Explaining Subsumption in Description Logics

Deborah L. McGuinness
Dept. of Computer Science
Rutgers University
and
AT&T Bell Laboratories
Murray Hill, NJ 07974 USA
dlm@research.att.com

Alex Borgida
Dept. of Computer Science
Rutgers University
New Brunswick, NJ 08903
borgida@cs.rutgers.edu

September 1994

Abstract

Description Logic-based systems include extensive, complex reasoning components that may produce results that surprise users, yet these systems typically provide little or no explanation support. In this paper, we explore the explanation of subsumption reasoning in Description Logics from one perspective—explanation for knowledge engineers. We adopt a deductive framework for presenting the variety of inferences supported in these systems, and thereby achieve a simple, uniform specification of the notion of explanation. We then address the problem of overly-long and overly-detailed explanations by introducing the notions of atomic descriptions and atomic explanations. We also provide an implementation perspective by discussing the design space and presenting some desiderata for explanation modules. We have implemented our approach in the CLASSIC knowledge representation system.

1 Introduction

Systems that represent complex domains or that perform extensive or intricate inferences often surprise users with their conclusions. Expert systems often fall into this category of systems, and many provide a facility for justifying their conclusions [9, 16]. In expert systems based on rules or Prolog, a simple trace of the program execution/rule firing appears to provide a sufficient basis on which to build an explanation facility. Of course, a full explanation facility may provide more extensive information using domain, problem solving, or user knowledge as in [16] or [15]. In any case, the explanation is often presented as a chain of "inferences" by which a conclusion was reached.

Description Logic-based Systems (DLs) are another knowledge representation and reasoning technology that is increasingly used in applications [13, 10, 20, 7]. One benefit of DLs is that they have a richer representation formalism than standard rule-based systems; at the same time, their reasoning mechanisms are both more extensive and more intricate. Moreover, for efficiency reasons, DLs are often implemented procedurally using all sorts of optimized data structures and algorithms. This makes it impossible to provide even a rudimentary explanation facility based on tracing the execution of the implemented system, and may explain why current DLs do not provide explanation modules, although empirical evidence indicates that explanations are helpful and sometimes necessary.

We present here a model for the process of explaining the reasoning of DLs. The explanations described in this paper are geared for expert users: knowledge engineers who have expectations about what inferences should be performed and who are trying to debug the system. The key idea of our formalization is to cast the various kinds of reasoning into a standard format: proof rules in the style of "natural semantics" [12, 3]. Using these rules, which were developed post-facto from the semantics of the language, we can generate a logical chain of inferences to support any conclusion reached by the system. A second contribution of the formalism is a way of breaking up the process of explanation into more elementary steps, which allow the system and the user to avoid some of the uninteresting details of the explanation. Finally, we discuss how this theoretical framework was realized as part of the CLASSIC knowledge representation system [5].

2 Description Logics - An Introduction

We provide a brief introduction to DLs by presenting some of the features of the CLASSIC system, which is representative of the family. (CLASSIC also happens to embody the core of the KRSS [17] effort and is therefore of independent interest.) CLASSIC's basic elements are individuals, roles and concepts. Individuals in a knowledge base are grouped into classes called concepts; roles are ordinary binary relations that relate individuals to each other. A complete knowledge base consists of a so-called *terminological component*, containing concept and role definitions, and an *assertional component*, describing individuals, their inter-relationships and descriptions in terms of concepts. Although our work also covers assertional information, in this paper we will consider exclusively reasoning in the terminological component.

Figure 1 shows an example knowledge base containing the definitions for `RED-WINE`,

```
RED-WINE ≡ (and WINE (fills color Red)

CABERNET-SAUVIGNON ≡
   (and WINE (fills grape-varietal Cabernet-Sauvignon) (fills color Red))

BLENDED-WINE ≡
   (and WINE (at-least 2 grape-varietal) (all grape-varietal WINE-GRAPE))
```

Figure 1: Sample Definitions

CABERNETSAUVIGNON, and BLENDED-WINE. (In writing our examples, we use the following conventions: concepts in upper case, roles in lower case, and individuals with their first letter capitalized.) Assuming that roles color and grape-varietal, and concept WINE have been previously introduced, this KB describes a RED-WINE as something that is both a WINE and something whose color has value Red.

A BLENDED-WINE, on the other hand, is a WINE made from two or more grape-varietals, all of which must be WINE-GRAPEs; i.e., the instances of BLENDED-WINE are all and only those objects that are instances of WINE, and that have 2 or more fillers for the grape-varietal role.

The main deduction in DLs is *subsumption*: deciding whether one description, D1, is more general than another one, D2 (i.e., whether D2 logically implies D1). For example, CABERNET-SAUVIGNON is subsumed by (or "isa") RED-WINE, a relationship expressed in our notation as CABERNET-SAUVIGNON $\Longrightarrow$ RED-WINE.

Subsumption is used in many ways in a DL:

1. *Classification.* DL-based systems deduce and maintain subclass hierarchies based on the semantics of term definitions, producing a partial order where more general concepts are parents of more specific ones. Subsumption is used to determine a concept's position in the hierarchy. For example, suppose we introduce in Figure 1 a new concept RED-TABLE-WINE, defined as a WINE whose color is Red and which is made from at least three grape-varietals that are WINE-GRAPEs; then RED-TABLE-WINE would be classified below both RED-WINE and BLENDED-WINE because it is subsumed by both of them.

2. *Incoherence.* A concept is incoherent if it can be proven to have no satisfying instances. For example, the concept (**and** (**at-most** 1 color) (**fills** color White) (**fills** color Red)) is incoherent since White is distinct from Red, and so the color attribute has 2 fillers, although it is supposed to have at most one. CLASSIC distinguishes a special bottom concept, called **Nothing**, whose denotation is the empty set.

3. *Disjointness.* Two concepts are disjoint when they can have no common instances, i.e., when their conjunction is subsumed by **Nothing**.

4. *Equivalence.* Two concepts are equivalent, written as $A \equiv B$, if and only if $A \Longrightarrow B$ *and* $B \Longrightarrow A$.

CONCEPT TERM	DENOTATION
Thing	$\Delta^{\mathcal{I}}$
Nothing	$\emptyset$
(**and** C D)	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
(**or** C D)	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
(**not** C)	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
(**all** p C)	$\{d \in \Delta^{\mathcal{I}} \mid p^{\mathcal{I}}(d) \subseteq C^{\mathcal{I}}\}$
(**some** p C)	$\{d \in \Delta^{\mathcal{I}} \mid (\exists d') d' \in p^{\mathcal{I}} \wedge d' \in C^{\mathcal{I}}\}$
(**at-least** n p)	$\{d \in \Delta^{\mathcal{I}} \mid \lvert p^{\mathcal{I}}(d)\rvert \geq n\}$
(**at-most** n p)	$\{d \in \Delta^{\mathcal{I}} \mid \lvert p^{\mathcal{I}}(d)\rvert \leq n\}$
(**same-as** p q)	$\{d \in \Delta^{\mathcal{I}} \mid p^{\mathcal{I}}(d) = q^{\mathcal{I}}(d)\}$
(**fills** p b)	$\{d \in \Delta^{\mathcal{I}} \mid b^{\mathcal{I}} \in p^{\mathcal{I}}(d)\}$
(**one-of** b_1,...,b_m)	$\{b_1^{\mathcal{I}}, \ldots, b_m^{\mathcal{I}}\}$

ROLE TERM	INTERPRETATION
identity	$\{(d,d) \mid d \in \Delta^{\mathcal{I}}\}$
(**inverse** p)	$\{(d,d') \mid (d',d) \in p^{\mathcal{I}}\}$
(**restrict** p C)	$\{(d,d') \in p^{\mathcal{I}} \mid d' \in C^{\mathcal{I}}\}$
(**compose** p q)	$\{(d,d') \mid \exists e\ (d,e) \in p^{\mathcal{I}} \mathrm{and} (e,d') \in q^{\mathcal{I}}\}$
(**role-and** p q)	$\{(d,d') \mid (d,d') \in p^{\mathcal{I}} \mathrm{and} (d,d') \in q^{\mathcal{I}}\}$

Table 1: Syntax and Denotational Semantics of Concept and Role Constructors

These inferences can be particularly useful in knowledge bases that use type hierarchies and that are interested in finding equivalences and incompatibilities between types.

2.1 Syntax and Semantics

The language of descriptions is obtained recursively by starting from atomic symbols for *primitive* concepts and roles. In Table 1, we present the syntax of a simple DL, which we shall use for examples in this paper. (Here, and elsewhere, we use the symbols $C, D, \ldots$ to range over concept descriptions, $p, r, \ldots$ to range over roles, and $a, b, \ldots$ to indicate individuals.) The semantics of terms are given denotationally, using the familiar notion of a *structure* $\mathcal{I}$, which consists of a set $\Delta^{\mathcal{I}}$, and an interpretation function $\cdot^{\mathcal{I}}$, that assigns unary and binary relations over the domain $\Delta^{\mathcal{I}}$ to primitive concepts and roles respectively. Table 1 also presents the recursive extension of the interpretation function to more complex descriptions.

Subsumption — when one description logically implies another description — can be defined in terms of the denotational semantics: description B subsumes description C, written $\mathtt{C} \implies \mathtt{B}$, if and only if $C^{\mathcal{I}} \subseteq B^{\mathcal{I}}$ for every possible structure I. In other words, every individual recognized as an instance of C will also be recognized as an instance of B.

2.2 Subsumption: Implementation

Most practical DLs are implemented by constructing first data structures which represent descriptions in some canonical (normalized) form, and then performing operations on these data structures. For example, in CLASSIC information about various restrictions on a role `r` in some concept are grouped together and summarized into a record structure, with fields role-name, atleast, atmost, fillers, value-restriction. For example, for the description (**and** (**fills** `color Red`) (**fills** `color White`) (**at-least** `1 color`)) this data structure would resemble

{role-id=color, atleast=2, atmost=Maxint, fillers=['Red,'White], ...}

This data structure would be identical to the one generated for the description (**fills** `color (Red White)`). Then, if some description `B` contains such a record rr_B for role `R`, then any candidate subsumee `C` must also have a corresponding record rr_C, such that rr_B.role-id=rr_C.role-id, rr_B.atleast $\leq$ rr_C.atleast, rr_B.atmost $\geq$ rr_C.atmost, rr_B.fillers is a sub-list of the list rr_C.fillers, etc.

The implementation of reasoning with certain concept constructors, such as **same-as**, is in fact quite a bit more complex, relying on an encoding of concepts as graphs, and subsumption requiring graph traversal.

2.3 Subsumption – alternate specification

Alternatively, we can specify the subsumption relationship proof-theoretically. In logic, proofs are syntactic manipulations which usually have the property that formulas derivable by proofs are valid according to the semantics. (In the case of incomplete reasoners, only a subset of the valid formulas can be derived.) Proofs are built up starting from logical axioms or told assumptions using rules of inference, which show what new assertion(s) can be considered "proven", on the basis of previously demonstrated ones. For example, *modus ponens* can be expressed as

$$\frac{\rho \vdash \mathtt{X} \supset \mathtt{Y} \qquad \rho \vdash \mathtt{X}}{\rho \vdash \mathtt{Y}}$$

indicating that if the two antecedent assertions can be deduced (in the context of a KB ρ), then in the next step of the proof so can the consequent.

Under the name of "natural semantics", such kinds of rules have been successfully used in describing various aspects of programming languages [12], by considering the deductive properties of "*judgements*" other than just **is-true** (such as **has-type** or **evaluates-to**). The semantics of DLs, and in particular the rules for subsumption can also be expressed in this form [3], by considering the judgment $\alpha \Longrightarrow \beta$. For example, the subsumption relationship between **all**-restrictions (namely, that the value restriction needs to be more specific in order for the **all**-restriction to be subsumed) is expressed by the rule

$$\frac{\vdash \mathtt{C} \Longrightarrow \mathtt{D}}{\vdash (\mathbf{all}\ \mathtt{p}\ \ \mathtt{C}) \Longrightarrow (\mathbf{all}\ \mathtt{p}\ \ \mathtt{D})}$$

Similarly, part of the meaning of the **and** constructor, as concept intersection, is provided by the rule

$$\frac{\vdash \mathrm{C} \Longrightarrow \mathrm{E}}{\vdash (\textbf{and}\ \mathrm{C}, \ldots) \Longrightarrow \mathrm{E}}$$

A more complex inference rule is one that captures the idea that, given the size of the set of known fillers for a particular role, the minimum number of fillers for this role must be at least the size of the set. This can be expressed by the rule *fills-implies-at-least*:

$$\frac{\text{cardinality}(\{\text{fillers}\})=\text{n}}{(\textbf{fills}\ \text{r}\ \{\text{fillers}\}) \Longrightarrow (\textbf{at-least}\ \text{n}\ \text{r})}$$

One can use rules of this form to produce proofs, where every line of the proof must be justified by an instance of a proof rule, whose numerators are instantiated in previous lines of the proof. For example, consider `BLENDED-WINE`, defined in Figure 1. We might want to see a demonstration that `BLENDED-WINE` $\Longrightarrow$ (**all** `grape-varietal GRAPE`), assuming `WINE-GRAPE` is a subconcept of `GRAPE`. Starting from

`WINE-GRAPE` $\Longrightarrow$ `GRAPE`

one application of the first rule yields

(**all** `grape-varietal WINE-GRAPE`) $\Longrightarrow$ (**all** `grape-varietal GRAPE`)

and then one application of the second rule allows us to conclude that

`BLENDED-WINE` $\Longrightarrow$ (**all** `grape-varietal GRAPE`).

2.4 Mini-CLASSIC

In order to provide details of developing an explanation component, we will use in this paper an extremely simple language, MiniCLASSIC. It has no role constructors, and the concept constructors are restricted to the set {**Thing**, **and**, **all**, **at-least**}, plus a constructor **prim**, which explicitly marks sets of primitive concepts.

In Figures 2 through 4 we provide the inference rules for determining subsumption in this language. (Note that for this simple language, no inference rules approach the complexity of the *fills-implies-at-least* inference, let alone the more complicated inferences involving **same-as**, disjointness reasoning, etc.) The first set of rules define the basic properties of subsumption — it is reflexive, transitive, and allows the substitution of equivalent concepts.

The rules in the next group (Figure 3) are description-constructor specific and are said to determine the *structural subsumption relationship* [8]: when does one term, built with constructor **K**, subsume another term built with the same constructor. Every time a new constructor is added to the language, this set of rules needs to be expanded. For example, if the **at-most** constructor is added, then the obvious rule to add is

$$\frac{\text{n}>\text{m}}{\vdash (\textbf{at-most}\ \text{m p}) \Longrightarrow (\textbf{at-most}\ \text{n p})}$$

The last set of rules involves the "description equivalence" relationship ($\equiv$), and one could therefore think of each rule in Figure 4 as standing for two ordinary subsumption rules.

The distinction between the rules in the three categories may be significant in those DL reasoners that reduce descriptions to a normal form using equivalence rules, and then check subsumption by applying structural subsumption rules.

Ref	$\vdash C \Longrightarrow C$	Concepts subsume themselves
Trans*	$\dfrac{\vdash C \Longrightarrow D,\ \vdash D \Longrightarrow E}{\vdash C \Longrightarrow E}$	If a concept C is subsumed by D which is subsumed by E, then C inherits E's properties and thus is subsumed by E
Eq	$\dfrac{A \equiv B \vdash C \Longrightarrow D}{A \equiv B \vdash C' \Longrightarrow D'}$	C' and D' are C and D with one or more occurrences of A replaced by one or more occurrences of B

Figure 2: Basic Subsumption Rules

Prim-subset*	$\dfrac{FF \subset EE}{\vdash (\textbf{prim}\ EE) \Longrightarrow (\textbf{prim}\ FF)}$	If a concept's primitive set contains another set of primitives, then this concept is subsumed by the smaller set of primitives
THING	$\vdash C \Longrightarrow \textbf{Thing}$	**Thing** is the topmost concept and subsumes all others
And-rt	$\dfrac{\vdash C \Longrightarrow D,\ \vdash C \Longrightarrow (\textbf{and}\ EE)}{\vdash C \Longrightarrow (\textbf{and}\ D\ EE)}$	Concepts subsumed by two concepts are subsumed by the conjunction of those two concepts
And-lft	$\dfrac{\vdash C \Longrightarrow E}{\vdash (\textbf{and}\ C, \ldots) \Longrightarrow E}$	A more specific concept created by ANDING the old concept with new information will still be subsumed by the old concepts parents
All-restr*	$\dfrac{\vdash C \Longrightarrow D}{\vdash (\textbf{all}\ p\ C) \Longrightarrow (\textbf{all}\ p\ D)}$	If value restrictions are subsumed then all restriction is subsumed
At-least*	$\dfrac{n > m}{\vdash (\textbf{at-least}\ n\ p) \Longrightarrow (\textbf{at-least}\ m\ p)}$	Higher lower bounds are more restrictive, hence subsumed by lower bounds

Figure 3: Constructor Subsumption Rules

And-c	$\vdash (\textbf{and}\ \ C) \equiv C$	**and** concept equals concept
Al0	$\vdash (\textbf{at} - \textbf{least}\ 0\ p) \equiv \text{THING}$	0 is the atleast restriction on THING
All-thing	$\vdash (\textbf{all}\ p\ \text{THING}) \equiv \text{THING}$	THING is the value restriction on THING
All-and	$\vdash (\textbf{and}\ (\textbf{all}\ p\ C)(\textbf{all}\ p\ D)\ldots) \equiv (\textbf{and}\ (\textbf{all}\ p\ (\textbf{and}\ C\ D))\ldots)$	**and** distributes over **all**

Figure 4: Equivalence subsumption rules

3 Explanations of Subsumption as Proofs

It is clear from section 2.2 that code tracing is not an appropriate strategy for explanation. Since the implementation doesn't provide a basis for explanation, we need another strategy.

Expert systems, based on OPS5 say, typically use rule firing as a deductive mechanism. From a "deductive" perspective, rules can be viewed somewhat like implications, in which case rule firing corresponds to the *modus ponens* rule of inference in classical logic. Implementations then trace this inference (possibly optimizing the rule selection process), mirroring the deductive process. Similarly, the SLD resolution technique used in Prolog implementations mirrors quite faithfully the deductive process of applying Horn clauses through Modus Ponens, so that Prolog tracers can be viewed as naturally providing an explanation for their conclusions. We therefore turn to the proof theory of DLs as the basis for our explanations.

3.1 Formal Proof as Explanation

The proof theoretic view of DLs provides a simple proof tree structure, with one uniform kind of "rule" being applied everywhere, namely, the natural semantics inference rules, with zero or more antecedents (numerators) and one conclusion (the denominator). If DLs were implemented by directly encoding the above kinds of inference steps, we might be able to present explanations as a trace of internal rule firings, and thereby achieve a similar effect to expert system or Prolog debuggers. However, as mentioned earlier, typically the systems are implemented as complex procedures optimized according to considerations like specific combinations of constructors and operators, particular environmental parameters, etc. Thus, a simple trace of proof rule firings is not accessible and a trace of procedure calls would not be understandable to users.

Deductive databases, such as LDL[18] and CORAL[2], face a similar problem: although the knowledge base can be viewed as a declarative description of the system's beliefs, expressed in First Order Logic, and questions can be posed against this, the question answering process is optimized considerably and is carried out bottom-up. As a result, execution cannot be viewed as simple backward-chaining from the theory.

However, in both the above cases, as long as the underlying implementation is sound and complete with respect to the inference rules, one can construct *in a separate pass* a "proof tree" of the answer from the given facts and rules, and then use this as an explanation.

3.2 An example

In order to evaluate this proposal, let us consider a simple question and provide as an explanation the formal proof using the rules of inference presented in Figures 2 through 4.

Define a concept `A` as

$$\texttt{A} \equiv \ \textbf{(and (at-least } 3 \text{ grape) (prim WINE GOOD))}$$

If asked to explain why

1.	(**at-least** 3 grape) $\Longrightarrow$ (**at-least** 2 grape)	At-least
2.	(**and** (**at-least** 3 grape) (**prim** GOOD WINE)) $\Longrightarrow$ (**at-least** 2 grape)	And-lft,1
3.	(**prim** GOOD WINE)) $\Longrightarrow$ (**prim** GOOD WINE)	Ref
4.	(**prim** GOOD WINE)) $\Longrightarrow$ (**prim** WINE)	Prim-subset,3
5.	(**and** (**prim** GOOD WINE) (**at-least** 3 grape)) $\Longrightarrow$ (**prim** WINE)	And-lft,4
6.	A $\equiv$ (**and** (**prim** GOOD WINE) (**at-least** 3 grape))	Told
7.	A $\Longrightarrow$ (**prim** WINE)	Eq,5,6
8.	(**prim** WINE) $\equiv$ (**and** (**prim** WINE)	And-c
9.	A $\Longrightarrow$ (**and** (**prim** WINE))	Eq,8,7
10.	A $\Longrightarrow$ (**at-least** 2 grape)	Eq,6,2
11.	A $\Longrightarrow$ (**and** (**at-least** 2 grape) (**prim** WINE))	And-rt,10,9

Figure 5: Proof of subsumption

A $\Longrightarrow$ (**and** (**at-least** 2 grape) (**prim** WINE))

we could generate the proof in Figure 5. (Each proof step names the rule used and the line(s) from which the antecedent(s) are obtained.)

We notice a few things about this proof. First, it is considerably longer than one might have expected, given that this is such a simple subsumption relationship; it includes some lines that are obvious to most readers. Since we are interested in explaining more complex DLs, which have many more rules, we must consider pruning out self-evident rule applications to provide more concise proofs. In the above example, we could eliminate applications of Ref (reflexivity), And-c (adding an enclosing **and**), and Eq (substitution of equal concepts). After removing these inferences, the preceding proof would contain just those steps that seem most critical — At-least and Prim-subset.

Secondly, note that a classical way to manage complexity is decomposition into independent parts; this principle can be applied to proofs. In our case, we use And-rt and And-lft to break a conjunctive subsumer concepts into its component conjuncts, and then proceed with separate, smaller proofs of each.

Finally, note that the current proof needs to be presented in its entirety since it has inference rule applications that take proof line numbers as arguments. If proofs are very long, it might be more helpful to present individual steps of a proof that can stand alone. In other words, proof steps might take arguments that do not change according to the order of inference application, and do not draw their meaning by being a part of a particular proof.

3.3 The formal components of a subsumption explanation

According to Hempel and Oppenheim [11], the "basic pattern of scientific explanation" consists of

1. The *explanandum* — a sentence describing the phenomenon to be explained—and

2. The *explanans* — the class of those sentences which are adduced to account for the phenomenon. These fall into two classes: one containing sentences which state certain antecedent conditions; the other is a set of sentences representing general laws.

Our explanandum is a judgement of the form $B \Longrightarrow C$, where B and C are descriptions. The two classes of explanans are the general laws (rules of inference) of the DL, and the antecedent conditions that are the arguments to these laws. These arguments capture the antecedent conditions in the state of the objects. Of course, the antecedent conditions may themselves be in need of explanation, so the result will be a chain.

Based on our earlier observation concerning the desirability of conjunctions, we adopt a two-phase approach to explaining $B \Longrightarrow C$

1. find a description of the form (**and** $A_1 ... A_n$) that is equivalent to C;
2. show that $B \Longrightarrow A_i$ for i=1,...,n

What are the desirable properties of the A_i above? To begin with, we would want them to be as simple as possible. (For this reason we will call them *atomic descriptions*.) Next, the list should be as short as possible, so that no superfluous explanations will be given.

Note that the first step above (normalization), is essentially a demonstration of $C \equiv$ (**and** $A_1 ...$). However, this in turn could be replaced by explanations of why $C \Longrightarrow A_i$. Therefore, if the above two-phase approach works, we have found a way to replace explananda of the form $B \Longrightarrow C$, by a (unordered) set of explananda of the form $D \Longrightarrow A_i$, where A_i are atomic descriptions. We shall call such explananda *atomic subsumptions*. And explanations of atomic subsumptions will be called *atomic explanations*.

One general approach would then be to offer as a first step of an explanation for $B \Longrightarrow C$, a list of atomic descriptions A_i, and then ask the user to specify one or more atomic explanations involving B, C and some A_i. Alternatively, if a more automated approach is desired then the system can provide an explanation of all the atomic descriptions of C. In either case, one result is that rule And-rt is no longer applied explicitly, and hence it does not clutter up the explanations.

To complete the above approach, we need to clarify what atomic descriptions are, and to show that every subsumption proof can be carried out in the two stages suggested above. We will in fact see that an atomic explanation can be presented by explanans that include an inference rule (instance) and possibly more atomic subsumptions, so that this decomposition into atomic explanations can be carried out recursively even for explananda of the form $D \Longrightarrow A_i$.

3.3.1 Atomic descriptions

As noted above, an atomic description is supposed to be as simple as possible. This would mean that it should not have any further conjuncts of its own. Since conjunction is just set intersection of interpretations, conjuncts are at least as general as the original concept. Atomic descriptions therefore need to be in some sense maximally general, and without abbreviations introduced by naming.

We have found the task of defining atomic descriptions to be rather subtle, in the sense that the obvious rule *"an atomic description A cannot be equivalent to some*

*description of the form (***and***B C)"* is inadequate. We present here a list of examples that have proven to be useful in arriving at our current (tentative) definition:

1. Every description `D`, including any atomic ones, is equivalent to (**and** `D D'` **Thing**) where `D'` is any description that subsumes D. For example, the description (**at-least** `3 p`), which we would like say is atomic, is equivalent to (**and** (**at-least** `3 p`) (**at-least** `2 p`) **Thing**).
2. (**one-of** `1 2`), which we would deem atomic, is equivalent to (**and** (**one-of** `1 2 3`) (**one-of** `1 2 4 5`);
3. **Nothing**, which should be atomic, is quivalent to (**and** `C D`), for any pair of disjoint concepts such as (`C=(atleast 2 p), D=(atmost 1 p)`) or (`C=(atleast 6 q), D=(atmost 3 q)`). This leads to (**at-most** `0 r`) being equivalent to (**and** (**all** `r C`) (**all** `r D`)) for disjoint `C` and `D`.
4. When **role-and** is a role constructor, (**some** [**role-and** `child friend`] `C`) is atomic, if C is atomic, and the conjunction in the role cannot be "moved out".

In ordinary logic, clausal form has been a favorite way of identifying "atomic components" for formulas, especially under the term "prime implicant". One could therefore try to translate descriptions into formulae of FOL (see [4]), and verify atomicity by checking that the corresponding formula is a clause without conjuncts. Unfortunately, an atomic description such as (**some** [**role-and** `p q`] `C`) translates to $\lambda x.(\exists y)p(x,y) \wedge q(x,y) \wedge C(y)$, which does have conjunction.

Because of these examples, we propose the following definition:

Definition 1 *A set of* atomic descriptions $\mathcal{D}$ *has the property that if for any element* `D` *of* $\mathcal{D}$ *it is possible to find descriptions* `E` *and* `F` *such that* $D \equiv$ (**and** E F) *and such that at least one of* `E` *or* `F` *is also atomic, then one of the following three conditions holds:*

- $E \Longrightarrow F$
- $F \Longrightarrow E$
- `E` *or* `F` *involve new identifiers (for roles, concepts, individuals, integers) not appearing in* `D`.

The above definition appears to be sufficient for the needs of a language such as CLASSIC. However, it is known to have problems in the presence of role constructors, such as **restrict**, which allow new roles to be created with the use of general descriptions (e.g., role `son` is (**restrict** `child` (**fills** `gender Male`))). The difficulty lies in the fact that a description such as

(**all** (**restrict** `friend` (**all** `age` (**one-of** 31 32)))
(**all** `shoe-size` (**one-of** 8 9 10)))

representing persons whose 31 or 32 year old friends have shoe size between 8 and 10, can then be rewritten as a conjunction

(**and**
(**all** (**restrict** `friend` (**all** `age` (**one-of** 31 32)))
(**all** `shoe-size` (**one-of** 8 9 10 31)))

(**all** (**restrict** friend (**all** age (**one-of** 31 32)))
(**all** shoe-size (**one-of** 8 9 10 32))))

so that it is not atomic according to our definition above.

Based on our definition, (**all** grape (**prim** MERLOT)) is atomic, but (**all** grape (**and** (**prim** MERLOT) (**prim** FRENCH))) is not, because it is equivalent to

(**and** (**all** grape (**prim** MERLOT)) (**all** grape (**prim** FRENCH)))

Similarly, (**fills** color Red) is atomic, but (**fills** color Red Rose) is not because it is equivalent to (**and** (**fills** color Red) (**fills** color Rose)).

The *atomization process* is expected to have the property that if it produces a description D of the form (**and** E_1 ... E_n), then E_i does not subsume any other E_j. Note that "typical" normalization processes will obey this definition since they group information about individual constructors together and remove simple redundancies, e.g., they reduce (**and** (**at-least** 2 r) (**at-least** 3 r)) to (**at-least** 3 r).

3.3.2 Atomic explanations

As stated earlier, atomic explanations are the building blocks of our explanations: each will contain one inference that justifies belief in the atomic subsumption in question. It consists of the inference type and any appropriate arguments. It should be able to represent a justification for any line of a proof such as the one in Figure 5.

Definition 2 *An* explanation step *has the form*

A ⟹ B **because** ruleId(<argument list>)

where B is an atomic description, ruleId is usually the name of an inference rule, and argumentlist is a set of bindings for (some) of the variables in the inference rule.

The simplest atomic explanation is *told information*: if someone asks why A ≡ (**and** (**prim** GOOD WINE) (**at-least** 3 grape)), it is because we were explicitly told this definition:

A ⟹ (**and** (**prim** GOOD WINE) (**at-least** 3 grape)) **because** told-info

In order to eliminate some other obvious inferences, we also report told-info as the reason why A is subsumed by each of the syntactically occurring conjuncts of its definition, i.e.,

A ⟹ (**prim** WINE) **because** told-info

A ⟹ (**at-least** 3 grape) **because** told-info

The inference told-info does not require any arguments. A simple and frequent inference type that does require an argument is transitivity. If a new concept B is created as a subconcept of A, i.e., B ≡ (**and** A ...), then one reason why Bs have at least 3 grapes is because of transitivity through A. We usually refer to this inference as **inheritance**. Inheritance requires an argument since we need to know from what concept information was inherited. In this case, the atomic explanation would be

B ⟹ (**at-least** 3 grapes) **because** inheritance(A).

Suppose we are given an additional concept C, having B as a conjunct; then C ⟹ B ⟹ A, so C would have (**at-least** 3 grape) by transitivity and inheritance through both B and A, whether or not B stated anything explicitly about the grape

role. We therefore limit inheritance to report only from the concept that contains the description as told information. In the above case, the explanation of why C satisfied the description (**at-least** 3 grape) would be

C $\Longrightarrow$ (**at-least** 3 grapes) **because** inheritance(A)

To proceed further, consider explaining why

(**all** wines (**at-least** 4 grapes)) $\Longrightarrow$ (**all** wines (**at-least** 3 grapes))

This is justified by one application of All-restr, the inference rule reprinted here from Figure 3

$$\text{All-restr}^* \quad \frac{\vdash C \Longrightarrow D}{\vdash (\textbf{all}\ p\ C) \Longrightarrow (\textbf{all}\ p\ D)}$$

In this case, p=wines, C = (**at-least** 4 grape), and D = (**at-least** 3 grape), and we can refer to the "law" used in this deduction by the name of the rule and the arguments involved:

(**all** wines (**at-least** 4 grapes)) $\Longrightarrow$ (**all** wines (**at-least** 3 grapes))
because All-restr(p=wines, C = (**at-least** 4 grape), D = (**at-least** 3 grape))

Observe that the preceding is not a complete explanation, since it left us with an intermediate explanandum: why was (**at-least** 4 p) subsumed by (**at-least** 3 p)? The answer to this is a separate atomic explanation — an application of the At-least rule about the ordering of **at-least** terms.

3.4 Explanation chains

From the above example, and the general form of the inference rules we can see that atomic explanations naturally *chain* backward: to explain one atomic description, we refer to a rule (instance) and one or more antecedents to that rule. These antecedents take the form of atomic descriptions. (If they were not atomic, it could usually be proved that the original judgement was also not an atomic description). In turn, we offer atomic explanations for each of these antecedents, until we bottom out in one of several ways: inference rules without antecedents (e.g., told-information), or ones whose antecedents are theorems of mathematics (e.g., At-least ordering rule). In our case, the latter are taken as self-evident, although one might invoke a different explanation mechanism to deal with them, if they are too complex.

Note that in principle, the user could extend the set of inference rules which stop the chaining. For example, one option in CLASSIC is to stop explanation chains when a user-defined rule fires.

To this point, we have simplified the situation by considering only cases where there is only one way to justify an atomic subsumption. In some cases, these can be derived in multiple ways. For example, the fact that B subsumes C could be explicitly told as well as inherited, or it could be inherited from several distinct parents. In general, in such cases we form explanation sets out of every applicable atomic explanation, and we have multiple (disjunctive) explanation chains. It is also possible to branch (conjunctively) into several explanations in those cases where an inference rule has multiple antecedents. Because it is possible for explanation chains to become long and branching, we suggest giving the user a lot of control over their generation.

To conclude, let us reconsider the example from Section 3.2. Asking why

A $\Longrightarrow$ **(and (at-least** 2 grape) **(prim** WINE))

is now equivalent to explaining why A is subsumed by each of the two atomic descriptions of the subsumer: **(at-least** 2 grape) and **(prim** WINE):

A $\Longrightarrow$ **(at-least** 2 grape) **because** at-least-ordering(N=3,M=2,r=grape)

For readability, we associate templates with the various inference rules, so that the above is actually reported as at-least-ordering on role grape: 3 is greater than or equal to 2

In this case the inference chain ends with a theorem of arithmetic. Note that if we want to further abbreviate our explanations, we could leave out the relatively abvious statement $3 \geq 2$, so that the atomic explanation would be just A $\Longrightarrow$ **(at-least** 2 grape) **because** at-least-ordering(r=grape)

A $\Longrightarrow$ **(prim** WINE) **because** primitive-subset({Wine,Good}, {Wine})

Note again the implicit application of rule And-lft.

3.5 Filtering

We have already discussed some ways in which explanations are abbreviated by not reporting intermediate steps in proofs, such as transitivity of inheritance and the conjunction rules. Good explanations need to do much more extensive pruning though.

One possibility is to allow the end users to choose to "turn off" the reporting of additional inferences. This can be accomplished either by providing different "levels of sensitivity", or by giving the user access to switches controlling each individual kind of inference.

An additional kind of filtering can be applied to the arguments of inference rules. For example, we might want to provide as arguments only variables that appear in the numerator of the inference rule but **not** in the denominator. This makes sense in cases where there is only one possible way of instantiating the denominator, given the subsumer and subsumed descriptions. For example, the at-least-ordering and primitive-subset inferences used in the previous sections could easily be understood even without arguments.

An example of a case where arguments cannot be dropped is one in which some concept A inherits some description from one of its many parents: omitting the parent from which A inherited might require the user to look at all of A's parents to discover which one contained the restriction.

4 Developing an Explanation System

In order to build an explanation system based on the preceding theory, we need to accomplish the following tasks in any particular situation:

- Identify the (sub)language of atomic descriptions.
- Present appropriate rules for normalizing a description into atomic conjuncts ("atomization").
- Identify appropriate rules of inference (schematized with meta-variables) for subsumption reasoning in this system. These rules, with their arguments, will be used to form the atomic explanations.

- Develop algorithms for reconstructing the subsumption proof using the rules of inference.

4.1 Atomic Descriptions

Let us consider the case of MiniCLASSIC. Here we propose the following syntax for atomic descriptions:

```
<a-descr> ::=
        Thing|
        (at-least <integer> <role> ) |
        (prim <identifier> ) | /* note only 1 primitive concept on each sub-descr */
        (all <role-name> <a-descr> ) /* allows nested alls */
```

Our first task is to make sure that the descriptions generated by this grammar are indeed atomic:

Theorem 1 *The descriptions generated from <a-descr> in the above grammar are atomic.*

Proof We need to show that if some description D is generated by the above grammar and $D \equiv$ (**and** E F) for *arbitrary* MiniCLASSIC descriptions E and F, then $E \Longrightarrow F$ or vice-versa, or at least one of E and F involve new identifiers. The proof is by structural induction on the form of `D`, and is probably most easily done in the semantic domain. To begin with, note that if $D \equiv$ (**and** E F) then $D \Longrightarrow E$ and $D \Longrightarrow F$. If either E or F are **Thing** or D itself, then the desired result of $E \Longrightarrow F$ or $F \Longrightarrow E$ holds immediately. So we need to look only for those cases where E is not D or **Thing**.

- D=**Thing**: then $D \Longrightarrow E$ implies that $E^{\mathcal{I}} =$ **Thing**$^{\mathcal{I}}$, so we are done.
- D=(**prim** Y): if $D \Longrightarrow G$ then $G^{\mathcal{I}}$ must contain $Y^{\mathcal{I}}$. If G is not (**prim** Y) (i.e., D), then $G^{\mathcal{I}} =$ **Thing**$^{\mathcal{I}}$, and we are done again.
- D=(**at-least** n p): if $D \Longrightarrow G$ and $G \not\equiv$ **Thing**, then G must be equivalent to (**at-least** `m p`), for $m \geq n$. But then either $m_E \leq m_F$ or $m_F \leq m_E$ since integers are totally ordered; hence either $E \Longrightarrow F$ or $F \Longrightarrow E$.
- D=(**all** p C): if $D \Longrightarrow G$ then $G \equiv$ (**all** pH) such that $C \Longrightarrow H$. Hence $D \equiv$ (**and** (**all** p H_E)(**all** p H_F)), which itself is $\equiv$ (**all** p (**and** H_E H_F)). But then $C \equiv$ (**and** H_E H_F), and since C is atomic by induction (since it is an a-descr from the syntax), we must have $H_E \Longrightarrow H_F$ or $H_F \Longrightarrow H_E$, and hence $E \Longrightarrow F$ or $F \Longrightarrow E$.

■.

4.2 Atomization

In order to be able to find a normal form for every concept in the language, we need to prove that the above set of constructors is sufficiently large/complete.

Theorem 2 *Every description C in Mini*CLASSIC *can be written in the form* (**and** $A_1 ... A_n$), *where* A_j *is derivable from* `<a-descr>`.

Proof By induction on the syntactic structure of C. When C is **Thing** or (**at-least** n p), C is an a-descr. For C=(**prim** X ... Y), we have $C \equiv$ (**and** (**prim** X)...(**prim** Y)). If C=(**all** p C'), and C' is not an a-descr, then by induction it can be written as (**and** $A'_1 ... A'_n$), and in turn $C \equiv$ (**and** (**all** p A'_1)...(**all** p A'_n)). Finally, if $C =$ (**and** $C_1 ... C_n$) then by induction $C_i =$ (**and** $A_{i,1} ... A_{i,m_i}$), and by distributivity of **and**, C is the conjunction of all the atomic $A_{k,l}$. ■.

Note that the above proof identifies the rules of inference needed for converting to atomic normal form.

An open question is whether the set of atomic constructors should be allowed to contain redundancy. For example, in CLASSIC, there is a special constructor **range**, for representing ranges of integers, as well as a general constructor, **one-of**, for representing sets of values. The first is redundant because (**range** 4 7) ≡ (**one-of** 4 5 6 7). However, one might still want to conduct explanations in the notation closer to the one in which the user encoded the knowledge-base.

4.3 Atomic Explanations

Our atomic explanations contain inference names and appropriate arguments. The explanation system designer therefore needs a complete list of inference rules. If an implementor is lucky enough to have all the inferences, in the form of proof rules, the implementor can name each rule and create one argument for each unbound variable in the proof rule. In reality, there are other factors to consider.

1. The implementation may be available but the proof rules may not be explicitly written down. To find all the inferences, look at the places where a data structure could be modified. Any potential change is due to an inference. Determining the arguments may involve a detailed code analysis to find all the conditions that held before this inference could fire.
2. Even if we have a list of proof rules, if a system evolves, the proof rules may not accurately reflect the state of the system so they need to be checked.
3. Given a complete current set of proof rules, we may want to group some of the rules together under one name to minimize the number of distinct inferences that a knowledge engineer might need to understand. For example, if we were reporting all the inferences in Figure 3, we might would want to group And-rt and And-lft into one rule. In our implementation, we have grouped together some complicated related inferences.
4. In some cases, we may want to collapse a sequence of two or more inferences together into one inference. For example, if we are given a definition for concept $A \equiv$ (**and** (**at-least** 1 r) B), we have grouped together one application of told-information, with reflexivity and and-left so that we can say $A \Longrightarrow$ (**at** − **least** 1 r) because of told-information. We could have also proved this using transitivity. Here we did not want to eliminate the inference, but in certain cases, we want to collapse it with others. It is particularly useful to collapse inferences into the terminating inference — told-information in our case.
5. Some rules are best used in a limited manner. We gave an example limiting the reporting of the transitivity rule as inheritance in Section 3.3.2.

6. We also may want to skip explaining some rules. For example, there are a significant number of equivalence rules in CLASSIC, such as the distribution of **and**, which we do not explain. Although one analysis of the rules in CLASSIC approaches 100 in number, by grouping and eliminating "obvious" inferences such as the equivalence rules, we reduced the number of types of inferences in our explanation component to 40.

7. Consider eliminating some arguments to inference rules, as illustrated in Section 3.5.

5 Implementing an Explanation System

The main task of the implementation is to recover the sequence of inference rule applications that demonstrate the subsumption to be explained so that they can be turned into atomic explanations.

There are several desiderata that come to mind concerning the relationship between the description logic reasoner and the explanation component. First, there is the issue of *cost*: Clearly, every new capability, such as explanation, carries with it some extra cost in terms of processing time and space. Since we view explanation as a debugging tool, it seems evident that it would be desirable if these costs where minimized when an explanation is **not** desired (e.g., when the DL-based system is already debugged, and in the field). Conversely, it is not unreasonable to expect that asking for an explanation might delay the reporting the answer in a manner that is observable by the user.

A second important issue is that of *modularity*. On the one hand, it is very likely that the DL reasoner will have been developed before the explanation system, and ideally the explanation component would be a separate "add-on" component. This desire is however tempered by the need to accept that the DL reasoner may itself evolve in terms of the inferences it performs (especially in the case of incomplete reasoners), and hence explanation will have to track this evolution [19].

In order to accommodate the first desire, to minimize cost when explanation is not asked for, we rejected an approach which kept up-to-date all justification structures while processing the knowledge base. Thus, explanation on demand will be required. For full modularity, the ideal solution would have been to derive a proof using some general-purpose theorem-prover, which is provided the axioms for the subsumption relationship. In order to deal with the undecidability aspects of most theorem-provers, we would very likely want to use "hints" from the way in which subsumption was found by the normal, procedural reasoner. This paradigm is in fact illustrated for the **same-as** constructor in the next subsection.

A second alternative is to conceptualize the explanation component as follows: beginning with told information about the subsumption to be explained, throw out any derived information, rederive the logical closure on the objects *while building detailed explanation structures*, and finally present (a pruned version of) the explanation structures. This can be viewed essentially as adding an extra argument to the procedure that checks subsumption: when the switch is set to 'explain', the processing is altered so that additional data structures (resembling a truth-maintenance structure) are set up to record the inference and its arguments. In order to minimize computational cost and the problem of maintaining consistency, we have chosen to utilize the current de-

duction algorithms, but to affect existing code as little as possible. When one inference is added to the main system, the person adding the inference needs to write the main code and also reflect this inference in the explanation algorithm.

The following are some of the principles embodied in the implementation of the CLASSIC explanation component:

1. *Exploit cached subsumption.* The subsumption hierarchy contains information that explains some inferences trivially. For example, if an explanation is being generated for `A` $\Longrightarrow$ `C`, and `C` is contained in a parent, `D`, of `A`, then one atomic explanation is inheritance(`D`).

2. *Use pre- and post-processing effectively.* While it may be appropriate to explain most inferences by some recalculating and documenting algorithm, some inferences may be handled more efficiently in a pre- or post-processing phase. For example, some inferences may only be appropriate explanations if no other explanation has been found; such cases can be handled most effectively in a post processing.

   In CLASSIC, one such inference is attribute-implies-at-most-1. If a role is known to be an attribute, then it must have an **at-most** restriction of one. Since this inference is obvious to most users and extensive in most applications, we choose to only provide it when there is no other reason for a particular (**at-most** 1 `r`) restriction.

3. *Minimize dual algorithm development.* The majority of inferences are explained using calls to the main inference routines. This code has been modified to accept an explanation flag that is set if explanation structures are to be modified. The only additions to the main code are calls to augment explanation structures when inferences are fired. Additional inference explanation is straight forward as long as a new inference does not take some new type of argument

   While we do suggest separating out inferences that can be handled more efficiently and that are unlikely to change, separating out complicated inferences that may need to evolve over time could result in a maintenance problem. Sometimes separation may be necessary since some main deductive routines or structures would need radical changes in order to support explanation, such was our experience with some **same-as** inferences, but efforts should be taken to minimize the number of such separations.

6 Explaining Co-reference Constraints

It will not always be possible to calculate explanations using the main deduction code with minimal code impact. We were forced to separate explanation calculation for one of the most complicated inferences in CLASSIC — **same-as** handling, because extensive changes to the main deductions would have been required in order to provide sufficient information for explanations. In this case one algorithm is much more efficient at determining subsumption, while another is more effective at finding the chain of inferences that were involved.

The **same-as** concept constructor is useful in situations where one needs to relate values of different attributes of some object. For example, if persons have a `name` attribute, corresponding to their last name, as well as attributes for two parents, we

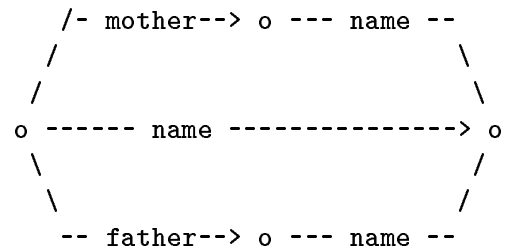

Figure 6: Graph data structure representing **same-as**

might be interested in the concept of "traditional person", whose last name is the same as that of their father and mother; this can be described as:

(1) `(same-as (mother name) (name))` **and** `(same-as (father name) (name))`
Reasoning with this constructor is a form of equality reasoning, where we have versions of the familiar rules of transitivity, symmetry and reflexivity, as well as substitution of equal for equals. For example, the above description is equivalent to

(2) `(same-as (father name) (mother name))` **and** `(same-as (father name) (name))`
The inference rules for **same-as** are presented below (none of them have antecedents):

$\vdash$ (**same** – **as** $pp\ pp$) $\equiv THING$
$\vdash$ (**same** – **as** $pp\ qq$) $\equiv$ (**same** – **as** $qq\ pp$)
$\vdash$ (**same** – **as** $qq.p\ rr.p$) $\equiv$ (**same** – **as** $qq\ rr$)
$\vdash$ (**same** – **as** $p.qq\ p.rr$) $\equiv$ (**all** p (**same** – **as** $qq\ rr$))
$\vdash$ (**and** (**same** – **as** $vv.qq\ rr$)(**same** – **as** $vv\ ww$)) $\Longrightarrow$ (**same** – **as** $ww.qq\ rr$)

Reasoning with **same-as** is not as straightforward as with other constructors because there are an unbounded number of inferences that can be drawn from a single construct, and there is no canonical reduced form for them. For example, the description (**same-as** (ff) (ff . qq)) is subsumed by (**same-as** (ff)(ff . qq . . . . . qq)) for arbitrary number of qq. The difficulty is essentially the same as for regular expressions: there is no canonical form for regular expressions, though their identity (or containment for that matter), can be deduced from a finite automaton representation. An efficient implementation of **same-as** reasoning ([1]), which is part of the CLASSIC system, works on exactly this principle: a data structure, called a description graph in [6], represents the coreference relationships in a concept, and to see whether that concept is subsumed by a term like (**same-as** (pp) (rr)), one needs to trace the paths pp and rr in the automaton/graph, and verify that they end at the same node. For example, the original description (1) results in the graph in Figure 6.

Unfortunately, the identical graph might be obtained from the concept (2). Yet we would want to give different explanations of why **same-as** `(mother name) (father name)` holds in the two cases: in one case it is derived by transitivity, in the other it is told information.

This then is an extreme example of a situation where the internal, efficient representation of a concept makes explanation of inferences impossible. Interestingly, the

following simple logic program can, in principle, deduce all and only the **same-as** relationships entailed by some conjunction of told **same-as** restrictions (possibly nested inside **all** restrictions).

```
told-eq([father,name], [name]).
told-eq([mother,name], [name]).

base-eq(X,Y) :-told-eq(X,Y).
base-eq(X,Y) :-told-eq(Y,X).
eq(X,X).
eq(X,Y) :- base-eq(X,Y).
eq(X,Y) :- append(Xa,Xb,X), base-eq(Xa,Va),append(Va,Xb,V),eq(V,Y).
```

The original equalities are represented as ground assertions involving the predicate `base-eq`. Assertions of the form (**all** p (**same-as** q r)) can either be converted directly to `base-eq([p,q],[p,r])`, or can be encoded with a new predicate `all` directly as `all(X,eq(q,r))`, together with a rule `base-eq([X|Y],[X|V]) :- all(X,eq(Y,V))`. Clearly, this program can be augmented to record the sequence of rule applications and told equalities which lead to the demonstration of some **same-as** construct, and these can then be presented as part of an explanation.

Readers familiar with Prolog will however realize that the above program will easily enter an infinite loop: performing a depth-first search starting from goal `eq([a,b,c],[d,e,f])`, in the presence of `base-eq([a],[m])`, one application of the last rule will lead to the goal `eq([m,b,c],[d,e,f])`, which, after another application of the last rule, will lead *back* to `eq([a,b,c],[d,e,f])`. This problem can be avoided by forcing a breadth-first search of the goal tree, or equivalently, by putting a bound on the depth to which we explore. For example, we can introduce two additional arguments, `Now` and `Depth`, for the `eq` predicate, and modify the last rule to read

```
eq(X,Y,Now,Depth) :- Now < Depth,
                     append(Xa,Xb,X),
                     base-eq(Xa,Va),append(Va,Xb,V),
                     New is Now + 1, eq(V,Y,New,Depth).
```

This latter approach is only feasible if we can bound a priori the number of inference rule applications. Although this could be done on purely theoretical grounds, we can obtain a much more accurate bound by considering the implementation data structure: if `(same-as (pp) (qq))` demonstrably holds on some concept graph G, then only those equalities are potentially relevant which end up pointing to nodes on the paths traced from the root by following paths labeled by pp and qq. (To see this, observe that we can omit all other equalities, and yet in the constructed graph `(same-as (pp) (qq))` would continue to hold.) Although we cannot identify the actual equalities, we can *count* them, by counting the in-degree (minus 1) of every node in the set traversed when following paths `pp` and `qq`. (The initial node is not counted, and the final, common, node is only counted once.) For example, if we were to check that (**same-as** (name) (father name)) holds in the above graph, the relevant sum of in-degrees would be $(3-1)+(1-1)$, which is 2.

For the second example, the count would be $(3-1)+(1-1)+(1-1)$, which is also 2 in this case.

Therefore the maximal depth to be given for any particular search can be computed from the existing implementation structure, and then passed to the explanation component; in turn, this essentially does an (inefficient, but bounded) exhaustive search of all possible inferences until a proof is found. This illustrates our point that the implemented description reasoner can be used as a source of hints for reconstructing detailed formal proofs, which can then be presented as explanations.

7 Conclusion

We have developed a deductive framework for explaining description logics based on natural semantics style proof rules. We implemented our approach in CLASSIC and we can explain every inference that CLASSIC performs. For illustrative purposes, this paper has focussed on concept subsumption. We presented the rules of inference for a simple description logic and used them to provide formal proofs of subsumption relationships. We introduced the notions of atomic descriptions, atomic explanations, and explanation chains to provide a methodology for simplifying explanations. We also described various strategies filtering that can be used to shorten explanations. We drew on our implementation and application experience to gather some desiderata for future DL explanation system designers. Finally, we included some special challenges posed to most explanation systems by the **same-as** constructor, and used this to reinforce why "standard" tracing explanation approaches will not work, and also show how implementations can be exploited in explanation design.

Work beyond the scope of this report, but included in our current implementation, includes explaining individual reasoning (in particular, propagations and rule applications), explaining errors, and simple "why not" explanations. Work in progress includes expanded pruning for inferences, and presentation using a declarative description of the pruning in the form of meta information. We are also providing more assistance for users to discover the "right" question to ask. In error situations, this means automatically generating the specific question that may be asked to retrieve the explanation chain for the error.

New types of explanation questions may be required too, for example, if part of reasoning includes inferring containment relationships. Then questions of (positive and negative) containment in addition to subsumption are necessary.

References

[1] Aït-Kaci, *A Lattice Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures*. Ph.D. thesis, University of Pennsylvania, 1984.

[2] T. Arora, R. Ramakrishnan, W.G. Roth, P. Seshadri, and D. Srivastava. Explaining Program Evaluation in Deductive Systems. Submitted for Publication, 1993.

[3] A. Borgida. From Type Systems to Knowledge Representation: Natural Semantics Specifications for Description Logics. In *International Journal of Intelligent and Cooperative Information Systems*, pp.93–126, March 1992.

[4] A. Borgida, "On the relationship between Description Logic and First Order Logic Queries", *Proc. CIKM'94*, (to appear).

[5] A. Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnick. CLASSIC: A Structural Data Model for Objects. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 59–67, June 1989.

[6] A. Borgida, P. F. Patel-Schneider, "A Semantics and Complete Algorithm for Subsumption in the CLASSIC Description Logic", *Journal of Aritificial Intelligence Research, vol. 1*, 1994, pp.277-308.

[7] R. J. Brachman, P. G. Selfridge, L. G. Terveen, B. Altman, A. Borgida, F. Halper, T. Kirk, A. Lazar, D. L. McGuinness, and L. A. Resnick. Knowledge representation support for data archaeology. In *First International Conference on Information and Knowledge Management*, November 1992.

[8] W. Cohen, A. Borgida, H. Hirsh, "Computing Least Common Subsumers in Description Logics", *Proceeding of AAAI'92 Conference*, pp.754–760, San Jose, CA, Jul y 1992.

[9] W. J. Clancey. The Epistemology of a Rule-Based Expert System—a Framework for Explanation. In *Artificial Intelligence 20*, pages 215–251, 1983.

[10] P. Devanbu, P. Selfridge, B. Ballard, and R. J. Brachman. Steps towards a knowledge-based software information system. In *Proc. IJCAI-89*, Detroit, MI, August 1989.

[11] C. G. Hempel and P. Oppenheim. Studies in the Logic of Explanation. In *The Structure of Scientific Thought*, E. H. Madden, editor, Riverside Press, pages 19–29, 1960.

[12] G. Kahn. Natural Semantics Rapport de Recherche No. 601, INRIA, Sophia Antipolis, France.

[13] E. Mays, C. Apte, J. Griesmer, J. Kastner. Organizing Knowledge in a Complex Financial Domain. In *IEEE Expert*, Fall 1987, 61–70.

[14] J. D. Moore. Explanation in Expert Systems- A Survey. University of Southern California/Information Sciences Institute Tech Report ISI/RR-88-228, December 1988.

[15] J. D. Moore. A Reactive Approach to Explanation. PhD thesis, Department of Computer Science, University of Southern California, 1989.

[16] R. Neches and W. R. Swartout and J. Moore. Explainable (and Maintainable) Expert Systems. In *Proc. IJCAI-85*, Los Angeles, CA., (1985), 382–389.

[17] P. F. Patel-Schneider. Working Version: Description Logic Specification from the KRSS Effort. AT&T Bell Laboratories, 1992.

[18] O. Shmueli, and S. Tsur. Logical Diagnosis of LDL Programs. In *New Generation Computing*, OHMSHA, LTD and Springer-Verlag, Volume 9, pages 277–303, June 1991.

[19] W. R. Swartout and J. D. Moore. Explanation in Second Generation Expert Systems. In Jean-Marc David, Jean-Paul Krivine, and Reid Simmons, editors, *Second Generation Expert Systems*. Springer-Verlag, in Press.

[20] Wright, J. R., Weixelbaum, E. S., Brown, K., Vesonder, G. T., Palmer, S. R., Berman, J. I., Moore, H. H., "A knowledge-based configurator that supports sales, engineering, and manufacturing at AT&T network systems" In *Proceedings of the Innovative Applications of Artificial Intelligence Conference*, pp.183–193, 1993.