Constructing Legal Arguments with Rules in the Legal Knowledge Interchange Format (LKIF)

Thomas F. Gordon

Fraunhofer FOKUS
Berlin, Germany
thomas.gordon@fokus.fraunhofer.de

Abstract. The Legal Knowledge Interchange Format (LKIF), being developed in the European ESTRELLA project, defines a knowledge representation language for arguments, rules, ontologies, and cases in XML. In this article, the syntax and argumentation-theoretic semantics of the LKIF rule language is presented and illustrated with an example based on German family law. This example is then applied to show how LKIF rules can be used with the Carneades argumentation system to construct, evaluate and visualize arguments about a legal case.

1 Introduction

The Legal Knowledge Interchange Format (LKIF) is an XML application being developed in the European ESTRELLA project (IST-4-027655) with the goal of establishing an open, vendor-neutral standard for exchanging formal models of the law, suitable for use in legal knowledge-based systems. By the end of the ESTRELLA project, LKIF will enable four kinds of legal knowledge to be encoded in XML: arguments, rules, ontologies and cases. The focus of the present paper is the LKIF language for modeling legal rules.

Legal rules express norms and policy. These are not only norms or policies about how to act, but also about how to reason about the law when planning actions or determining the legal consequences of actions. For example, the definition of murder as the "unlawful killing of a human being with malice aforethought" expresses both the legal (and moral) norm against the intentional killing of another human being and the reasoning policy creating a presumption that an accused person has committed murder once it has been proven that he killed another human being intentionally. Such presumptions are not sufficient for proving guilt. Rather, a guilty verdict would be legally correct only at the end of a properly conducted legal trial. If during this trial the defendant is able to produce evidence of the killing having been done in self-defense, for example, a guilty verdict would be correct only if the prosecution meets its burden of persuading the court or jury that the killing was in fact not done in self-defense.

Thus, the semantics of the LKIF rules is based not on the model theory of first-order logic, but rather on the dialectical and argumentation-theoretic

P. Casanovas et al. (Eds.): Computable Models of the Law, LNAI 4884, pp. 162–184, 2008.

approach to semantics articulated by Ron Loui in "Process and Policy: Resource-Bounded Non-Demonstrative Reasoning" [25]. Essentially, legal rules are interpreted as policies for reasoning in resource-limited, decision-making processes. In argumentation theory, such reasoning policies are viewed as inference rules for presumptive reasoning, called *argumentation schemes* [37]. Arguments are instances of argumentation schemes, constructed by substituting variables of a scheme with terms of the object language. An *argument graph* is constructed from a set of arguments. A set of argumentation schemes defines a search space over argument graphs. Reasoning with argumentation schemes can be viewed as heuristic search in this space, looking for argument graphs in which some disputed claim is acceptable or not given the arguments in the graph. In dialogues, the parties take turns searching this space, looking for counterarguments. Turn-taking, termination conditions, resource limitations and other procedural parameters are determined by the applicable rules of the legal proceeding, i.e. by the argumentation protocol for the particular type of dialogue.

The rest of this article is organized as follows. First, we provide an informal introduction and overview of LKIF rules, including some examples. This is followed by the formal definition of its abstract syntax. Then we define the semantics of the rule language, by mapping rules to argumentation schemes, using the Carneades model of argument [19]. LKIF rules is then illustrated with a more lengthy legal example about support obligations, based on German family law. This example is also used to illustrate an XML syntax for interchanging rule bases in LKIF, presented in the following section. Then we show how the Carneades argumentation system can use LKIF rule bases to construct and visualize arguments about cases. Finally, we conclude with a brief dicussion of related work, summarize the main results and suggest some ideas for future work.

2 Informal Overview

For simplicity and readability, we will be using a concrete syntax based on Lisp *s-expressions* to represent rules. Variables will be represented as symbols beginning with a question mark, e.g. `?x` or `?y`. Other symbols, as well as numbers and strings, represent constants, e.g. `contract`, `23.1`, or `"Jane Doe"`.

An *atomic sentence* is a simple declarative sentence containing no logical operators (negation, conjunction or disjunction). For example, the sentence "The mother of Caroline is Ines." can be represented as `(mother Caroline Ines)`.

If `P` is an atomic sentence, then `(not P)` is a *negated atomic sentence*. Sentences which are either atomic sentences or negated atomic sentences are called *literals*. The *complement* of the literal `P` is `(not P)`, and the complement of `(not P)` is `P`.

Rules are *reified* in this language, with an identifier and a set of properties, enabling any kind of meta-data about rules to be represented, such as a rule's date of enactment, issuing governmental authority, legal source text, or its period of validity. We do not define these properties here. Our focus is on defining the syntax and semantics of these rules.

Rules have a *body* and a *head*. The terms 'head' and 'body' are from logic programming, where they mean the conclusions and antecedents of a rule, respectively, interpreted as Horn clauses. Unlike Horn clause logic, a rule in our system may have more than one conclusion, including, as will be explained shortly, negated conclusions.

Here is a first example, a simplified reconstruction of a rule from the Article Nine World of the Pleadings Game [17], meaning that all movable things except money are goods.

```
(rule §-9-105-h
    (if (and (movable ?c)
             (unless (money ?c)))
        (goods ?c)))
```

`§-9-105-h` is an identifier, naming the rule, which may be used as a term denoting the rule in other rules.

We use the term *condition* to cover both literals and the forms `(unless P)`, called *exceptions*, and `(assuming P)`, called *assumptions*, where `P` is a literal. The head of a rule consists of a list of *literals*. Notice that, unlike Horn clause logic, rules may have negative conclusions. Negated atomic sentences may also be used in the body of a rule, also in exceptions and assumptions. Exceptions and assumptions are allowed only within the body of rules. The example rule above illustrates the use of an exception.

Legal rules are defeasible generalizations. Showing that some exception applies is one way to defeat a rule, by *undercutting* [27] it. Intuitively, a rule applies if its conditions are met, *unless* some exception is satisfied . A party who wants to use some rule need not show that no exception applies. The burden of proof for exceptions is on those interesting in showing the rule does not apply. Assumptions on the other hand, as their name suggests, are assumed to hold until they have been called into question. After an assumption has been questioned, a party who wants to use the rule must prove the statement which had been assumed.

Another source of defeasibility is conflicting rules. Two rules conflict if one can be used to derive `P` and another `(not P)`. To resolve these conflicts, we need to be able to reason (i.e. argue) about which rule has priority. To support reasoning about rule priorities, the rule language includes a built-in predicate over rules, `prior`, where `(prior r1 r2)` means that rule `r1` has priority over rule `r2`. If two rules conflict, the arguments constructed using these rules are said to *rebut* each other, following Pollock [27].

The priority relationship on rules is not defined by the system. Rather, priority is a substantive issue to be reasoned (argued) about just like any other issue. One way to construct arguments about rule priorities is to apply the argumentation scheme for arguments from legal rules to meta-level rules, i.e. rules about rules, using information about properties of rules, such as their legal authority or date of enactment. The reification of rules and the built-in priority predicate make this possible. In knowledge bases for particular legal domains, rules can be prioritized both *extensionally*, by asserting facts about which rules have priority over which other rules, and *intensionally*, using meta-rules about priorities.

For example, assuming metadata about the enactment dates of rules has been modeled, the legal principle that later rules have priority of earlier rules, *lex posterior*, can be represented as:

```
(rule lex-posterior
    (if (and (enacted ?r1 ?d1)
             (enacted ?r2 ?d2)
             (later ?d2 ?d1))
        (prior ?r2 ?r1)))
```

Rules can be defeated in two other ways: by challenging their validity or by showing that some exclusionary condition applies. These are modeled with rules about validity and exclusion, using two further built-in predicates: `(valid <rule>)` and `(excluded <rule> <literal>)`, where `<rule>` is a constant naming the rule, not its definition. The second argument of the `excluded` predicate is a compound term representing a literal. Thus, literals can also be reified in this system.

The `valid` and `excluded` relations, like the `prior` relation, are to be defined in models of legal domains. Rules can be used for this purpose. For example, the exception in the previous example about money not being goods, even though money is movable, could have been represented as an exclusionary rule as follows:

```
(rule §-9-105-h-i
    (if (money ?c)
        (excluded §-9-105-h (goods ?c))))
```

To illustrate the use of the validity property of rules, imagine a rule which states that rules which have been repealed are no longer valid:

```
(rule repeal
    (if (repealed ?r1)
        (not (valid ?r1))))
```

This rule also exemplifies the use of negation in the conclusion (head) of this rule.

3 Syntax

This section presents a formal definition of an s-expression syntax for rules, in Extended Backus-Naur Form (EBNF)[1]. This syntax is inspired by the Common Logic Interchange Format (CLIF) for first-order predicate logic, which is part of the draft ISO Common Logic standard.[2] While inspired by CLIF, no attempt is made to make this rule language conform to Common Logic standard.[3]

[1] EBNF is specified in the ISO/IEC 14977 standard.

[2] http://philebus.tamu.edu/cl/

[3] Common Logic is a family of concrete syntaxes for first-order predicate logic, with its model-theoretic semantics and classical, monotonic entailment relation. These semantics are sufficiently different as to not make it useful to attempt to make the syntax of our rule language fully compatible with CLIF.

The syntax uses the Unicode character set. White space, delimiters, characters, symbols, quoted strings, boolean values and numbers are lexical classes, not formally defined here. For simplicity and to facilitate the development of a prototype inference engine using the Scheme programming language, we will use Scheme's lexical structure, as defined in the R6RS standard, which is based on the Unicode character set.[4]

Variable and Constant Symbols

```
variable ::= symbol
constant-symbol ::= symbol
```

Variable and constant symbols are disjunct. As mentioned in the informal overview, variables begin with a question mark character. Symbols are case-sensitive. Constant symbols may include a *prefix* denoting a *namespace*. Some mechanism for binding prefixes to namespaces is presumed, rather than being defined here. The prefix of a constant symbol is the part of the constant symbol up to the first colon. The part of the constant symbol after the colon is the local identifier, within this namespace.

Here are some example variable and constant symbols:

```
?x
?agreement
contract-1
lkif:permission
event-calculus:event
```

Term

A term is either a constant or a compound term. A constant is either a variable, constant symbol, string, number, or boolean value. A compound term consists of a constant symbol and a list of terms.

```
constant ::= variable | constant-symbol
          | string | number | boolean
term ::= constant | | '´' term |
       '(' constant-symbol term* ')'
```

Quoted terms are used, as in Lisp, to denote lists. Here are some example terms:

```
?x
contract-1
"Falkensee, Germany"
12.345
#t
(father-of John)
'(red green)
```

[4] http://www.r6rs.org/

Literal

Literals are atomic sentences or negations of atomic sentences.

```
atom ::= constant-symbol
      | '(' constant-symbol term* ')'
literal ::= atom | '(' 'not' atom ')'
```

Notice that constant symbols can be used as atomic sentences. This provides a convenient syntax for a kind of propositional logic.

The following are examples of literals:

```
liable
(initiates event1 (possesses ?p ?o))
(holds (perfected ?s ?c) ?p)
(children Ines '(Dustin Caroline))
(not (children Tom '(Sally Susan)))
(applies UCC-§-306-1 (proceeds ?s ?p))
```

Rule

Since Horn clause logic is widely known from logic programming, it might be helpful to begin the presentation of the syntax of LKIF rules by noting that it can be viewed as a generalization of the syntax of Horn clause logic, in the following ways:

1. Rules are reified with names.
2. Rules may have multiple conclusions.
3. Negated atoms are permitted in both the body and head of rules.
4. Rule bodies may include exceptions and assumptions.
5. Both disjunctions and conjunctions are supported in the bodies of rules.

Here is the formal definition of the syntax of rules:

```
condition ::= literal
            | '(' 'unless' literal ')'
            | '(' 'assuming' literal ')'

conjunction ::= condition
              | '(' 'and' condition condition+ ')'

disjunction ::= '(' 'or' conjunction conjunction+ ')'

body ::= condition | conjunction | disjunction

head ::= literal
       | '(' 'and' literal literal+ ')'
```

```
rule ::= '(' 'rule' constant-symbol
             '(' 'if' body head ')' ')'
       | '(' 'rule' constant-symbol
             literal literal* ')'
```

The second rule form is convenient for rules with empty bodies. These should not be confused with Prolog 'facts', since they are also defeasible. Conditions which are not assumptions or exceptions are called *ordinary conditions*.

Here are a few examples of rules and facts, reconstructed from the Article Nine World of the Pleadings Game [17]:

```
(rule §-9-306-3-1
    (if (and (goods ?s ?c)
             (consideration ?s ?p)
             (collateral ?si ?c)
             (collateral ?si ?p)
             (holds (perfected ?si ?c) ?e)
             (unless (applies §-9-306-3-2
                              (perfected ?si ?p))))
        (holds (perfected ?si ?p) ?e)))

(rule §-9-306-2a
    (if (and (goods ?t ?c)
             (collateral ?s ?c))
        (not (terminates ?t
                (security-interest ?s)))))

(rule F1 (not (terminates T1
                  (security-interest S1))))
(rule F2 (collateral S1 C1))
```

Reserved Symbols

The following predicate symbols have special meaning in the semantics, as explained in Section 4, and are thus reserved: `prior`, `excluded`, `valid`, and `applies`.

4 Semantics

We now proceed to define the semantics of the LKIF rules language. Due to space limitations, knowledge of the Carneades model of argument [19] is presumed. A rule denotes a *set* of argumentation schemes, one for each conclusion of the rule, all of which are subclasses of a general scheme for arguments from legal rules.[5]

[5] We do not claim that argumentation schemes can be modeled as or reduced to rules. Here we go in the other direction: each rule is mapped to a set of argumentation schemes.

Applying a rule is a matter of instantiating one of these argumentation schemes to produce a particular argument. Reasoning with rules is viewed as a process of applying these schemes to produce arguments to put forward in dialogues.

The scheme for arguments from legal rules is based on the rule language we developed for the Pleadings Game [17], but has also been influenced by Verheij's reconstruction of Reason-Based Logic in terms of argumentation schemes [36]. The scheme can be defined informally as follows:

Premises

1. r is a legal rule with ordinary conditions $a_1, \ldots, a_n$ and conclusion c.
2. Each a_i in $a_1 \ldots a_n$ is presumably true.

Conclusion. c is presumably true.

Critical Questions

1. Does some exception of r apply?
2. Is some assumption of r not met?
3. Is r a valid legal rule?
4. Does some rule excluding r apply in this case?
5. Can some rule with priority over r be applied to reach an contradictory conclusion?

Our task now is use this scheme to define the semantics of the formal language of Section 3, by mapping rules in the language to schemes for arguments in Carneades. We begin by mapping rule conditions to argument premises.

Definition 1 (Condition to Premise). *Let p be a function mapping conditions of rules to argument premises, defined as follows:*

$$p(c) = \begin{cases} c & \text{if } c \text{ is a literal} \\ \bullet s & \text{if } c \text{ is } \texttt{(assuming } s\texttt{)} \\ \circ s & \text{if } c \text{ is } \texttt{(unless } s\texttt{)} \end{cases}$$

If a conclusion of a rule is an atomic sentence, `s`, then the rule is mapped to a scheme for arguments *pro* `s`. If a conclusion of the rule is a negated atomic sentence, `(not s)`, then the rule is mapped to a scheme for arguments *con* `s`.

Definition 2 (Scheme for Arguments from Rules). *Let r be a rule, with conditions $a_1 \ldots a_n$ and conclusions $c_1 \ldots c_n$. Three premises, implicit in each rule, are made explicit here. The first, $\circ\upsilon$, where $\upsilon =$* `(not (valid` r`))`*, excepts r if it is an invalid rule. The second, $\circ\epsilon$, where $\epsilon =$* `(excluded` r c_i`)`*, excepts r if it is an* excluded *with respect to c_i by some other rule. The third, $\circ\pi$, where $\pi =$* `(priority` r_2 r`)`*, excludes r if another rule, r_2, exists of higher priority than r which is applicable and supports a contradictory conclusion.*

For each c_i in $c_1 \ldots c_n$ of r, r denotes an argumentation scheme of the following form, where d is 'pro' if c_i is an atomic sentence and 'con' if c_i is a negated atomic sentence:

$$\frac{p(a_1) \ldots p(a_n), \circ\upsilon, \circ\epsilon, \circ\pi}{d\ c_i}$$

To construct an argument from one of these argumentation schemes, the variables in the scheme need to be systematically renamed and then instantiated using a *substitution environment*, i.e. a mapping from variables to terms, constructed by unifying the conclusion of the argumentation scheme with some goal atomic statement, as in logic programming.

The `valid` and `excluded` relations used in the argumentation scheme are to be defined in the models of legal domains, as explained in Section 2. Rules can be used to define the priority relation, as in the Pleadings Game [17] and PRATOR [28]. Legal principles for resolving rule conflicts, such as *lex posterior*, can be modeled in this way, as illustrated in Section 2.

The `applies` predicate is a 'built-in', meta-level relation which cannot be defined directly in rules. It is defined as follows:

Definition 3 (Applies). *Let σ be a substitution environment and G be an argument graph. Let r be a rule and S be the set of argumentation schemes for r, with all of the variables in these schemes systematically renamed. There are two cases, for atomic literals and negated literals. The rule r* applies *to a literal* `P` *in the structure $\langle \sigma, G \rangle$, if there exists a* pro *argumentation scheme s in S, if* `P` *is atomic, or a* con *argumentation scheme, if* `P` *is negated, such that the conclusion of s is unifiable with* `P` *in σ, and every premise of s, with its variables substituted by their values in the σ,* holds *in G.*

Given a set of rules and an argument graph, this definition of the `applies` predicate enables some meta-level reasoning. It allows one to find rules which can be used to generate defensible pro and con arguments for some goal statement or to check whether a particular rule can be used to generate a defensible pro or con argument for some statement.

The semantics of negation is dialectical, not classical negation or negation-as-failure. Exceptions do not have the semantics of negation-as-failure. The closed-world assumption is not made. In Carneades, a negated sentence, `(not p)`, is acceptable just when the complement of the proof standard assigned to `p` is satisfied, where the complement of a proof standard is constructed by reversing the roles of pro and con arguments in the standard. See [19] for details.

5 A German Family Law Example

Let's now illustrate LKIF rules using a small, toy legal domain, roughly based on German family law. The question addressed is whether or not a descendent of some person, typically a child or grandchild, is obligated to pay financial support to the ancestor.

§1601 BGB (Support Obligations). Relatives in direct lineage are obligated to support each other.

```
(rule §-1601-BGB
   (if (direct-lineage ?x ?y)
       (obligated-to-support ?x ?y)))
```

§1589 BGB (Direct Lineage). A relative is in direct lineage if he is a descendent or ancestor. For example, parents, grandparents and great grandparents are in direct lineage.

```
(rule §-1589-BGB
    (if (or (ancestor ?x ?y)
            (descendent ?x ?y))
        (direct-lineage ?x ?y)))
```

§ 1589 BGB illustrates the use of disjunction in the body of a rule.

§1741 BGB (Adoption). For the purpose of determining support obligations, an adopted child is a descendent of the adopting parents.

```
(rule §-1741-BGB
    (if (adopted-by ?x ?y)
        (ancestor ?x ?y)))
```

§1590 BGB (Relatives by Marriage). There is no obligation to support the relatives of a spouse (husband or wife), such as a mother-in-law or father-in-law.

```
(rule §-1590-BGB
    (if (relative-of-spouse ?x ?y)
        (not (obligated-to-support ?x ?y))))
```

§ 1590 BGB illustrates the use of negation in the head of a rule.

§1602 BGB (Neediness). Only needy persons are entitled to support by family members. A person is needy only if unable to support himself.

```
(rule §-1602a-BGB
    (if (not (needy ?x))
        (not (obligated-to-support ?y ?x))))

(rule §-1602b-BGB
    (if (not (able-to-support-himself ?x))
        (needy ?x)))

(rule §-1602c-BGB
    (if (able-to-support-himself ?x)
        (not (needy ?x))))
```

In § 1602 we see examples of negation in both the head and body. This example also illustrates that it is not always possible to represent a section of a piece of legislation as a single LKIF rule. Thus, although LKIF brings us closer to the ideal of "isomorphic modeling", this goal remains illusive, at least if one takes the view that each section of legal code always expresses a single rule.

§1603 BGB (Capacity to Provide Support). A person is not obligated to support relatives if he does not have the capacity to support others, taking into consideration his income and assets as well as his own reasonable living expenses.

```
(rule §-1603-BGB
    (if (not (capacity-to-provide-support ?x))
        (not (obligated-to-support ?x ?y))))
```

§1611a BGB (Neediness Caused By Immoral Behavior). A needy person is not entitled to support from family members if his neediness was caused by his own immoral behavior, such as gambling, alcoholism, drug abuse or an aversion to work.

```
(rule §-1611a-BGB
    (if (neediness-caused-by-own-immoral-behavior ?x)
        (excluded §-1601-BGB (obligated-to-support ?y ?x))))
```

Here we have interpreted § 1611a BGB to be an exclusionary rule. If one instead takes the view that it states conditions under which there is no obligation to provide support, independent of the general obligation to provide support stated in § 1601 BGB, then the following LKIF rule would be a more faithful representation:

```
(rule §-1611a-BGB
    (if (neediness-caused-by-own-immoral-behavior ?x)
        (not (obligated-to-support ?y ?x))))
```

§91 BSHG (Undue Hardship). A person is not entitled to support relatives if this would cause him undue hardship.

```
(rule §-91-BSHG
    (if (undue-hardship ?x (obligated-to-support ?x ?y))
        (excluded §-1601-BGB (obligated-to-support ?x ?y)))))
```

As with § 1611a BGB, we have interpreted § 91 BSHG as an exclusionary rule, mainly to illustrate how statements are reified in LKIF and can be quoted in other statements. Here the statement `(obligated-to-support ?x ?y)` is quoted in the statement `(excluded s1601-BGB (obligated-to-support ?x ?y))`.

6 XML Syntax

The Legal Knowledge Interchange Format (LKIF) defines two ways to representing arguments, rules, ontologies and cases in XML. One uses OWL, the Ontology Web Language [26], to define concepts and relations for the structure of arguments, rules and cases. Particular arguments, rules and cases are represented in

OWL as instances of these classes. LKIF ontologies are defined directly in OWL. This approach offers the advantage of uniformity. An entire knowledge base can be represented using a single, widely supported existing standard, OWL, and be developed, maintained and processed using existing OWL editors and other tools.

As it turns out, however, rules and arguments cannot be conveniently written or maintained using generic OWL editors, such as Protege [14] or TopBraid Composer [35], at least not without first extending them with 'plug-ins' for special purpose editors, along the lines of the Protege plug-in for the Semantic Web Rule Language [24].

Moreover, OWL is not, strictly speaking, an XML format. Rather, OWL is defined at a more abstract level. OWL documents can be 'serialized' using a variety of concrete syntaxes. Some of these are XML-based, for example using RDF/XML [7]. Other serializations of OWL, some based on the Notation 3 [8] language, aim to be compact and more readable and thus do not use XML. For this reason, implementing a translator for LKIF documents encoded in OWL requires the document to first be preprocessed into some canonical concrete syntax, using for example Jena [23], a Java library for the Semantic Web.

For these reasons, LKIF offers an alternative, more compact, XML syntax. This syntax is defined using the XML Schema Definition Language [13]. An eqivalent definition of the grammar using Relax NG [10], an ISO standard schema definition language (ISO/IEC 19757), is also available. One advantage of Relax NG is that it offers a compact, readable language for schema definitions, in addition to an XML language. Here is the Relax NG version of the compact syntax of LKIF Rules:

```
start = element lkif  Statement*, Rule*, ArgumentGraph*

Rule = element rule
    attribute id  xsd:ID ?,
    attribute strict  "no" | "yes" ?,
    (Literal+ | Implies)

Literal = Statement | Not
Statement = element s
    attribute id  xsd:ID ?,
    attribute src  xsd:anyURI | xsd:string ?,
    ((text* &  Statement*)*)?
Not = element not  Statement
Implies = (Head, Body) | (Body, Head)
Head = element head  Literal+
Body = element body  Or | Condition+
Or = element or  (Condition | And)+
And = element and  Condition+
Condition = Literal
```

```
        | element if  attribute role  text ?, Literal
        | element unless  attribute role  text ?, Literal
        | element assuming  attribute role  text ?, Literal
```

The specification of the compact syntax for argument graphs has been omitted, since the focus of this article is LKIF's rule language.

The German family law example of the previous section can be represented in XML using the compact syntax as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<lkif>
  <rule id="s1601-BGB">
    <body><s>direct-lineage ?x ?y</s></body>
    <head><s>obligated-to-support ?x ?y</s></head>
  </rule>

  <rule id="s1589a-BGB">
    <body><s>ancestor ?x ?y</s></body>
    <head><s>direct-lineage ?x ?y</s></head>
  </rule>

  <rule id="s1589b-BGB">
    <body><s>descendent ?x ?y</s></body>
    <head><s>direct-lineage ?x ?y</s></head>
  </rule>

  <rule id="s1741-BGB">
    <body><s>adopted-by ?x ?y</s></body>
    <head><s>ancestor ?x ?y</s></head>
  </rule>

  <rule id="s1590-BGB">
    <body><s>relative-of-spouse ?x ?y</s></body>
    <head><not><s>obligated-to-support ?x ?y</s></not></head>
   </rule>

  <rule id="s1602a-BGB">
    <body><not><s>needy ?x</s></not></body>
    <head><not><s>obligated-to-support ?y ?x</s></not></head>
   </rule>

  <rule id="s1602b-BGB">
    <body><not><s>able-to-support-himself ?x</s></not></body>
    <head><s>needy ?x</s></head>
  </rule>
```

```
  <rule id="s1602c-BGB">
    <body><s>able-to-support-himself ?x</s></body>
    <head><not><s>needy ?x</s></not></head>
  </rule>

  <rule id="s1603-BGB">
    <body><not><s>capacity-to-provide-support ?x</s></not></body>
    <head><not><s>obligated-to-support ?x ?y</s></not></head>
  </rule>

  <rule id="s1611a-BGB">
    <body>
      <s>neediness-caused-by-own-immoral-behavior ?x</s>
    </body>
    <head>
      <s>excluded s1601-BGB <s>obligated-to-support ?y ?x</s></s>
    </head>
  </rule>

  <rule id="s91-BSHG">
    <body>
      <s>undue-hardship ?x <s>obligated-to-support ?x ?y</s></s>
    </body>
    <head>
      <s>excluded s1601-BGB <s>obligated-to-support ?x ?y</s></s>
    </head>
  </rule>
</lkif>
```

7 Reasoning with LKIF Rules Using Carneades

'Carneades' is the name of both a compuational model of argumentation [19] and an implementation of this model in PLT Scheme [30]. The ESTRELLA Reference

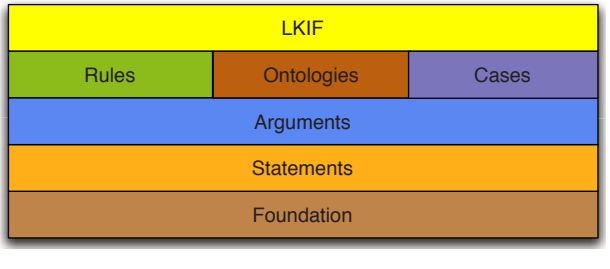

Fig. 1. Module Layers

Inference Engine for LKIF rules is being built using this implementation of Carneades.

This ESTRELLA platform, of which the LKIF rules inference engine is a part, has the layered architecture shown in Figure 1

Each layer consists of one more modules, where a module may make use of the services of another module in the same layer or any layer below it in the diagram. Conversely, no module depends on the services of any module in some higher layer.

Since the higher layers build upon the lower layers, we will describe the lowest layers first:

Foundation. The foundation layer consists of modules for configuring the system for a particular installation (`config`), managing possibly infinite sequences of data generated lazily (`stream`), and for heuristically searching problem spaces (`search`).

Statements. The statement layer provides a module for comparing and decomposing statements (`statement`), abstracting away syntactic details which are irrelevant for the higher layers, and a module implementing a unification algorithm (`unify`), needed for implementing inference engines for logics with variables ranging over compound terms, such as first-order logic and LKIF rules.

Arguments. The argument layer provides modules for constructing, evaluating and visualizing argument graphs, also called 'inference graphs' (`argument`, `argument-diagram`). It also provides modules (`argument-state`, `argument-search`) for applying argumentation schemes to search heuristically for argument graphs in which some goal statement is acceptable (i.e. presumably true) or not acceptable. An `argument-builtins` module provides an argument generator for common goal statements about arithmetic, strings, lists, dates and so on.

Rules. The rule layer implements LKIF rules. It provides a `rule` module for representing defeasible legal rules and generating arguments from sets of rules.

Ontologies. The ontologies layer provides a module for defining and reasoning with concepts, using Description Logic [4].

Cases. The cases layer provides a module for representing legal precedents and constructing arguments from these precedents using argumentation schemes for case-based reasoning.

LKIF. The Legal Knowledge Interchange Format (LKIF) layer provides a module for importing and exporting arguments, rules, ontologies and cases in XML.

Ontologies are represented in LKIF using the OWL Web Ontology Language [11]. The ESTRELLA module for reasoning with ontologies is still being designed. It may communicate with an external description logic reasoner, for example via the DIG interface [6], or translate ontologies into LKIF rules, using the description logic programming intersection of description logic and Horn clause logic [20]. However, since LKIF rules is more expressive than Horn clause logic, it may be possible to translate a larger subset of description logic into LKIF rules.

A first prototype of an implementation of case-based argumentation schemes, based on a reconstruction [38] of schemes modeled by HYPO [3] and CATO [1] has been completed and is currently being evaluated.

Some people have expressed surprise at the ordering of these layers, perhaps because of familiarity with Berners-Lee's vision of the Semantic Web [9], which has a similar architecture, consisting of the following layers, from bottom to top: Unicode and URI, XML, RDF, ontology, logic, proof and trust. Statements are expressed using the RDF layer. Rules are represented in the logic layer. One difference between these architectures is that the proof layer, which seems closely related to argument, is above the logic layer where rules reside in Berners-Lee's model, whereas rules are built on top of the argument layer in our system. Another difference is that ontologies form a foundation for logic and proof in Berners-Lee's model, whereas ontologies, rules and cases are all at the same layer in our system, as they are all interpreted as knowledge representation formalisms from which arguments can be constructed, using argumentation schemes appropriate for each type of knowledge.

We close this section with an example showing how to use Carneades to load the Germany family law example rule base, ask a query, and visualize the result. To simplify the example, let's suppose the rule base has been extended with rules defining 'direct-lineage' in terms of common-sense family relations, along with some facts about a case, for example that Gloria is needy and an ancestor of Dustin, but Dustin does not have the capacity to provide support. We will omit the definitions of family relations (such as ancestor, descendant, parent, grandparent, sibling, and relative) to keep this short. The facts of the case can be represented in LKIF rules as follows.

```
(rule* facts
  (ancestor Dustin Tom)
  (ancestor Tom Gloria)
  (needy Gloria)
  (not (capacity-to-provide-support Dustin)))
```

Let's suppose the XML file for the rule base is stored in a file named `"family-support.xml"`. This file can be imported to create a rule base as follows:

```
(define family-support
   (add-rules empty-rulebase
              (import "family-support.xml")))
```

This code defines `family-support` to be the rule base created by importing the `"family-support.xml"` LKIF file. Now we can pose a query, about whether Dustin is obligated to support his grandmother, Gloria, as follows:

```
> (define s1 (initial-state '(obligated-to-support Dustin Gloria)
                            default-context))
> (define g1 (generate-arguments-from-rules family-support null))
> (define r (make-resource 50))
```

```
> (define results (find-best-arguments depth-first r 1 s1 (list g1
   builtins)))
> (define s2 (stream-car results))
> (view* (state-arguments s2)
         (state-context s2)
         (state-substitutions s2) #t)
```

We begin by defining a problem space. The first command defines the root, initial state of the problem state. The query, `(obligated-to-support Dustin Gloria)`, is part of this initial state. The next command defines the transitions available between states in the search space. These transitions are induced by the rules available in the `family-support` rule base. Since the search space can be infinite, we use resources to limit the amount of searching done and assure the search process terminates. The `(make-resource 50)` constructs a resource with 50 units. Each state visited during the search for a solution consumes one unit of this resource. The `find-best-arguments` command in this example looks for arguments for Dustin being obligated to support Gloria, in this case using a (resource-limited) depth-first search strategy. A few others search strategies are also available, including breadth-first, and iterative-deepening. A stream of solution states is returned, where a stream is conceptually a sequence of states, where each member of the sequence is computed as needed. Thus, to backtrack and search for further solutions, one only needs to access subsequent members of the stream. If the search process fails, finding no state satisfying the query, the resulting stream will be empty. Each state in the search space contains an argument graph. The goal of the search, using the `find-best-arguments` command, is to find the best arguments pro and con the statement in the query, given the resources supplied and the number of turns to alternate between the roles of proponent and opponent of the statement. That is, in the terminology of the Carneades model of argument, we are interested in finding argument graphs which provide sufficient grounds, or reasons, for 'accepting' the statement of the query, presumptively, as true, when taking the perpective of the proponent, and finding extensions of these argument graphs which succeed in countering these arguments, making the statement of the query no longer acceptable, when taking the perspective of the opponent.

The `(view* ...)` command displays a diagram of an argument graph, using GraphView [12]. Figure 2 shows the argument graph found first by the `find-best-arguments` command above.

One way to look for a counterargument is to repeat the `find-best-arguments` command, but this time with 2 turns. In the second turn the system takes the perspective of the opponent.

```
> (define results
      (find-best-arguments depth-first r 2 s1
          (list g1 builtins)))
```

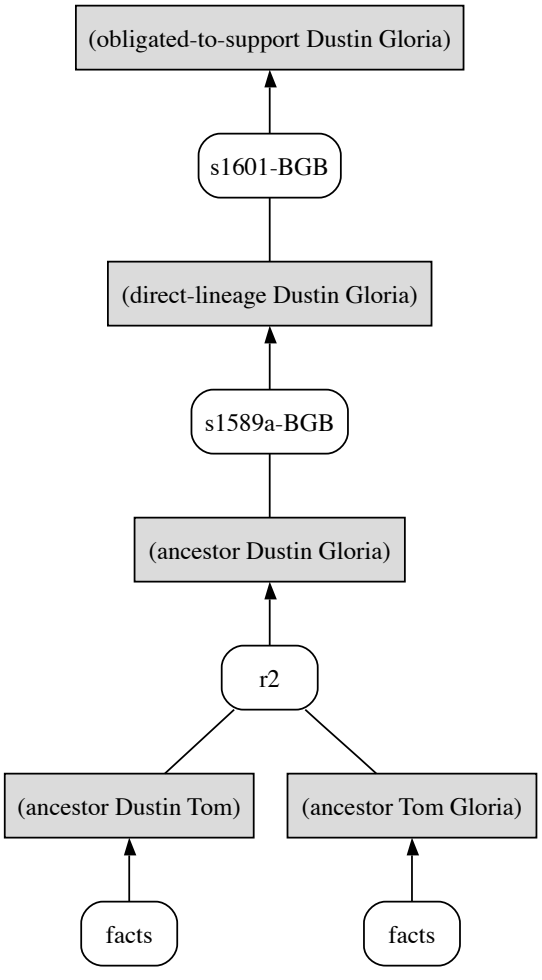

Fig. 2. An argument pro the obligation to support

In this example, a counterargument was found, as shown in Figure 3. Since `find-best-arguments` returns the best arguments for both sides, it would have returned the first argument graph again, shown in Figure 2, had it been unable to find a counterargument to this argument on the second turn.

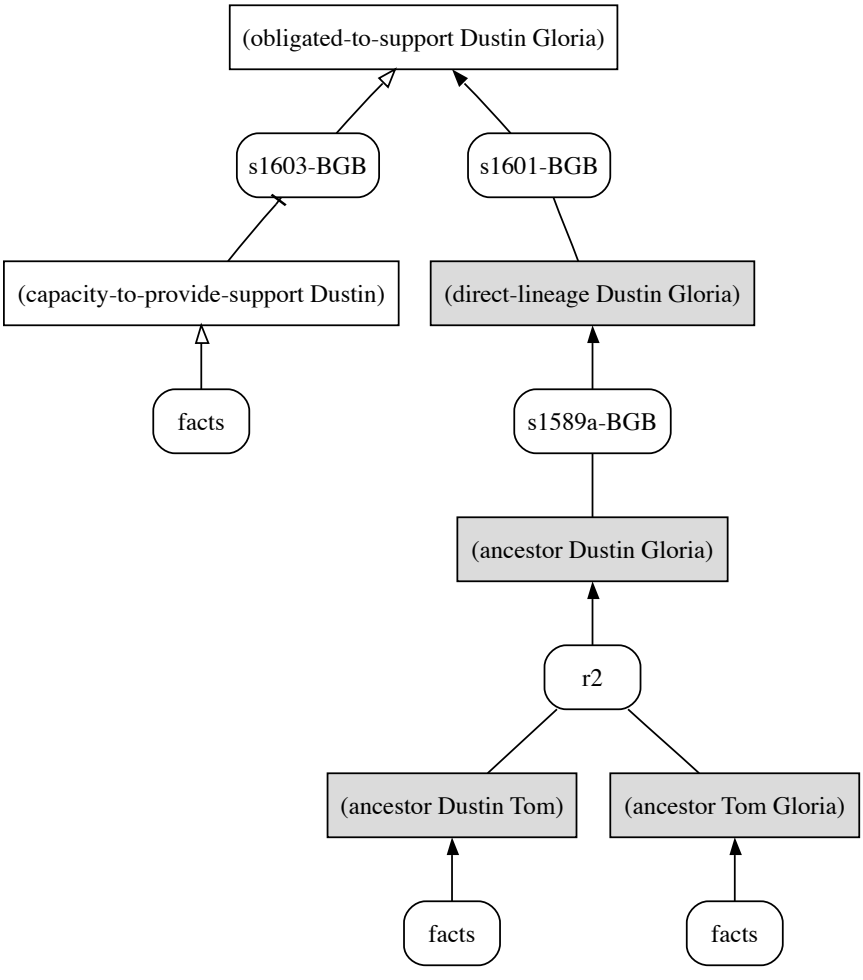

Fig. 3. A counterargument

8 Discussion

LKIF rules builds on the results of about 20 years of research in Artificial Intelligence and Law. Edwina Rissland, Kevin Ashley and Ronald Loui published a good summary of the field of Artificial Intelligence and Law, as of 2003, in a special issue of the Artificial Intelligence Journal [32]. A recent treatise on AI and Law is Giovanni Sartor's "Legal Reasoning: A Cognitive Approach to the Law" [34].

A major lesson from research on Artificial Intelligence and Law is that legal reasoning cannot be viewed, in general, as the application of some deductive logic, such as first-order predicate logic, to some theory of the facts and relevant legal domain. In fact, no one in the field ever seriously took the position that legal reasoning in its entirety could be viewed this way, although some critics

have misunderstood or misrepresented the field by assuming this to be the case. As pointed out by Rissland et al. [32]:

> Contrary to some popular notions, law is *not* a matter of simply applying rules to facts via *modus ponens*, for instance, to arrive at a conclusion. *Mechanical jurisprudence*, as this model has been called, is somewhat of a strawman. It was soundly rejected by rule skeptics like the realists. As Gardner puts it, law is more "rule-guided" than "rule-governed."

The reference to Gardner here, is to Anne Gardner's thesis "An Artificial Intelligence Approach to Legal Reasoning" [15], one of the first books to be published in the field. Legal reasoning is not only deductive, because legal concepts cannot be defined by necessary and sufficient conditions. Better, one *can* define legal concepts this way, but such definitions are only hypotheses or theories which will not be blindly or "mechanically" followed, using deduction, when one tries to apply these concepts to decide legal issues in concrete cases. Legal concepts are, as the legal philosopher H.L.A. Hart put its, "open-textured" [22]. Whether or not a legal concept applies in a particular case requires the interpretation, or reinterpretation of the legal sources, such as statutes and case law, in light of such things as the history of prior precedent cases, the intention of the legislature, public policy, and evolving social values.

The process of determining whether the facts of a case can be "subsumed" under some legal concept is one of argumentation. Legal argumentation is a dialogue, guided by procedural norms, called "protocols". Which protocol applies depends on the particular type of dialogue and the task at hand.

Although argumentation has always been at the heart of work on modeling legal reasoning in the field of AI and Law, it wasn't until two papers on computational models of legal argumentation in a special issue of the International Journal of Man-Machine Studies on AI and Law [16,33] that argumentation became a hot topic in AI and Law and efforts began in earnest to use argumentation theory to integrate case-based, rule-based and other approaches to legal reasoning. The procedural aspects of argumentation, i.e. as a dialogue and not just a way of comparing pros and cons, began to come into focus [17]. A dialogical approach to integrating arguments from rules and cases was presented not much later [29].

The rule language developed here is much like the one the author developed for the Pleadings Game [17]: rules are reified and subject to exceptions; conflicts between rules can be resolved using other rules about rule priorities; the applicability of rules can be reasoned about and excluded by other rules; and the validity of rules can be questioned. The rule language of the Pleadings Game is similar to other systems developed independently at about the same time [21,28]. All of these systems viewed reasoning with legal rules as argumentation, but unlike in our semantics for LKIF rules, none of them interpreted legal rules as argumentation schemes. Rather, these prior systems either represented legal rules as sentences in a nonmonotonic logic [21,28] or compiled rules

to a set of such sentences [17].[6] Verheij was the first to explicitly discuss the modeling of legal rules using argumentation schemes [36] but like the Pleadings Game interprets rules as abstractions of sets of formulas in a nonmonotonic logic, rather than interpreting rules as abstractions of arguments, i.e. as argumentation schemes in Walton's sense [37]. With the exception of the Pleadings Game, all of these prior systems model argumentation as deduction in a nomonotonic logic, i.e. as a defeasible consequence *relation*. In the Pleadings Game, argumentation was viewed procedurally, as dialogues regulated by protocols, but this was accomplished by building a procedural layer on top of a nonmonotonic logic. In LKIF, the relational interpretation of rules is abandoned entirely, in favor of a purely procedural view, and is thus more in line with modern argumentation theory in philosophy [37] and legal theory [2]. Argumentation cannot be reduced to logic.

The rule language presented here is syntactically similar to the rule languages of the Pleadings Game [17] and the PRATOR system [28]. Our main original contribution is the particular argumentation-theoretic semantics we have given these rules, by mapping them to argumentation schemes using the Carneades model of argument. This approach has at least two advantages:

1. The system can be extended with comparable models of other argumentation schemes. Argumentation schemes provide a unifying framework for building hybrid reasoners. The ESTRELLA platform will make use of this feature to support legal reasoning with ontologies, rules and cases, in an integrated way.
2. Despite the expressiveness of the rule language, which would result in an undecidable logic using the relational approach, since the semantics of LKIF rules is purely procedural, argumentation protocols can be defined for using these rules in legal proceedings which are guaranteed to terminate with procedurally just legal conclusions [31,2,5]

The ESTRELLA reference inference engine for LKIF rules has been fully implemented, in PLT Scheme [30]. Our work in the near future, together with our colleagues in the European ESTRELLA project, will focus on completing the modules for reasoning with cases and ontologies and validating LKIF in pilot applications, for example in the domain of European tax directives.

Our primary goal with LKIF rules has been to develop a knowledge representation formalism for legal rules which is theorectically well-founded, reflecting the state-of-the-art in AI and Law, and practically useful for building legal knowledge-based systems. Rule-based systems have been commercially successful, also for legal applications, but all of the products currently available on the market, to our knowledge, interpret rules either as formulas in propositional or first-order logic or as production rules. Legal rules are neither material implications nor procedures for updating variables in working memory, but rather schemes for constructing

[6] Technically speaking, the rules in PRATOR also may be viewed as domain-dependent inference rules, since they may not be used contrapositively, but nonetheless they are formulated as sentences in the object language.

legal arguments. There are many kinds of rules and correspondingly many kinds of formalisms for modeling rules. LKIF rules is designed to be better suited for modeling legal rules than existing alternatives on the market.

Acknowledgments

This is an extended version of a paper published at the International Conference on Artificial Intelligence and Law [18]. The work reported here was conducted as part of the European ESTRELLA project (IST-4-027655). I would like to thank Alexander Boer, Trevor Bench-Capon, Tom van Engers, Jonas Pattberg, Henry Prakken, Doug Walton, and Adam Wyner for fruitful discussions about topics related to this paper.

References

1. Aleven, V.: Teaching Case-Based Argumentation Through a Model and Examples. Ph.d., University of Pittsburgh (1997)
2. Alexy, R.: A Theory of Legal Argumentation. Oxford University Press, New York (1989)
3. Ashley, K.D.: Modeling Legal Argument: Reasoning with Cases and Hypotheticals. Artificial Intelligence and Legal Reasoning Series. MIT Press, Bradford Books (1990)
4. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. (eds.): The Description Logic Handbook – Theory, Implementation and Applications. Cambridge University Press, Cambridge (2003)
5. Bayles, M.D.: Procedural Justice; Allocating to Individuals. Kluwer Academic Publishers, Dordrecht (1990)
6. Bechhofer, S.: The DIG Description Logic interface: DIG 1.1. Technical report, D1 Implementation Group, University of Manchester (2003)
7. Beckett, D.: Rdf/xml syntax specification (revised) (February 2004), `http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/`
8. Berners-Lee, T.: Notation 3 (1998), `http://www.w3.org/DesignIssues/Notation3`
9. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American 284(5), 34–43 (2001)
10. Clark, J.: Relax ng (September 2003), `http://relaxng.org`
11. Deborah, S.U.D.L.M., McGuinness, L. (Knowledge Systems Laboratory and van Harmelen, F.: OWL web ontology language overview, `http://www.w3.org/TR/owl-features/`
12. Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz — open source graph drawing tools. In: Mutzel, P., Jünger, M., Leipert, S. (eds.) GD 2001. LNCS, vol. 2265, pp. 483–484. Springer, Heidelberg (2002)
13. Fallside, D.C., Walmsley, P.: Xml schema part o: Primer, 2nd edn (2004), `http://www.w3.org/TR/xmlschema-0/`
14. S.C.: for Biomedical Informatics Research. The protege ontology editor and knowledge acquisition system (October 2007), `http://protege.stanford.edu/`
15. Gardner, A.: An Artificial Intelligence Approach to Legal Reasoning. MIT Press, Cambridge (1987)

16. Gordon, T.F.: An abductive theory of legal issues. International Journal of Man-Machine Studies 35, 95–118 (1991)
17. Gordon, T.F.: The Pleadings Game; An Artificial Intelligence Model of Procedural Justice. Springer, New York, Book version of 1993 Ph.D. Thesis; University of Darmstadt (1995)
18. Gordon, T.F.: Constructing arguments with a computational model of an argumentation scheme for legal rules. In: Proceedings of the Eleventh International Conference on Artificial Intelligence and Law, pp. 117–121 (2007)
19. Gordon, T.F., Prakken, H., Walton, D.: The Carneades model of argument and burden of proof. Artificial Intelligence 171(10-11), 875–896 (2007)
20. Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: Combining logic programs with description logics. In: Proceedings of the Twelth International World Wide Web Conference (WWW 2003), Budapest, Hungary, May 2003, pp. 48–57. ACM, New York (2003)
21. Hage, J.C.: Monological reason-based logic. a low level integration of rule-based reasoning and case-based reasoning. In: Proceedings of the Fourth International Conference on Artificial Intelligence and Law, pp. 30–39. ACM, New York (1993)
22. Hart, H.L.A.: The Concept of Law. Clarendon Press, Oxford (1961)
23. Hewlett-Packard. Jena – a semantic web framework (October 2007), `http://jena.sourceforge.net/`
24. Horrocks, I., Patel-Schneider, P., Boley, H., Tabet, S., Grosof, B.N., Dean, M.: SWRL: A semantic web rule language combining OWL and RuleML, `http://www.w3.org/Submission/SWRL/`
25. Loui, R.P.: Process and policy: resource-bounded non-demonstrative reasoning. Computational Intelligence 14, 1–38 (1998)
26. McGuinness, D.L., van Harmelen, F.: OWL Web Ontology Language overview, `http://www.w3.org/TR/owl-features/`
27. Pollock, J.: Defeasible reasoning. Cognitive Science 11(4), 481–518 (1987)
28. Prakken, H., Sartor, G.: A dialectical model of assessing conflicting argument in legal reasoning. Artificial Intelligence and Law 4(3-4), 331–368 (1996)
29. Prakken, H., Sartor, G.: Modelling reasoning with precedents in a formal dialogue game. Artificial Intelligence and Law 6(2-4), 231–287 (1998)
30. T. P. S. Project. PLT Scheme, `http://www.plt-scheme.org/`.
31. Rawls, J.: A Theory of Justice. Belknap Press of Harvard University Press (1971)
32. Rissland, E.L., Ashley, K.D., Loui, R.P.: AI and law: A fruitful synergy. Artificial Intelligence 150(1–2), 1–15 (2003)
33. Routen, T., Bench-Capon, T.: Hierarchical formalizations. International Journal of Man-Machine Studies 35, 69–93 (1991)
34. Sartor, G.: Reasoning with factors. Technical report, University of Bologna (2005)
35. TopQuadrant. Topbraid composer (October 2007), `http://www.topbraidcomposer.org/`
36. Verheij, B.: Dialectical argumentation with argumentation schemes: An approach to legal logic. Artificial Intelligence and Law 11(2-3), 167–195 (2003)
37. Walton, D.: Fundamentals of Critical Argumentation. Cambridge University Press, Cambridge (2006)
38. Wyner, A., Bench-Capon, T.: Argument schemes for legal case-based reasoning. In: JURIX 2007: The Twentieth Annual Conference on Legal Knowledge and Information Systems (2007)