption-based argumentation (with the use of variables in rules, for example), and investigating the use of preferences, as modelled in legal reasoning, to provide effective means of conflict resolution.

5 Discussion

This article has presented an overview of existing work conducted on defeasible and argumentation-based reasoning engines. Our discussion has included a review of a wide range of types of formalisms, semantics, algorithms and technical implementation details, an overview of which can be seen in Table 1.

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 loss of performance in comparison with compiled implementations (as indicated by Maher and colleagues experimental results [52]), although this is of decreasing importance with modern interpreter design. 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 enforces that a knowledge base is loaded in as a text file containing strict and defeasible rules. 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 knowledge base rules in RuleML format. A foundational issue does remain, however. Prolog strictly adheres to its own execution order (DFS) when it comes to search and is not very flexible when the programmer wishes to apply a different execution philosophy, for example BFS, or lazy evaluation. This inflexibility is surmountable for much functionality (admissible set construction etc.) but not for argument generation that has its own mechanics. Generation of arguments is best programmed in languages that support lazy/on-demand/just-in-time generation of data structures with functionality such as threads (Java), continuations (Scheme), or yield mechanisms (Python).

From a theoretical aspect we can observe that several of the early implementations did indeed inherit problems associated with Prolog, such as looping with certain theories, as evident in the tests conducted by Maher and colleagues [52] on d-Prolog. In addition early implementations sometimes computed unintuitive conclusions due to the reliance on Prolog semantics which allows a statement to be labelled as only provable or not provable (that is, 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 colleague's DeLP utilises consistency constraints when generating dialectical argumentation trees to prevent logical fallacies from occurring. 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 [40] believe that common sense reasoning should be defeasible in a way that is not explicitly programmed. However, other authors, such as Maher and colleagues [52] and Antoniou and Bikakis [4] argue that knowledge in many domains is naturally represented using priority relations. Other authors, such as Pollock and Vreeswijk appear to take the middle ground - offering support for numerical values, but not exclusively relying on them in the inference process.

Table 1 Defeasible Reasoning Engines Overview

System Name (ref)	Reasoning	Lang Supported	Technology	Interface	Complexity Analysis	Empirical Eval	Extra Notes
Nathan ([48])	Defeasible	First order	ANSI C	Command line	None	Simple benchmarks	None
d-Prolog ([56])	Defeasible	First order	Prolog meta	Command line	None	Simple benchmarks (See [52] and [4])	None
EVID ([16])	Defeasible	First order	Prolog meta	Command line	None (although discussed)	Simple benchmark	*howdefeatit* query
OSCAR ([61])	Defeasible	First order	LISP program	Command line	None (although considered)	Unavailable	Used within "rational agent" implementation
Deimos ([52, 63])	Defeasible	Propositional	Haskell	Web and Command line	Undertaken (available in seperate paper)	Comprehensive benchmarking with DTScale	Only supports query answering
Delores ([52])	Defeasible	Propositional	C	Command line	Undertaken (but not detailed)	Comprehensive benchmarking with DTScale	Only supports total answering and requires preprocessing of theory
Phobos ([65, 66])	Plausible	Propositional	Haskell	Web and Command line	Undertaken (but not included)	Comprehensive benchmarking with custom tools	None
DeLP ([40])	Defeasible (dialectic argumentation)	First order	Prolog/JAM	Web, Agent component and command line	See [20]	Simple benchmarks and anecdotal evidence	None
DR-PROLOG ([4])	Defeasible	First order	Prolog meta	Web and command line	None	Comprehensive benchmarking with DTScale	RuleML support and ambiguity propogation
IACAS ([76])	Argumentation	First order	LISP program	Command line	None	Simple benchmarks	Chisholms Theory of Knowledge support
AS ([78])	Argumentation	First order	Ruby	Web (command line)	None	Benchmarks	Produces diagram of argument structure
Vreeswijk's Admissible Defence Sets ([77])	Argumentation	First order	Ruby	Web (command line)	Detailed complexity analysis	Comprehensive examples	Discussion of empirical complexity analysis
Argue tuProlog ([12])	Argumentation	First order	Java	Software Component and GUI	None	None (although discussed)	API to allow integration
CaSAPI ([37])	Argumentation	First order	Prolog meta	Command line	None	Simple examples	Translation (by-hand) from a given formalism possibly required

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 strongly argued against the use of "a body of simple and eclectic examples about birds and penguins etc." [16] to test defeasible reasoning systems. However, only around the start of 2001 did authors of defeasible reasoning implementation begin to discuss in depth technical methods to improve efficiency and include comprehensive experimental results within their work. Examples of this type of work include Maher and Colleagues [52], Rock and Billington [66], and more recently work by Antoniou and Bikakis [4].

In this article 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 empirical experiments can be seen with the evaluation of the Deimos and Phobos implementations which confirmed the expected theoretical complexity results. Perhaps more interesting is the empirical analysis of Maher and colleague's Delores implementation [52], which 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 the implementation may have gone unnoticed. Also worth noting is that the evaluations 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 those used within testing of early implementations) he has also been advocating the need to empirically evaluate an algorithm's performance using, for example, techniques as discussed by Nudelman [54]. However, the provision of benchmark examples has a serious challenge. As will be clear from this article, there many different models of formal argument, often with specific notions of argument. This has consequences, of course, impacting on the interoperability of argument systems and the exchange of test sets. We are currently investigating this issue in more detail.

We would like to draw attention to several other interesting aspects of existing work that has become apparent as the survey was conducted. First, buried within Causey's work on the EVID implementation he states that there appears to be no consensus as to what is required within a defeasible reasoning implementation. Although this has not been fully investigated, we believe this may still be a problem today. Causey's ideas on creating a desiderata for such reasoning engine implementations are interesting. These include insisting an engine must provide detailed explanation of justifications and allowing the user to over-ride any conclusions made by the application. However, they 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 user's understanding of the reasoning process is practically a universally accepted requirement of

defeasible and argumentative reasoning engines, as this was implemented in even the very first implementations. Increasingly these traces have become more advanced, with Gerard Vreeswijk's recent work [77] generating a very useful graphical representation of the argument relations. The inclusion of functionality within the EVID implementation 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. Recent work by García *et al* continues the to develop the theme of explanation [41].

Overall, we hope that this article has contributed to raising awareness that research on models of argumentation is on the cusp of a move into a phase of engineering applicable implementations. Continued research into the empirical evaluation of models of argumentation will continue to ensure this subject area has real societal benefit.

References

[1] Jose Julio Alferes and Luis Moniz Pereira. *Reasoning with Logic Programming.* Springer-Verlag New York, Inc., 1996.

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

[3] G. Antoniou. *Nonmonotonic Reasoning.* MIT Press, 1997.

[4] G. Antoniou and A. Bikakis. Dr-prolog: A system for defeasible reasoning with rules and ontologies on the semantic web. *Knowledge and Data Engineering, IEEE Transactions on*, 19(2):233–245, 2007.

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

[6] T. J. M. Bench-Capon. Persuasion in practical argument using value-based argumentation frameworks. *J Logic Computation*, 13(3):429–448, 2003.

[7] P. Besnard and A. Hunter. *Elements of Argumentation.* In preparation, 2006.

[8] D. Billington and A. Rock. Propositional plausible logic: Introduction and implementation. *Studia Logica*, 67(2):243–269, 2001.

[9] A. Bondarenko, P. M. Dung, R. Kowalski, and F. Toni. An abstract, argumentation-theoretic approach to default reasoning. *Artificial Intelligence*, 93(1–2):63–101, 1997.

[10] G. Boolos, J. Burgess, and R. Jeffrey. *Computability and Logic.* Cambridge University Press, 2002.

[11] D. Bryant and P. J. Krause. 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*, pages 469–472. Springer, 2006.

[12] D. Bryant, P. J. Krause, and G. Vreeswijk. Argue tuprolog: A lightweight argumentation engine for agent applications. In *Proceedings of 1st International Conference on Computational Models of Argument (COMMA06)*, pages 27–32. IOS Press, 2006.

[13] M. Capobianco, C. Chesnevar, and G. Simari. 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.*, 2004.

[14] M. Capobianco and C. I. Chesnevar. Introducing dialectical bases in defeasible argumentation. In *Proceedings of the 6th Workshop on Aspectos Teoricos de la Inteligencia Artificial (ATIA)*, pages 1–10, San Juan, Argentina, 1999.

[15] M. Capobianco, C. I. Chesnevar, and G. R. Simari. Argumentation and the dynamics of warranted beliefs in changing environments. *Autonomous Agents and Multi-Agent Systems*, 11(2):127–151, 2005.

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

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

[18] C. Cayrol, S. Doutre, and J. Mengin. Dialectical proof theories for the credulous preferred semantics of argumentation frameworks. In *ECSQARU '01: Proceedings of the 6th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pages 668–679, London, UK, 2001. Springer-Verlag.

[19] C. Cayrol, S. Doutre, and J. Mengin. On decision problems related to the preferred semantics for argumentation frameworks. *J Logic Computation*, 13(3):377–403, 2003.

[20] L. A. Cecchi, P. R. Fillottrani, and G. R. Simari. On the complexity of delp through game semantics. In *Proc. 11th Intl. Workshop on Nonmonotonic Reasoning (NMR 2006). J. Dix and A. Hunter (Eds.)*, pages 386–384. IfI Technical Report Series, Clausthal University, 2006.

[21] C. I. Chesnevar and A. G. Maguitman. Arguenet: an argument-based recommender system for solving web search queries. In *Proceedings. 2004 2nd International IEEE Conference in Intelligent Systems*, pages 282–287, 2004.

[22] C. I. Chesnevar, A. G. Maguitman, and R. P. Loui. Logical models of argument. *ACM Comput. Surv.*, 32(4):337–383, 2000.

[23] C. I. Chesnevar, G. R. Simari, and A. J. Garca. Pruning search space in defeasible argumentation. In *Proc. of Workshop on Advances and Trends in Search in Artificial Intelligence*, pages 40–47, 2000.

[24] R. Chisholm. *Theory of Knowledge*. Prentice-Hall, New Jersey, 1997.

[25] P. Cholewinski, V. W. Marek, A. Mikitiuk, and M. Truszczyński. Computing with default logic. *Artif. Intell.*, 112(1-2):105–146, 1999.

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

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

[28] E. Denti, A. Omicini, and A. Ricci. tuProlog: A light-weight prolog for internet applications and infrastructures. In *PADL*, pages 184–198, 2001.

[29] E. Denti, A. Omicini, and A. Ricci. Multi-paradigm java-prolog integration in tuProlog. *Sci. Comput. Program.*, 57(2):217–250, 2005.

[30] Y. Dimopoulos, B. Nebel, and F. Toni. Finding admissible and preferred arguments can be very hard. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *KR2000: Principles of Knowledge Representation and Reasoning*, pages 53–61, San Francisco, 2000. Morgan Kaufmann.

[31] Y. Dimopoulos, B. Nebel, and F. Toni. On the computational complexity of assumption-based argumentation for default reasoning. *Artif. Intell.*, 141(1):57–78, 2002.

[32] Y. Dimopoulos and A. Torres. Graph theoretical structures in logic programs and default theories. *Theor. Comput. Sci.*, 170(1-2):209–244, 1996.

[33] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *J. Log. Program.*, 1:267–284, 1984.

[34] P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77(2):321–358, 1995.

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

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

[37] D. Gaertner and F. Toni. Casapi - a system for credulous and sceptical argumentation. In *Proceedings of First International Workshop on Argumentation and Nonmonotonic Reasoning, Arizona, USA*, 2007.

[38] A. Garcia, D. Gollapally, P. Tarau, and G. Simari. 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, 2000.

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

[40] A. J. Garcia and G. R. Simari. Defeasible logic programming: an argumentative approach. *Theory Pract. Log. Program.*, 4(2):95–138, 2004.

[41] A.J. García, N.D. Rotstein, and G.R. Simari. Dialectical Explanations in Defeasible Argumentation. In *Proceedings of the Ninth European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU-07)*, pages 295–307. Springer LNCS, 2007.

[42] M. Garey and D. Johnson. *Computers and Intractability*. W H Freeman., 1979.

[43] A. Van Gelder, K. A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[44] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

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

[46] H. Jakobovits and D. Vermeir. Dialectic semantics for argumentation frameworks. In *Proceedings of the 7th Int. Conf. on Artificial Intelligence and Law*, pages 53–65. ACM Press, 1999.

[47] R. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22:424–436, 1979.

[48] R. Loui and G. Simari. *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, 1994. Last accessed: 13th February 2007.

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

[50] J. Loyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[51] M. J. Maher, A. Rock, G. Antoniou, D. Billington, and T. Miller. *Deimos*. available from http://www.cit.gu.edu.au/ arock/defeasible/Defeasible.cgi, 2001. Last accessed: 10 Sept 2006.

[52] M. J. Maher, A. Rock, G. Antoniou, D. Billington, and T. Miller. Efficient defeasible reasoning systems. *International Journal on Artificial Intelligence Tools*, 10(4):483–501, 2001.

[53] Sun Microsystems. *Welcome to NetBeans.* available from http://www.netbeans.org/, 2007. Last accessed: 10 Sept 2007.

[54] E. Nudelman. *Empirical Approaches to the Complexity of Hard Problems.* PhD thesis, Stanford University, Stanford, CA., 2005.

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

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

[57] D. Nute. Defeasible logic. In Dov Gabbay, Christopher J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 3: Nonmonotonic Reasoning and Uncertain Reasoning*, pages 353–395. Oxford University Press, Oxford, 1994.

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

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

[60] John L. Pollock. How to reason defeasibly. *Artif. Intell.*, 57(1):1–42, 1992.

[61] John L. Pollock. Rational cognition in OSCAR. In *Agent Theories, Architectures, and Languages*, pages 71–90, 1999.

[62] H. Prakken and G. Vresswijk. Logics for defeasible argumentation. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic (Second Edition)*, pages 218–319. Kluwer Academic Publishers, The Netherlands, 2002.

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

[64] A. Rock. *Phobos.* available from http://www.cit.gu.edu.au/ arock/plausible/Plausible.cgi, 2006. Last accessed: 10 Sept 2006.

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

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

[67] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2nd edition, 2003.

[68] M. Schroeder. 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*, pages 140–149, London, UK, 1999. Springer-Verlag.

[69] D. A. Shortliffe. *MYCIN: Computer-Based Medical Consultations*. Elsevier, 1976.

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

[71] G.R. Simari. *A Mathematical Treatment of Defeasible Reasoning and its Implementation*. PhD thesis, Washington University in Saint Louis, 1989.

[72] P. Tarau. 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.*, pages 109–123, London, UK, 1999.

[73] P. Tolchinsky, S. Modgil, U. Cortes, and M.Sanchez-Marre. Cbr and argument schemes for collaborative decision making. In *Proceedings of 1st International Conference on Computational Models of Argument (COMMA06)*. IOS Press, 2006.

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

[75] G. Vreeswijk. *Studies in Defeasible Argumentation*. PhD thesis, Free University of Amsterdam, The Netherlands, 1993.

[76] G. A. W. Vreeswijk. IACAS: an implementation of Chisholm's principles of knowledge. In *The proceedings of the 2nd Dutch/German Workshop on Nonmonotonic Reasoning, Utrecht.*, pages 225–234, 1995.

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

[78] G. A. W. Vreeswijk. *Argumentation System (AS)*. available from http://www.cs.uu.nl/people/gv/code/AS/, 2006. Last accessed: 10 Sept 2006.

[79] G. A. W. Vreeswijk. *Vreeswijk's admissible defence sets*. available from *http*: *//www.cs.uu.nl/ gv/code/grd$_a$dm/*, 2006. Last accessed: 29 Sept 2006.