DeLP Viewer: a Defeasible Logic Programming Visualization Tool [⋆]

Sebastián Escarza[1,2], Martín L. Larrea[1,2], Silvia M. Castro[2], and Sergio R. Martig[2]

[1] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)
[2] Laboratorio de Investigación y Desarrollo en Visualización y Computación Gráfica (VyGLab)
Departamento de Ciencias e Ingeniería de la Computación (DCIC)
Universidad Nacional del Sur (UNS)
Bahía Blanca, Buenos Aires, Argentina
{se, mll, smc, srm}@cs.uns.edu.ar

Abstract. Defeasible Logic Programming (DeLP) is a knowledge representation formalism that combines results from Logic Programming and Defeasible Argumentation to provide reasoning based on contradictory and potentially incomplete information. DeLP allows information representation by using weak rules and provides an argumentation inference mechanism for warranting the entailed conclusions. It is necessary to comprehend the relationships between the arguments involved in DeLP derivation to understand the reasoning process and justify the replies provided by the DeLP system. In order to reach such a degree of understanding we present DeLP Viewer, a DeLP visualization tool for representing the inference process performed by a DeLP reasoner. Albeit there are many applications designed for argumentation visualization, our proposal provides a visual representation for the underlying logic and argumentative structure behind DeLP, allowing the interactive exploration of the entire reasoning process plus the internals of the involved arguments.

1 Introduction

Formal modeling of real world problems has been object of study in Computer Science since its beginning. These models give to intelligent agents the ability to deal with information, reason about it, and infer new knowledge. However, when a real world problem is modeled, it is usual to have potentially contradictory and incomplete information.

DeLP is a knowledge representation formalism that combines results from Logic Programming and Defeasible Argumentation to provide reasoning based on potentially incomplete and contradictory information. It provides the possibility of representing information in the form of weak rules in a declarative manner, and a defeasible argumentation inference mechanism for warranting the entailed conclusions [7].

The knowledge in a defeasible logic program can have intricate relationships that make complex the understanding of why some information is held and another is not. A set of dialectical trees representing the argumentation is built during the reasoning

[⋆] This work was supported in part by the PGI 24/N020 and 24/ZN19, Secretaría General de Ciencia y Tecnología, Universidad Nacional del Sur, Bahía Blanca, Argentina and the PICT 2003 N^{o}15043, ANPCyT, SeCyT, Argentina.

process. Such a set is called a *dialectical explanation* or δ*-Explanation*, and justify the status of a literal. The main worry for DeLP users is the construction of mental images about how arguments are confronted to derive conclusions, i.e., mental images about dialectical explanations.

In order to aid users to understand the justification of the derived conclusions, and to improve the analysis of the DeLP process, we present DeLP Viewer, a defeasible logic programming visualization tool. The goal is to assist users to gain insight into the dialectical process and, consequently, to easily design, understand, encode and debug defeasible programs. Our approach provides a node-link-based interactive visualization that preserves the internal rule-based structure of arguments, exploits the hierarchical structure of the dialectical tree, and allows on-demand exploration of the arguments content and their relationships.

This work is organized as follows. First we provide some preliminaries on DeLP to give the needed background to understand the problem we try to solve. Second we analyze related work in argumentation visualization and logic programming visualization. Third we present our tool, the rationale behind design decisions, the user domain constraints and the applied approaches. After that, we illustrate the main purpose of our tool in a concrete application scenario. Finally, we outline some conclusions.

2 Preliminaries on DeLP

DeLP is a logic programming language similar to *Prolog*, but it introduces additional constructors that enable the representation of potentially contradictory information and a defeasible argumentation inference mechanism. DeLP considers two kinds of program rules: *strict rules* used for representing strict (sound) knowledge, and *defeasible rules* used for representing defeasible knowledge, i.e., tentative information that may be used if nothing could be posed against it [7].

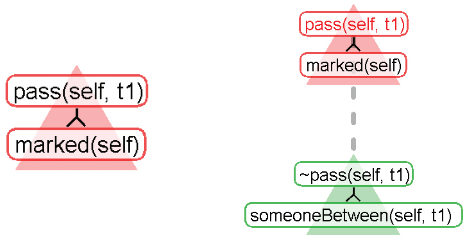
Rules in a DeLP program are combined to support or reject a claim. This combination is a logical derivation in which conclusions of some rules are used as premises of others using transitive logical dependencies. That combination of DeLP rules is an argument in favor of or against such claim (see Fig. 1). By contrast with other argumentation systems, arguments in DeLP are derived from the logic program and have internal structure. An argument is regarded as an explanation for a claim that is represented by a literal in such a logic program.

The claim of an argument may contradict some premise or the claim of another. In such a case, it is said that the contradictory argument *attacks* the other. Each attacker can be further attacked by many other arguments and these relationships between arguments result in a tree-shaped structure called *dialectical tree* (see Fig. 1). Every argument attacked by at least one undefeated argument becomes *defeated*, and every argument without (undefeated) attackers becomes *undefeated*. If the root argument of the dialectical tree results undefeated, that dialectical tree represents an argumentation that supports the claim of such argument.

Given a queried literal and a DeLP program, the DeLP reasoner builds a set of dialectical trees, trying to give support or contradict the arguments for or against the query. The sequence of obtained dialectical trees determines the reply of the system

$$
\begin{array}{lr}
\vdots & \\
\mathit{marked}(\mathit{self}) \leftarrow \mathit{true} & (1) \\
\mathit{pass}(P,M) \mathrel{-\!\!\prec} \mathit{marked}(P) & (2) \\
 & \\
\mathit{someoneBetween}(\mathit{self},t1) \leftarrow \mathit{true} & (3) \\
\sim\!\mathit{pass}(P,M) \mathrel{-\!\!\prec} \mathit{someoneBetween}(P,M) & (4) \\
\vdots &
\end{array}
$$

(a)



(b) (c)

Figure 1: Key concepts in DeLP. The figure shows a DeLP program fragment (a), an argument for $\mathit{pass}(\mathit{self},t1)$ that results from combining rules 1 and 2 (b), and a dialectical tree in which the root argument has been attacked and defeated (c). Rules in (a) have been numbered for referencing. Rules 1 and 3 are strict and rules 2 and 4 are defeasible. The literals labeled with true are removed from visualization for the sake of clarity.

and is the explanation for such a reply. More detailed descriptions about the dialectical process behind DeLP can be found in [7] and [11].

3 Related Work

Although Nonmonotonic Reasoning, Logic Programming and Defeasible Argumentation are not novel fields of research, only a few years ago the DeLP formal definition was published [7]. Since then, an ever growing theoretical frame and some practical applications [6, 11] have been developed. However, previous specific work on visualizing DeLP does not exist. DeLP combines Logic Programming and Defeasible Argumentation. In both areas, there are many visualization approaches.

Logic Programming Visualization is a Software Visualization subfield. The main efforts in this area are devoted to Prolog language execution visualization and debugging. *AORTA* diagrams are widely accepted as graphical representation of Prolog execution [2, 5, 4]. However, these approaches have not direct applicability to our problem. DeLP users want to comprehend the dialectical process after the reasoner's execution. They need to understand logic dependencies and argument relationships at a higher abstraction level than those provided by *AORTA* diagrams. DeLP users do not want to visualize how logical variables are bound. They want to understand the argumentation for the system reply.

In Defeasible Argumentation, many visualization tools with different purposes have been developed [8]. The most relevant are Araucaria [9], Rationale [1], Avers [12], Belvedere [13], Athena [10] and Reason!able [14]. These argument visualization tools were thought to help in the visual building of argumentations by hand. The user adds arguments, establishes their support/rebut relationships, and no automatic processing is performed at this level. All these tools use node-link diagrams to represent the argumentation and the nodes are represented by boxes or circles. The arguments' content is plain (unstructured) text in natural language. Also, these tools share the presence of rebut and support concepts, but under different terminologies and red/green or red/blue color schemes are used to distinguish between them. Additionally, some of the applications mentioned above have semiautomatic and automatic layout algorithms to help users to keep arguments visually arranged. Finally, all these tools provide a basic subset of interactions containing argument selection, scrolling and zooming.

Nevertheless, these applications present drawbacks to be applied in DeLP, because they were designed to represent arguments in natural language. But in DeLP, the arguments have an underlying logic that must be represented to understand the dialectical process. Such a logic structure, internal to each argument, cannot be represented by ordinary argumentation visualization tools, and, in consequence, interactions that enable the exploration of such structures are not provided by them either.

4 DeLP System Overview

Before delving into the visualization design, we will outline the macro-structure and the typical working flow of the DeLP System. The DeLP system is composed by two main components: the DeLP reasoner and the DeLP Viewer. The former is responsible for deriving arguments from the defeasible logic program, building dialectical trees, and analyzing the defeating relationships in order to answer user queries. The latter is the visualization module and the tool presented in this article. Both modules are linked together by the dialectical explanation. This explanation is an XML file generated by the reasoner and received by our tool. Figure 2 shows an overview of the whole system.

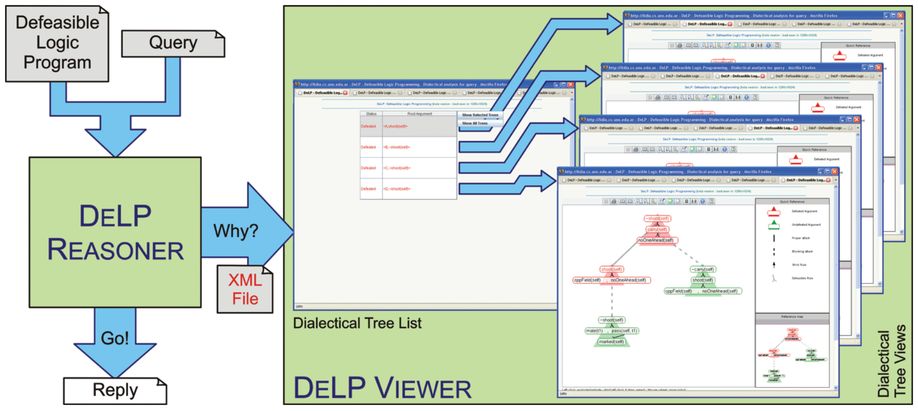


Figure 2: DeLP System overview. The main components of the system and the typical working flow performed by the user are illustrated.

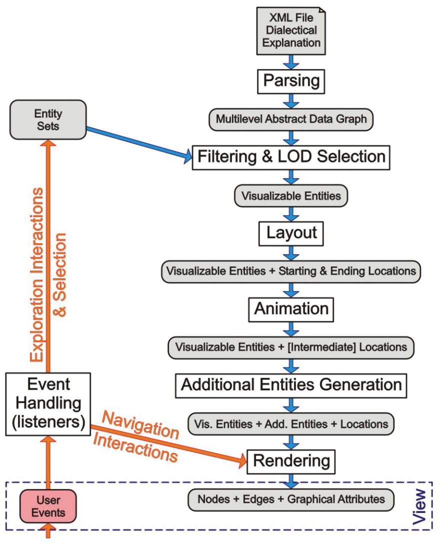
Given a defeasible logic program and a query, both specified by the user as plain text, the DeLP reasoner analyzes the logic program and derives a reply for the query. After that, the user can ask the system for examining the dialectical explanation by invoking the visualization module. Initially, the DeLP Viewer lists the dialectical trees involved in the reasoning process. The user can select which dialectical trees wants to visualize. For each selected tree, the system generates an interactive visual representation and shows it in a separate window. The details concerning each dialectical tree view are presented in the following sections.

5 Visualization Design

Visualization is concerned with two main aspects: building the visual representation and providing useful interactions [3]. However, other issues had to be considered to reach fulfilling results. Along the following subsections we present and discuss them.

5.1 The DeLP Viewer Visualization Pipeline

The backbone of our application is the DeLP Viewer visualization pipeline (see Fig. 3). For each dialectical tree to be shown, a new instance of this pipeline is configured and executed. The division of the process into stages results in a better design because the tasks throughout the pipeline can be addressed independently.



XML File Dialectical Explanation
Parsing
Multilevel Abstract Data Graph
Entity Sets
Filtering & LOD Selection
Visualizable Entities
Layout
Visualizable Entities + Starting & Ending Locations
Animation
Visualizable Entities + [Intermediate] Locations
Additional Entities Generation
Vis. Entities + Add. Entities + Locations
Rendering
Nodes + Edges + Graphical Attributes
Exploration Interactions & Selection
Event Handling (listeners)
Navigation Interactions
User Events
View

Figure 3: The DeLP Viewer visualization pipeline. Many details were simplified due to illustration purposes. The figure shows the stages involved in data transformation from the explanation parsing to the view creation, and the modules implementing user interaction feedback.

The pipeline begins parsing the dialectical explanation XML file given by the reasoner. As the parsing result, a multilevel graph representation of the dialectical tree is built. The next stage in the pipeline filters entities that will not be visualized (e.g., the content of arguments that are not completely shown). After that, the layout algorithm defines starting and ending locations for each visible entity. Then, the animation stage interpolates these locations to get intermediate positions for the entities. After the animation stage, additional entities are generated, i.e., entities whose location is not calculated by the layout algorithm. Finally, in the rendering stage, the entities are drawn and graphical attributes like color are applied.

User interactions are originated in the view. Some interactions are solved locally by only re-executing the rendering phase, and some others involve the manipulation

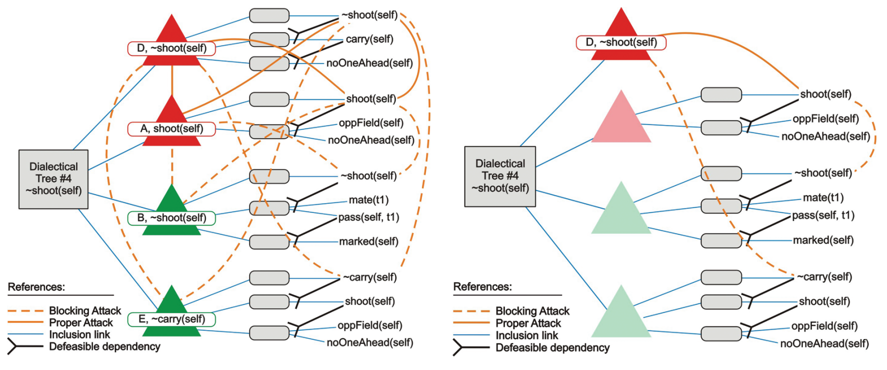


Figure 4: The dialectical tree multilevel data structure with its four nesting levels. In (a) an instance of the underlying data structure for the dialectical tree shown in Fig. 6d can be seen. Apart from the inclusion relationships (drawn in blue), the edges for argument attacks and logic dependencies among rules are represented in this figure. In (b) only the visually mapped entities are shown. When an argument implosion is performed (as in the case of the argument D for $\sim shoot(self)$), its content loses their mapping.

of the entity sets establishing which arguments are completely shown, which ones are selected, etc.

5.2 The Dialectical Tree Multilevel Data Structure

One of the main benefits of our tool is the visual representation of the underlying logic structure. The data structure representing dialectical trees is a directed acyclic graph (DAG) that has four nesting levels. First, we have the whole dialectical tree which includes, in a second level, the arguments. In the third level there are nodes representing rule heads and bodies, and, in the last level, we have the literals. An instance of this structure for the dialectical tree shown in Fig. 6d can be seen in Fig. 4.

This structure represents a tree (i.e., the dialectical tree) whose nodes (i.e., arguments) are also trees. The arguments have a tree-like structure given by the underlying logic. Additionally, there is a third hierarchy: the inclusion tree itself. This tree constitutes a clustering scheme in which arguments can be thought as clusters that group rule heads and bodies.

5.3 The Visual Representation

The visual representation involves the definition of the spatial substrate, the visual elements used to represent arguments, relationships, logic dependencies, etc; and the graphical attributes of those elements. This implies the setting up of mappings from the abstract data to the graphical space. Additionally, our tool exploits well known representation conventions in DeLP to help users to feel familiarized faster with the visualization.

DeLP Viewer uses a node-link visual representation to show each dialectical tree. Literals, rule bodies and heads, and arguments are represented by nodes. Attack relationships and weak and strict logic dependencies are represented by edges. Only nodes are positioned by our layout algorithm. Edges are straight lines that are placed using the nodes locations.

As was stated previously, a key aspect in our visualization is the underlying logical structure of the argumentation. This structure must be present in the visual representation to enable its reconstruction in the user mental map. Our approach places nodes in such a way that dependencies and relationships become evident and node overlap is avoided by means of a bounding box-based scheme. Additionally, the algorithm keeps the elements of each level aligned to enforce their position inside the hierarchy. These considerations facilitate the user insight. The DeLP Viewer visualization layout can be seen in Fig. 5.

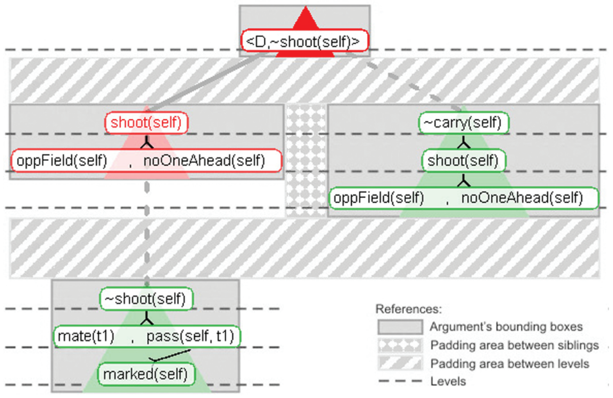

<D,~shoot(self)>
shoot(self)
oppField(self) , noOneAhead(self)
~carry(self)
shoot(self)
oppField(self) , noOneAhead(self)
~shoot(self)
mate(t1) , pass(self, t1)
marked(self)
References:
Argument's bounding boxes
Padding area between siblings
Padding area between levels
Levels

Figure 5: The DeLP Viewer dialectical tree layout. This is the dialectical tree observed in Fig. 6d but here the root argument is imploded and, consequently, its content is not visible. Several additions were made to this screen capture for illustrating layout aspects.

Dialectical trees do not have any associated visual element because they are represented by more than one entity. Arguments are represented with triangles because the triangle enforces the idea of a tree-shaped hierarchy behind each argument. Rounded rectangles are used for representing rule heads and bodies and literal representation is performed through textual labels. DeLP Viewer shows defeated arguments in red and undefeated arguments in green according to the DeLP conventions.

Edges are represented by straight lines and rendered using the DeLP notation. To avoid confusion when many attacks are produced over the same argument, the attacked literals are represented with different color. Additionally, alpha blending is used in order to make exploded arguments translucent enabling the visualization of the attacking edges under them. All these visual encodings can be appreciated in Fig. 6.

5.4 Interactions

Gaining insight about the reasoning process from a static representation is a difficult task. Interactions are needed to tune up the visualization and obtain richer perspectives that result in a better understanding of that process.

DeLP Viewer provides typical visualization interactions like *geometric zooming and scrolling*. An overview display is used to keep the user in context. Additionally, *argument selection* provides users with the ability to mark arguments to perform subsequent operations over them. Finally, DeLP Viewer provides two forms of semantic zooming. With *argument explosion/implosion*, the user can toggle between two argument representations: one shows only the literal supported by the argument and the other the entire argument content. Those different levels of detail are shown in Fig. 5 and Fig. 6d respectively. The other form of semantic zooming is the *literal label switching*. For each argument, the user can toggle between full length labels to get the whole information or abbreviated ones to reduce the visual complexity.

6 An Application Example

In this section we illustrate the practical usefulness of our proposal in a concrete application scenario. We consider a DeLP program representing the knowledge base (KB) for a robotic soccer player agent (more details can be found in [11]). The soccer agent uses DeLP in its decision making process.

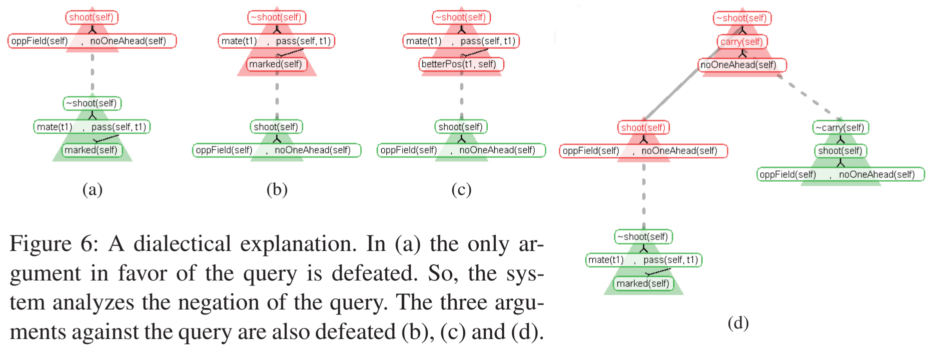
```
shoot(P) —< oppField(P), noOneAhead(P).          carry(P) —< noOneAhead(P).
~shoot(P) —< mate(M), pass(P,M).                 ~carry(P) —< shoot(P).
~shoot(P) —< carry(P).                           ~carry(P) —< mate(M), pass(P,M).

pass(P,M) —< marked(P).                          marked(t1) ← true.
pass(P,M) —< betterPos(M,P).                     marked(self) ← true.
~pass(P,M) —< shoot(P).                          oppField(self) ← true.
~pass(P,M) —< carry(P).                          noOneAhead(self) ← true.
~pass(P,M) —< marked(M).                         mate(t1) ← true.
~pass(P,M) —< someoneBetween(P,M).               someoneBetween(self,t1) ← true.
                                                 betterPos(t1,self) ← true.
```

Program 1: DeLP program representing the KB of a robotic soccer player agent.

Program 1 shows the KB of the soccer agent. In this case, defeasible rules were used to model the decision logic of the agent, and strict rules were used to represent the current situation of the agent, his teammate $t1$, and their opponents. The agent can find arguments for shoot, pass or carry the ball, and these arguments can engage in contradiction in ways that are not perceivable at first sight.

The main purpose of our tool is to show the justification for a given query. To illustrate this point, we will consider the query $shoot(self)$ for the Program 1. From the examination of program rules it is difficult to venture an answer without a deeper analysis in which the dialectical explanation is rebuilt in some way. Must or must not the soccer agent shoot? If you ask to the DeLP System, it answers '`undecided`'. The dialectical explanation for such a query and the Program 1 can be appreciated in Fig. 6.

After a quick examination of the dialectical explanation, the user can easily realize about how the argumentation proceeds, inducing dependencies and drawing conclusions. In the example, the reasons for and against shooting the ball are exposed. In Fig. 6a, a counter-argument for shooting the ball is found and it relies in the fact that our agent should pass the ball because he is being marked. In Fig. 6b and 6c the two arguments against shooting the ball are defeated because our agent is in the opposite field and there is no opponent ahead. Finally, in Fig. 6d, the argument against shooting that relies in carrying the ball is defeated by an argument against carrying the ball that suggests shooting instead. So, no argument can be built without at least one argument in contradiction, explaining, in that way, the reasoner's answer. Understand why this happens is not a trivial task from the program rules.



shoot(self)
oppField(self) , noOneAhead(self)
~shoot(self)
mate(t1) , pass(self, t1)
marked(self)
(a)
~shoot(self)
mate(t1) , pass(self, t1)
marked(self)
shoot(self)
oppField(self) , noOneAhead(self)
(b)
~shoot(self)
mate(t1) , pass(self, t1)
betterPos(t1, self)
shoot(self)
oppField(self) , noOneAhead(self)
(c)
~shoot(self)
carry(self)
noOneAhead(self)
shoot(self)
oppField(self) , noOneAhead(self)
~carry(self)
shoot(self)
oppField(self) , noOneAhead(self)
~shoot(self)
mate(t1) , pass(self, t1)
marked(self)
(d)

Figure 6: A dialectical explanation. In (a) the only argument in favor of the query is defeated. So, the system analyzes the negation of the query. The three arguments against the query are also defeated (b), (c) and (d). Hence, the system answers `undecided`.

Dialectical trees allow DeLP programmers to quickly analyze the foundations on which each argumentation line relies. The automatic building of such visual representations reduce the time and effort required in the dialectical explanation analysis, and enables the possibility of introducing modifications into the DeLP program and observe their effects in real time. In this sense, our tool has proven to be valuable in performing these application domain specific and typical tasks.

7 Conclusions and Future Work

We have presented DeLP Viewer: a visualization tool intended to help users to understand dialectical explanations. By contrast with other argumentation visualization tools, DeLP Viewer represents not only the dialectical argumentation, but also the underlying logical structure present in DeLP. This is a key aspect in order that users can gain insight about the relationships among arguments that justify the replies of the DeLP reasoner. The nested representation used is a novel aspect in Argumentation Visualization.

Major design decisions have been discussed along the paper. An important topic to emphasize is the use of DeLP drawing conventions to exploit the user's previous knowledge and improve the visualization experience.

As future work, we aim to include interactions intended to deal with some scalability issues of our tool, and to provide linking and brushing among dialectical trees to provide better visual queries. Additionally, we expect to perform some user evaluation of our tool. We need to obtain a quantitative measure of both the effectiveness of DeLP Viewer and the rate in which our tool improves the user experience.

References

[1] Rationale by Austhink. http://rationale.austhink.com/, Last visited on July 2009.

[2] M. Brayshaw, M. Eisenstadt, and J. Paine. *The Transparent Prolog Machine*. Intellect Books, 1991.

[3] S. K. Card, J. Mackinlay, and B. Shneiderman, editors. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, 1999.

[4] J. Domingue. Visualizing knowledge based systems. In J. Stasko, J. Domingue, M. Brown, and B. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 16. MIT Press, January 1998.

[5] M. Eisenstadt and M. Brayshaw. The truth about prolog execution. In J. Stasko, J. Domingue, M. Brown, and B. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 15. MIT Press, 1998.

[6] A. García, C. Chesñevar, N. Rotstein, and G. Simari. An abstract presentation of dialectical explanations in defeasible argumentation. *First international workshop on Argumentation and Non-Monotonic Reasoning (ArgNMR 07)*, pages 17–32, May 2007.

[7] A. García and G. Simari. Defeasible logic programming: An argumentative approach. *Theory and Practice of Logic Programming (TPLP)*, Vol 4(1):95–138, 2004.

[8] P. A. Kirschner, S. J. Buckingham Shum, and C. S. Carr, editors. *Visualizing argumentation: software tools for collaborative and educational sense-making*. Springer-Verlag, London, UK, 2003.

[9] C. A. Reed and G. W. Rowe. Araucaria: Software for argument analysis, diagramming and representation. *International Journal on Artificial Intelligence Tools (IJAIT)*, Vol. 13(4):961–979, 2004.

[10] B. Rolf and C. Magnusson. Developing the art of argumentation. a software approach. In *Proceedings of the 5th International Conference on Argumentation*, 2002.

[11] N. Rotstein, A. García, and G. Simari. Reasoning from desires to intentions: A dialectical framework. *Twenty-Second AAAI Conference on Artificial Intelligence (AAAI-07)*, July 2007.

[12] S. W. van den Braak, G. A. W. Vreeswijk, and H. Prakken. Avers: an argument visualization tool for representing stories about evidence. In *ICAIL '07: Proceedings of the 11th International Conference on Artificial Intelligence and Law*, pages 11–15. MIT Press, 2007.

[13] D. Suthers, A. Weiner, J. Connelly, and M. Paolucci. Belvedere: Engaging students in critical discussion of science and public policy issues. In *AI-Ed 95: Proceedings of the 7th World Conference on Artificial Intelligence in Education*, 1995.

[14] T. J. van Gelder. Argument mapping with reason!able. *The American Philosophical Association Newsletter on Philosophy and Computers*, Vol 2(1):85–90, 2002.