EXPLAINING REASONING IN DESCRIPTION LOGICS

BY DEBORAH L. MCGUINNESS

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Alexander Borgida

and approved by

____

____

____

____

New Brunswick, New Jersey

October, 1996

ABSTRACT OF THE DISSERTATION

EXPLAINING REASONING IN DESCRIPTION LOGICS

by Deborah L. McGuinness

Dissertation Director: Alexander Borgida

Knowledge-based systems, like other software systems, need to be debugged while being developed. In addition, systems providing "expert advice" need to be able to justify their conclusions. Traditionally, developers have been supported during debugging by tools which offer a *trace* of the operations performed by the system (e.g., a sequence of rule firings in a rule-based expert system) or, more generally by an explanation facility for the reasoner. Description Logics, formal systems developed to reason with taxonomies or classification hierarchies, form the basis of several recent knowledge-based systems but do not currently offer such facilities.

In this thesis, we explore four major issues in explaining the conclusions of procedurally implemented deductive systems, concentrating on a specific solution for a class of description logics. First, we consider how to explain a highly optimized procedural implementation in a declarative manner. We begin with a formal proof-theoretic foundation for explanation and we illustrate our approach using examples from our implementation in the CLASSIC knowledge representation system. Next, we consider the issue of handling long, complicated deduction chains. We introduce methods designed to break up description logic queries and answers into small, manageable pieces, and we show how these are used in our approach and how they support automatically

generated explanations of followup questions. Next, we consider the problem of explaining negative deductions. We provide a constructive method for explanation based on generating counter-examples. Finally, we address the issue of limiting both object presentation and explanation. We offer a meta-language for describing interesting aspects of complicated objects and use this language to limit the amount of information that should be presented or explained. The work in this thesis has been motivated by design and application work on a description logic-based system and a significant portion of our work has been implemented for CLASSIC and is in use.

Acknowledgements

Throughout the decade that I spent pursuing this dissertation, I have had the privilege of working with many extraordinary people who have shaped my thoughts and my work. I owe a great deal to all of them and I would like to acknowledge some of the major contributors here.

My thesis advisor, Alex Borgida, has been the guiding inspiration of this work. He has contributed in countless ways, always giving generously of himself. He constantly expanded my views and always found ways to make my work better. More than anyone else, he has impacted my way of approaching a research problem. I am deeply indebted to him.

I would like to thank my committee: Haym Hirsh, Kaz Kulikowski, and Johanna Moore for their insights, suggestions, and support.

Throughout my entire tenure at Rutgers, I have also been fortunate enough to be a researcher in the Artificial Principles Research Department of AT&T Bell Laboratories[1]. I believe I have had a rare opportunity to grow as a researcher in one of the most stimulating places in the world. Ron Brachman provided seemingly limitless energy and support of the group and for my work. One thing I always appreciated about him was his skill at quickly sensing the idea with the greatest potential from a vast sea of possibilities and his talents at focussing one on that idea and imparting increased enthusiasm in the process.

I also owe many thanks to the core CLASSIC group. I am deeply indebted to Lori Alperin Resnick for her collaboration in implementing the explanation system for CLASSIC. My work has improved in many directions as a result of discussions with her and because of our extensive collaborations on applications. I also owe Peter Patel-Schneider

[1]Most recently, this group became part of AT&T Laboratories

my gratitude for his many suggestions, thesis chapter and conference submission readings, and general support. His depth and breadth of technical knowledge has been a tremendous resource for me. I also wish to thank Charles Isbell for his implementation work on my initial meta language for CLASSIC. I also must thank our initial application developer, Jon Wright. It is through my work with him that I found my thesis topic since it became clear that CLASSIC required an explanation component in order to be usable by our Business unit customers. Also, I wish to thank Merryll Abrahams for her work integrating the explanation component into the graphical interface of NEO-CLASSIC. Throughout the six year course of my work on CLASSIC applications, I have also received valuable input from a number of others including Prem Devanbu, Charlie Foster, Scott Hofmeister, Diane Litman, Harry Moore, and Elia Weixelbaum.

Many others have contributed to my dissertation experience through discussions in the halls of the labs, Rutgers, or at professional meetings. Some who have impacted this work include: Phillip Bohannon, Martin Carroll, William Cohen, Henry Kautz, Divesh Srivastava, Bill Swartout, Rich Thomason, and his knowledge representation students at University of Pittsburgh.

Finally, I would like to thank my friends and family for their unflagging emotional support. It has been a long road and I could not have imagined it without their help.

Dedication

This document is dedicated to my family and friends for supporting me on the long path to the Ph.D.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

DL	is for Description Logic
DLKRS	is for Description Logic-based Knowledge Representation and reasoning System
FOPC	is for First Order Predicate Calculus
KB	is for KnowledgeBase
KRS	is for Knowledge Representation and reasoning System
TMS	is for Truth Maintenance System

Chapter 1

Introduction

Knowledge-based systems, like other software systems, need to be debugged while being developed. In addition, systems providing "expert advice" need to be able to justify their conclusions. Traditionally, developers have been supported during debugging by tools that offer a *trace* of the operations performed by the system (e.g., a sequence of rule firings in a rule-based expert system as exemplified in [41] (part 2,chapter 3)), or, more generally by an explanation facility for the reasoner (e.g., [96]). Description Logics (DLs) form the basis of several recent knowledge-based systems (e.g., [130, 26, 45, 108, 119, 73, 76, 1, 47, 84, 86]), but do not currently offer such facilities. This is especially troublesome since DL reasoners perform a considerable variety of inference types, some of whose results have repeatedly proven, in practice, to be unexpected by developers.

Description Logic-based systems have been used in a variety of applications. In some cases their role is to enforce or verify constraints and deduce new components in the design or configuration of systems data [130, 91]. In such systems, either a list of objects is input and the system verifies if the objects are consistent together, or some information is input and its consequences are calculated. Other applications essentially use description logics to organize instances or classes of objects. Given some previously defined classes of interest and a description of a new object, the application automatically determines the relationships between the new object and the pre-existing classes. This approach has been used, for example, to automatically classify web pages [73].

Description logics have also been used in a database context for several tasks. One task has been expressing and organizing queries [26, 32]. This is especially useful in situations where there are many queries, a number of which might be re-used, such as data-mining applications. Another task has been describing the integrated schema of a federated database, thereby hiding details from the user on where and how the information is obtained from the database [76, 75, 109, 11, 5].

Description Logics have also been used for modelling domain knowledge. Some projects find a deep background ontology extremely useful [96, 58, 86]. Other projects use DLs solely to support the knowledge engineering stage of a project when constant changes to the knowledgebase occur (and during the phase when knowledge engineers need to question the implications of their changes most often).

In this thesis, we explore a variety of issues involved in explaining reasoning in description logic-based systems. One fundamental obstacle to explanations in these systems is that although their specification may be declarative, typically their implementation is highly optimized procedural code, needed for application performance. This means that a trace of code execution as the basis of explanations is out of the question. Note that other reasoning systems, such as those solving constraint-satisfaction problems, have similar properties. Next, we will point out four general problems to be considered in any such reasoning system, very briefly describe our solution in general terms, and then mention how this is instantiated for description logics.

The first problem we address is how to explain the deductions of an implemented system that has a procedural implementation. Since we can not sacrifice efficiency and force the implemented system to be more transparent, we provide a declarative, proof-theoretic view of the inference engine. As a result of the proof theoretic representation of inferences, we may view explanations as proofs and thus provide a declarative basis for explanations.

A second problem that arises is how to explain conclusions that may have been reached by a long and complicated series of deductions. Given our view that explanations are proofs, this problem becomes one of finding a way to provide manageable, meaningful *fragments* of proofs. Our approach essentially involves three steps: (i) A technique for breaking up complicated questions into a series of simpler questions, thus enabling a series of simpler answers. (ii) A way of presenting simple, single step answers that are meaningful in isolation. (iii) An automatic technique for generating the appropriate follow-up questions.

A third problem considered in the thesis is the issue of explaining "negative" conclusions – the failure to deduce certain conclusions. Rather than attempting to show why no proof could be found, we follow a more constructive method of justifying a negative conclusion: providing a counter-example. Our concern is with families of implementations that are not refutation-based (and hence do not produce counter-examples as part of their reasoning). We propose a way of exploiting the existing underlying implementation and the preceding explanation work to generate counter-examples.

A final problem addressed is the issue of how to present and explain manageable amounts of information about very large, complicated objects. We offer a declarative language for noting the "interesting" aspects of objects, and then describe how to "match" these descriptions against objects to limit the amount of information being presented to users.

We propose to address the above problems for a class of description logics. For now, a description can be thought of as simply a noun phrase, for example, "persons who have two or more children" or "persons all of whose children have Ph.D.s". The fundamental deduction involving descriptions is whether one implies or is more general than another. For example, "persons who have three children, all of whom have Ph.D.s" is implied by (subsumed by) the earlier "persons who have two or more children".

We exemplify our approach with an implementation for the CLASSIC knowledge representation system[1] (which is very similar to the specification of the "core" description logic [101]). CLASSIC is an example of a description logic reasoner that determines relationships between descriptions by first "normalizing" descriptions, and then comparing their resulting canonical form. (This is in contrast to other kinds of description reasoners, e.g., KRIS [8], which use more standard theorem-proving techniques such as tableaux.)

[1]Throughout the thesis, we sometimes refer to this simply as CLASSIC.

The first problem we consider – that of of providing a declarative foundation for explanations – is addressed with a proof theoretic representation of all inferences performed by CLASSIC. The second problem – that of handling long and complicated deductions – is addressed first by rewriting complicated questions as a conjunction of simpler ("atomic") questions and then explaining the simpler questions with a chain of simple answers based on single inference rule applications. We use the logical form of the inference rule to automatically generate any appropriate followup questions.

The third problem – explaining negative conclusions – translates in the case of description logics to explaining why one description is *not* implied by another description. Our counter-example-based appropach generates an instance of the alleged more specific description that is not an instance of the alleged, more general description. The form of that instance allows one to explain very easily why that individual meets the conditions of one description but not of the other.

The final problem – handling large, complicated objects – is handled by providing a language for declaring "interestingness" of aspects of objects. The "interesting" information is simply meta-information about the objects. We begin with a core description logic language and augment it so that users can write "concept patterns" that may be matched against the objects in the system. Only the portions of the objects that match the pattern will be printed or explained.

In this thesis, we focus on providing the logical foundation for explanations and the supporting infrastructure. Our goal is to provide a knowledge engineer or application developer with the information and tools they need to write an application that is well suited to their users. Thus, this thesis only peripherally addresses the final presentation mechanism, be it natural language, graphical depictions, or other presentation schemes.

At this point, we would like to briefly introduce description logics and the reasoning performed by them, motivate the problem of explaining their deductions, and outline the fundamental ideas of our proposed solutions. We propose to accomplish these tasks by using some small illustrative examples.

1.1 Reasoning with Description Logics: an Example

Description Logics form a family of formalisms for representing and reasoning with knowledge. The three fundamental notions of DLs are *individuals,* representing objects in the domain, *concepts,* describing sets of individuals, and *roles,* binary relations between individuals. Concepts are specified by conjoining previously defined concepts (which become its superconcepts) with any additional restrictions that must be true for all instances of the concept. For example, a concept `BOOKSHELF-SYS` may be defined as having the superconcept `AUDIO-SYS` and additional restrictions stating that if there are any `main-speakers`, they must be instances of the concept `SMALL`, and if there are any `subwoofers`, they must also be `SMALL`. These restrictions on `main-speakers` and `subwoofers` are called value restrictions since they restrict the value of the *filler of a role.* Another type of restriction is a number restriction, which provides a lower or upper bound on the *number* of fillers of a role. For example, we could have defined a concept that had *at least two* `main-speakers`. We can use the following form to show the information that has been *told* to the system about a concept or object:

---

concept: `BOOKSHELF-SYS`

---

superconcepts: `AUDIO-SYS`

restrictions

on roles: `main-speaker`

value restriction: `SMALL`

`subwoofer`

value restriction: `SMALL`

Another concept, `HIGH-QUAL-SYS`, may be defined as an `AUDIO-SYS` whose `main--speakers` have a minimum `price` of 300 dollars and whose `subwoofers` are `BIG`.

---

concept: `HIGH-QUAL-SYS`

---

superconcepts: `AUDIO-SYS`

restrictions

on roles: `main-speaker`

restrictions

on roles: `price`:

value

restriction: > 300

`subwoofer`

value

restriction: `BIG`

An individual is described by asserting that it is an instance of some previously defined concepts, and stating the additional restrictions that are true of this individual, especially fillers for roles.

For example, we may decide to consider two systems: `S0`, which is a `BOOKSHELF-SYS` whose `main-speaker`s cost less than 600 dollars and `S1`, which is both a `BOOKSHELF-SYS` and a `HIGH-QUAL-SYS`. These look as follows:

---

individual: `S0`

---

superconcepts: `BOOKSHELF-SYS`

restrictions

on roles: `main-speaker`

restrictions

on roles: `price`:

value

restriction: < 600

individual: `S1`

superconcepts: `BOOKSHELF-SYS`
`HIGH-QUAL-SYS`

The pictures presented above only include the information that has been explicitly told to the system. DLs endeavor to find information that is implicit in these definitions and assertions, and thus, they may *derive* additional information. One way of doing this is to gather together related information about an object, check for interactions and implications, and produce a "normal form" that includes both the told and derived information. This process is called normalization. For example, the normal form for a role restriction will contain the minimum number of fillers, the maximum number of fillers, and a representation of any value restrictions on its fillers. Role restrictions may also be nested, e.g., a `HIGH-QUAL-SYS` has a role restriction on `main-speaker` which has a role restriction on `price`.

The restrictions on the `main-speaker` role on `BOOKSHELF-SYS` may be represented as follows:[2]

restrictions
on role: `main-speaker`

[2] Any system-initialized information in these presentations, for example an **at-least** restriction of 0, is marked by *.

at-least:	0*
at-most:	infinite*
value restriction:	`SMALL`

————————————————————————————————-

Information about `S0` is gathered from information from its superconcept, `BOOKSHELF-SYS` and the additional stated restriction. Information about its `main-speaker` role looks as follows:

————————————————————————————————-

restrictions
on role: `main-speaker`

————————————————————————————————-

at-least:	0*		
at-most:	infinite*		
value restriction:	`SMALL`		
restrictions on roles:	`price`		
	at-least:	0*	
	at-most:	infinite*	
	value restriction:	< 600	

————————————————————————————————-

Similarly, for `S1`, information about its `main-speaker` role is shown below; this is gathered from restrictions on the `main-speaker` role in both of `S1`'s superconcepts.

————————————————————————————————-

restrictions
 on role: `main-speaker`

---

at-least:	0*		
at-most:	infinite*		
value restriction:	`SMALL`		
restrictions on roles:	`price`		
	at-least:	0*	
	at-most:	infinite*	
	value restriction:	> 300	

For the `subwoofer` role on `S1`, there is more work to do. First, the DL collects the information that subwoofers are both `BIG` and `SMALL`:

---

restrictions
 on role: `subwoofer`

---

at-least:	0*
at-most:	infinite*
value: restrictions:	`SMALL`
	`BIG`

If it is known that `BIG` and `SMALL` are disjoint concepts, the DL can further deduce that the value restriction is incoherent and thus the value restriction should be changed to point to a representation for incoherent concepts, which we will call `NOTHING`. Further, since there can be no instances of `NOTHING`, the **at-most** restriction should be changed to be zero. The new role information follows:

restrictions on role:	subwoofer	
	at-least:	0*
	at-most:	0
	value restriction:	NOTHING

After the normalization process is completed, the DL will decide how this object relates to objects already in the knowledge base. DLs maintain generalization hierarchies which place more general concepts such as `AUDIO-SYS` above more specific concepts like `BOOKSHELF-SYS`. Also, individuals are placed below concepts of which they are known to be instances. In this example, `S1` would be below both `HIGH-QUAL-SYS` and `BOOKSHELF-SYS`. If, in addition, we defined a concept `NO-MORE-THAN-1-SUBWOOF` as an `AUDIO-SYS` that has at most 1 subwoofer, then `S1` would also be below `NO-MORE--THAN-1-SUBWOOF`. This hierarchy is shown in Figure 1.1.

One method of determining when one object belongs below another object in the hierarchy uses a comparison of normalized forms: If an object A has a normalized form that contains everything (and usually more) than the normalized form of concept B, then A is at least as specific as B and it belongs below B in the hierarchy. If A is a concept, A is said to be *subsumed by* B. If A is an individual, A is said to be an *instance*

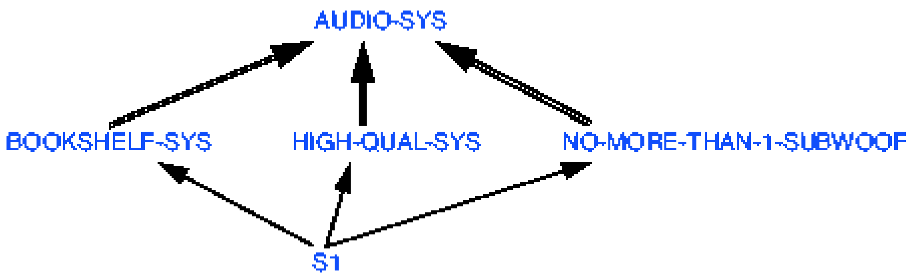

Figure 1.1: **A Simple Hierarchy**

of B.[3]

1.2 Explaining Description Logics: an Example

In knowledge base design or in later use, users may need to have the normalization deductions explained. For example, they may need to know how the DL determined that `S1` has an **at-most** restriction of zero. Additionally, they may need to know how the DL decided where to place an object in the generalization hierarchy (i.e., why one concept can be proven to be more general than another object). For example, users may need to know why `NO-MORE-THAN-1-SUBWOOF` is subsumed by `AUDIO-SYS` or why `S1` is an instance of `NO-MORE-THAN-1-SUBWOOF`.

A simple trace of the destructive data structure manipulations that were used to determine the normal form for `S1` would likely be incomprehensible because, in general, the causal structure would be missing. Instead, we propose a declarative statement of inferences that produced the result. In the case of `S1`, we might explain that the

[3]A precise definition of subsumption and instance recognition will be given in Sections 2.3 and 2.5.

at-most 0 restriction appears because of the incoherent (`NOTHING`) value restriction. In turn, the value restriction is incoherent because `SMALL` and `BIG` are disjoint.

To do this, this thesis proposes that we rely on a proof-theoretic characterization of the deductions sanctioned by the language, such as the deductive system introduced in [12]. For example, we might use inference rules such as

> *IF* a concept has value restriction `NOTHING` on role ?R
> *THEN* the concept has **at-most** 0 fillers on ?R.

Let us call this the INCOHERENT-VALUE-RESTRICTION rule.

When an "explanation switch" is turned on, the DL can augment its data structures to show how it deduced each piece of information.[4] For example, if explanation augmented the data structure with the inference rule used to determine that **at-most** 0 restriction on subwoofer, the following structure would result:

restrictions on role:	`subwoofer`		deduced by
	at-least:	0*	
	at-most:	0	INCOHERENT-VALUE-RESTRICTION
	value: restriction:	`NOTHING`	

Another rule, which can be named DISJOINT-PRIMS, says that

> *IF* C has two primitive superconcepts ?A and ?B that are disjoint,
> *THEN* C is incoherent.

If the DL was asked to augment its data structure to include an explanation of the incoherent value restriction on `subwoofer`, the augmented data structure would look like:

[4]At this point, the reader may think of the explanation procedure as a separate process, however later we will show our choice which is to use a low-overhead method for incorporating the explanation structure updates into the core inference engine.

rolename:	`subwoofer`	deduced by:
at-least:	0	
at-most:	0	INCOHERENT-VALUE-RESTRICTION
value:		
restriction:	`NOTHING`	DISJOINT-PRIMS: `SMALL, BIG`

The simplest kind of inference rule in a DL is a "told" rule that says: "If the user has input information directly, then the information is true". Thus, if we asked a DL why it has `SMALL` and `BIG` in the value restriction on `subwoofer`,[5] we would get:

rolename:	`subwoofer`	deduced by:
at-least:	0	
at-most:	0	INCOHERENT-VALUE-RESTRICTION
value:		
restrictions:	`NOTHING`	DISJOINT-PRIMS: `SMALL, BIG`
	`SMALL`	TOLD
	`BIG`	TOLD

Given an appropriate set of inference rules for the DL, the kind of augmentation of the data structures illustrated above can be carried out by an algorithm knowledgeable about what questions should be asked in response to discovering that a particular rule

[5] `SMALL` and `BIG` now show up in the value restriction field since they were specifically asked about. The previous printings of the value restriction field did not show them since `NOTHING` was more specific than each of them and thus superseded them.

was used. For example, if a piece of information is deduced by an application of the INCOHERENT-VALUE-RESTRICTION rule, one also needs to explain how the incoherence was determined. Or, if the DISJOINT-PRIMS rule was used, one also needs to explain where each of the disjoint primitives came from.

The final data structures capture the information required to reconstruct the reasoning process. We do **not** however advocate presenting these structures as explanations themselves. Instead, our goal is to provide the knowledge engineer with enough information from which to build an explanation appropriate for the user. We have experimented with presenting explanations that are basically pretty printed versions of the structures, natural language translations of them, or interactive graphical presentations of the information.

Positive subsumption explanations are only half of a complete job of explaining subsumption. Users also require explanations of negative subsumption relationships. One class is not a subclass of another class if it is *not* the case that every instance of the alleged subsumee class is an instance of the alleged subsumer class. For example, users may need an explanation of why `BOOKSHELF-SYS` is not a subclass of `HIGH-QUAL-SYS`. If the underlying system can produce a counterexample individual, then an explanation can refer to this individual. For example, `BOOKSHELF-SYS` can not be a subclass of `HIGH-QUAL-SYS` if `I1` is an instance of `BOOKSHELF-SYS` but `I1` is not an instance of `HIGH-QUAL-SYS`. Thus, an explanation can be formed using a counterexample individual.

In this case, if `I1` has no `subwoofer` and has (exactly) two `main-speakers` `M1` and `M2` and `M1` and `M2` are both `SMALL` and cost 100 dollars each, then `I1` is an instance of `BOOKSHELF-SYS` but it is not a `HIGH-QUAL-SYS` since the price restriction on the `main-speaker` role is not satisfied. This thesis provides a mechanism for generating instances of classes and a procedure for turning such an instance into a counterexample instance if it is possible.

An issue related to explanation and debugging is the ability to view the information maintained in a DL. For example, consider an individual `S2` that is asserted to be a

HIGH-QUAL-SYS with a subwoofer. S2's told information is shown below:

---

individual:	S2	
superconcepts:	HIGH-QUAL-SYS	
restrictions on roles:	subwoofer	
	at-least: 1	

Normalizing this individual will cause expansion of HIGH-QUAL-SYS as well as AUDIO--SYS. AUDIO-SYS might have quite a bit of information in it. For example, in an implemented graphical demonstration application [91] that configures stereo systems, basic concepts like AUDIO-SYS are quite large. These concepts contain typical domain knowledge including, for example, that a stereo system has at least one amplifying device, but they also contain some peripheral information relating, for example, how a stereo system should be graphically displayed in a rack output. In that implementation, a simple unfiltered printing of normalized stereo systems can take ten screens. An unfiltered printing of the normalized structures annotated with the inference rules that justify each piece of information is even more unwieldy. Even in this simple example, S2 could look as follows:

individual:	S2	deduced by
primitive superconcepts:	AUDIO-SYS	INHERITANCE: HIGH-QUAL-SYS
	(many other prims)	INHERITANCE: AUDIO-SYS

restrictions on roles:	`main-speaker` restrictions on roles:	`price`: value restriction: $>$ 300	INHERITANCE: `HIGH-QUAL-SYS`
	`subwoofer` at-least: 1		TOLD
	graphical-role-1 at-least: ...		INHERITANCE: `AUDIO-SYS`
	at-most: ...		INHERITANCE: `AUDIO-SYS`
	value restriction:	...	INHERITANCE: `AUDIO-SYS`
	restrictions on roles:	...	INHERITANCE: `AUDIO-SYS`
	(many other roles): ...	...	INHERITANCE: `AUDIO-SYS`

To deal with this problem, the presentation mechanism ought to accept a specification of information that should be printed or explained (or information that should be ignored). In a simple form, users could specify which roles are "interesting" (e.g., role `main-speaker`), and determine that graphical roles like `graphical-role-1` are "uninteresting" by omission. The system would then only print and explain information about interesting roles. In addition, users may want context to help determine what is interesting. For example, they may be interested in the specific price if it is known for the `main-speaker`. If it is not known, they may want to see the price range. In other contexts, they may only want to see the price range if it is more specific than the price range for speakers of a normal `HIGH-QUAL-SYS`, i.e., if the price range is more specific than > 300.

This thesis also provides a declarative language for specifying patterns that can be used to limit what is printed/explained. For example, we can say the equivalent of:

"On subclasses and instances of concept ____, show a value restriction of role ___ if ____" and

"Show filler of role ___ if ____"

Using a limited version of the pattern language to be presented in this thesis, we pruned the ten screen printouts in the stereo configurator to one or two screens.

1.3 Development History

We have implemented an explanation system based on the ideas above for the CLASSIC Knowledge Representation and Reasoning System (KRS). The system provides declarative explanations for all of the inferences done in the KRS. It can provide step by step explanations or explanations of entire chains of deductions. Additionally, explanations and presentations of objects can be filtered, thus limiting computation and output.

Although there has been no full-scale experimental evaluation of the utility of the explanation facility, the following remarks on the history of our development are intended to buttress the claim that this thesis is not just an "academic exercise", possibly devoid of any practical utility or based on unfounded intuitions.

Our research on explanation has been driven by the demands of knowledge engineers and users. The original motivation was provided by CLASSIC's first industrial configuration applications [130]. The knowledge engineers required explanations of reasoning in order for them to build and maintain their applications. The present author served as a "human explanation facility" while the group identified and refined a requirement specification for an automated facility. This exercise also showed that even description logic experts needed automated support in order to provide explanations quickly when knowledge bases were very large and complicated. As a result, we implemented a core explanation facility that has been in use for several years.

We have interviewed help desk staff, users of CLASSIC, users of other DLs, and knowledge engineers in order to gather data for refining and expanding our facility. We wrote a tutorial exercise [7] on CLASSIC's explanation facilities for use in a university course on knowledge representation and gathered student feedback. We also wrote two demonstration applications [91, 92, 7, 24] and presented this work and gathered feedback on numerous occasions.

An explanation system has been implemented in the research version of CLASSIC [103, 16]. A limited subset has been integrated into the development version of CLASSIC [125]; and our basic design is currently being integrated into the new (C++) version called NEOCLASSIC [99].

1.4 Thesis Outline

This thesis is a discussion of the underlying principles of the explanation and filtering facilities developed for CLASSIC, *generalized* to apply to other Description Logic-based systems implemented in the normalize/compare paradigm.

The rest of the thesis proceeds as follows: In Chapter 2, we provide an introduction to Description Logic-based systems. The principal ideas — concepts, individuals, and roles — are defined and examples are given. We present the formal syntax and semantics of CLASSIC, the particular description logic based system that we use for our examples. We introduce the main kinds of reasoning in DLs, dealing with concepts and individuals, and we provide technical details of the computations involved.

In Chapter 3, we lay the foundation of the thesis. Remembering that subsumption is the core inference in DLs, we start by presenting a theory for explaining subsumption deductions. Explanations are equated with proofs, and therefore we present a proof theory for DLs. Noting that proofs quickly become too long, we propose to present explanations as proof fragments. Our method relies on the notions of atomic description – which provides a way to simplify the questions asked during explanation, and atomic justification — which provides a way to simplify explanations themselves. We categorize inference rules into a variety of classes, showing how the logical form of these rules guides the automatic generation of follow-up questions. We include a section on implementation perspectives gathered from our experience implementing our approach in three instantiations of CLASSIC and from working with numerous DL users.

In Chapter 4 we focus on individual reasoning. Much of the foundation has already been laid with the concept discussion so this chapter concerns itself with the additional reasoning involved with individuals. This chapter also includes the work on recognizing and explaining errors, since, empirically, most errors arise in handling individuals and the additional reasoning concerning individuals affects error explanations. One interesting point is that with one simple notion – that of an "intermediate object" capturing an intermediate state of an object – our earlier introduced explanation mechanism naturally handles explanations of errors. We also discuss the importance and additional support for following up error explanations.

In Chapter 5, we motivate and deal with the problem of explaining non-subsumption. In addition to a simple technique, which just points to some unsatisfied atomic description, we investigate a more effective solution, which rests on the observation that if `A` is not subsumed by `B`, then it is possible to create a counter-example: an instance of `A` that is not an instance of `B`. We describe an approach for generating such a counter-example and provide an algorithm for generating such an individual for CLASSIC. An added advantage of this technique, is that it can help deal with incompleteness of the DL in the subsumption process.

In Chapter 6, we tackle a somewhat orthogonal issue. Objects are typically quite complex in DLs and explanations of complex objects may be unwieldy. We introduce a language for describing "interesting" aspects of objects for presentation and for explanation. This language is an extension of the base DL language, which incorporates meta information (information about an object) and epistemic information (knowledge about the contents of the knowledge base). We provide an algorithm for matching descriptions in the expanded language against CLASSIC descriptions. We also provide a number of examples showing the usefulness of such descriptions.

In Chapter 7 we discuss work that is related to ours. We include work on explaining expert systems, logic programming, deductive databases, theorem proving systems, and DLs themselves.

In the final chapter, we review the ideas presented and support claims about the generality of our approach by showing how it can be extended to deal with a new concept constructor.

Chapter 2

Description Logics

Description Logics, (e.g., BACK [120], CLASSIC [16], KRIS [8], LOOM [82]) are a family of formalisms for representing and reasoning with knowledge, surveyed, among others, in [79, 127, 13]. They are well suited for representing data and knowledge concerning individual objects that may be grouped into classes (which description logics call concepts) and are interconnected by binary relationships (roles).

For example, an individual, `My-stereo`, may be related by the role `price` to the number 1200, and by the role `speaker` to two other individuals, `Spk34` and `Spk66` (which are called the fillers for the `speaker` role on `My-stereo`). By convention, we use initial capitalization for individuals, all lower case for roles, and all uppercase for concepts. `My-stereo` would also be an instance of other concepts in the domain such as `STEREO-SYSTEM` and `EXPENSIVE-THING`.

Composite descriptions for concepts and sometimes roles may be formed using *description constructors* chosen from a pre-specified set for the particular DL. For example, a simple description of `My-stereo` is:

(**and** `STEREO-SYSTEM EXPENSIVE-THING`).

This description simply used the conjunction constructor **and** to join two concepts. Another description:

(**and** (**at-least** 2 `speaker`) (**at-most** 4 `speaker`))

is the conjunction of things that have 2 or more fillers for their `speaker` role and 4 or fewer fillers for the same role.

A description reasoner then makes deductions from a knowledge base of descriptions, possibly attributed to individuals, following the semantics of the constructors. Among other things, the DL detects various kinds of inconsistencies, organizes the concepts into a sub-class hierarchy, and classifies individuals under appropriate concepts. We will introduce description logic constructors and reasoning by example, using the syntax and semantics of the CLASSIC knowledge representation system. Most of the examples are drawn from either our application used to configure stereo equipment [91, 92] or one used to match foods and wines [24, 7].

In this chapter, we will introduce the various aspects of a typical DL knowledge base, including: the different kinds of elements appearing in it, the operations that can be performed on it, and the new information that can be derived from it. We first describe the concept constructors of the CLASSIC DL, as well as some additional useful ones; this is done initially by the use of examples and then with a formal syntax and semantics. Next, we examine the main deduction that is done using concepts — subsumption. Similarly, we present the notions relating to individuals and their principal associated deduction — recognition. After discussing some of the other kinds of objects (e.g., rules) and inferences (e.g., so-called propagations), we conclude by providing a fairly detailed look at techniques for computing subsumption and recognition.

The basic definitions of DL objects, subsumption, and recognition are required for reading most of the thesis. Sections 2.7 and 2.8 are of more restricted utility, and we begin each by mentioning the later sections for which they are required or useful.

2.1 Concepts

Concepts denote classes of objects. A concept may be either a *primitive*, whose instances are determined entirely by the user, or it may be *defined* by giving necessary and sufficient conditions for membership. Primitive concepts behave similarly to frames in many frame based systems, since the knowledge engineer states where a concept fits in a generalization hierarchy and also states what individuals are instances of this concept. For example, we can declare that `MAMMAL` is a primitive concept name, so its instances will be specified either directly, or by specifying instances of some subclass of `MAMMAL`. Likewise, we will wish to say that `PERSON` is a primitive concept name, which is more specific than `MAMMAL`. Primitive concepts will be introduced to the system by stating an expression of the form:

$$< \text{identifier} > \ \sqsubset \ < \text{necessary-conditions} >.$$

Thus, we would introduce PERSON using the expression PERSON $\sqsubset$ MAMMAL. Note that this provides a necessary condition for being a PERSON: namely, being an instance of MAMMAL.

If there is no specific necessary condition that we wish to express, we can use the concept called THING, which denotes the most general concept in the hierarchy. For example, we could introduce MAMMAL by stating that MAMMAL $\sqsubset$ THING.

In addition to THING, which is at the top of the concept hierarchy, DLs often have a name for an incoherent/bottom concept, typically called either BOTTOM or NOTHING.[1]

Typically primitive concepts are used to model "natural kinds" like MAMMAL and PERSON, which usually do not have "definitions". One differentiating feature of description logics is that they also allow users to *define* concepts using necessary *and* sufficient conditions. We will introduce defined concepts in the form

$$< \text{identifier} > \equiv < \text{necessary and sufficient conditions} >^2.$$

For example, a stereo system may be defined to be a "system" that has at least two speakers. The CLASSIC definition for this will be:

STEREO-SYS $\equiv$ (**and** SYSTEM (**at-least** 2 speaker)).

This states that STEREO-SYS is the intersection of individuals in two descriptions: SYSTEM (an identifier presumably introduced earlier) and the set of objects that have two or more fillers for the speaker role. A more complete definition might also include the restriction that every filler of the speaker role must be an instance of the concept SPEAKER. Thus, our final definition for STEREO-SYS could be:

---

[1]Some DLs, e.g., LOOM [82], have more than one bottom concept but for simplicity of presentation, we will only use one bottom concept as in CLASSIC.

[2]In CLASSIC, all identifiers mentioned in these conditions must have been introduced previously, thus eliminating the possibility of introducing recursive concepts. Thus, the definition PERSON $\equiv$ (**all** child PERSON) is forbidden.

```
STEREO-SYS ≡ (and SYSTEM
                 (at-least 2 speaker)
                 (all speaker SPEAKER)).
```

This definition can be thought of as a *term*, built up from the concept identifier `SYSTEM`, the role identifier `speaker`, and the *concept constructors* **and**, **at-least**, and **all**. In general, roles, such as `speaker`, are binary relationships between individuals: one individual may be related by the same role to many individuals. (For example, an individual `STEREO-SYS`, S1, is known to have at least two fillers for its `speaker` role, so that role will relate S1 to at least two other individuals.) Many DLs have a special type of relationship denoting a single-valued role like `manufacturer`. CLASSIC calls such relationships *attributes*.

Some systems allow roles to be also composite (e.g., (**inverse** `manufacturer`)). In this thesis we limit our discussion to primitive roles — roles which provide neither necessary nor sufficiency conditions for the relationship to hold.[3]

A *terminological knowledge base* consists of role and attribute names, defined concept names and their definitions, and primitive concept names and their necessary conditions. (The last two will be collectively referred to as *concept declarations.*)

Resuming the introduction of CLASSIC concept constructors, note that the **all** restriction is a recursive concept constructor. For example, a user may want to say that anything that fills the `speaker` role must have a manufacturer that is a `US-COMPANY`. This can be stated as:

(**all** `speaker` (**all** `manufacturer` `US-COMPANY`)).

In addition to *value restrictions*, expressed using **all**, CLASSIC allows the expression of *number restrictions*. The **at-least** and **at-most** constructors are used to place a minimum and maximum on the number of fillers of a role. For example, a "normal" home theater setup may include no more than two VCRs, expressed as the description

(**at-most** 2 `vcr`).

---

[3] CLASSIC 2.0 and beyond supports role hierarchies, which allow necessary (but not sufficient) conditions on roles. Inferences concerning role hierarchies are included in our implementation and are explained in appendices A and B.

Another kind of a restriction, sometimes called a co-reference constraint or a "same-as" constraint, requires the same individual to fill two different roles or role chains. For example, it could be that we would like to consider stereo systems which have the same manufacturer for both the tuner and the preamplifier. The CLASSIC term for this is:

(**same-as** (`tuner manufacturer`) (`preamplifier manufacturer`)).

Any object satisfying this description must have fillers `x` and `y` for the `tuner` and `preamplifier` attributes, respectively, such that the fillers of `manufacturer` for both `x` and `y` are the same object, `z`.

Sometimes it is useful to refer to particular individuals in descriptions. For example, some people may want a television that is made by a particular manufacturer - for example, `Sony`. Thus, they would want their television to satisfy the restriction:

(**fills** `manufacturer Sony`).

The **one-of** constructor can be used to describe a concept by explicit enumeration of the individuals in it. For example, someone may want their home theater system to contain only `tv`s made by `Sony` or `Toshiba`:

(**all** `tv` (**all** `manufacturer` (**one-of** `Sony Toshiba`))).

If a filler is known to be a number[4], it can be useful to put a numerical restriction on it. For example, it could be that a consumer has a particular price range in mind for speakers. This can be expressed using the **min** and **max** constructors as follows:

(**all** `speaker` (**all** `price` (**and** (**min** 200) (**max** 500)))).

This says that anything that fills the `speaker` role must have a price value that is no lower than 200 and no higher than 500.

For notational convenience, we also include the **prim** constructor. This allows us to indicate directly in a description when some identifier (e.g., PERSON) corresponds to a primitive concept, as opposed to a defined one, by using the term (**prim** PERSON); it saves us from having to always refer to a separate knowledge base to check the declaration of identifiers.

---

[4]Many DLs, including CLASSIC, have some predefined host object classes such as numbers and strings.

This is an appropriate point to discuss the role of defined and primitive concepts in a DL system. Since concepts cannot be recursive in CLASSIC (i.e., concept declarations may not refer to themselves directly or indirectly), it is possible to treat identifiers as macro definitions and apply some form of macro expansion. In the case of definitions introduced with <Id> $\equiv$ <necessary and sufficient conditions>, this is exactly what happens – whenever <Id> is encountered, it is replaced with <necessary and sufficient conditions>. In the case of primitive concepts, they must be supplemented by the necessary conditions in the declaration. Thus, if the knowledge bases contains the declaration `PERSON` $\sqsubset$ `MAMMAL`, then (**prim** `PERSON`) is expanded to

(**and** (**prim** `PERSON`) (**prim** `MAMMAL`))

which in turn might lead to

(**and** (**prim** `PERSON`) (**prim** `MAMMAL`) (**prim** `THING`))

when (**prim** `MAMMAL`) is expanded. The primitives can be grouped together to yield a final rewriting of (**prim** `PERSON MAMMAL THING`).

Many DLs allow roles to be filled by objects from the host programming language such as numbers and strings. In CLASSIC, these objects are said to be in the host realm, while other objects are in the CLASSIC realm. To distinguish between elements of each domain, CLASSIC also has built-in concepts `CLASSIC-THING` $\sqsubset$ `THING` and `HOST-THING` $\sqsubset$ `THING`. `CLASSIC-THING` and `HOST-THING` are disjoint, i.e., it is impossible to find an individual that is an instance of both concepts. Only CLASSIC individuals may have roles, hence only CLASSIC concepts may have role restrictions such as (**all** speaker `SPEAKER`). The constructors **min** and **max** only apply in the host realm to objects that are `NUMBER`s. CLASSIC has built-in concepts for the host types; in this thesis, we will only be using the host subclass `NUMBER` $\sqsubset$ `HOST-THING`. All user-defined concepts and all individuals belong to either `CLASSIC-THING` or `HOST-THING`, but not both; thus `CLASSIC-THING` and `HOST-THING` are disjoint.

The specific set of constructors chosen for a particular language is usually a compromise between the expressive needs of the users and concerns of computational efficiency. CLASSIC has chosen to limit the language of constructors in an effort to maintain efficiency. For example, it does not include "qualified number restriction", so there is no way to express the class of objects that have at least one `speaker` that is a `SUBWOOFER`. Other DLs include a constructor such as **at-least-C**, and could express the above concept simply as (**at-least-C** 1 `speaker SUBWOOFER`). (The computational intractability of reasoning with **at-least-C** was first shown in [94].)

CLASSIC has included one additional constructor called **test**[5] which serves as a "procedural escape hatch". If one cannot represent some concept using the other CLASSIC constructors, one can write a host language function that will test to see if an object satisfies some condition. For example, to describe the class of prime numbers, one could introduce the term (**test** primeCheck), where primeCheck is some unary LISP or C function that checks individual numbers for being prime. Such test concepts are treated as primitive "black boxes" for the purposes of reasoning about concepts, though, as we shall see, they do have an effect on individuals.

More generally, if one needed to express **at-least-C**, one could write a function atLeastTest that took three additional arguments: n (a number), r (a role) and D (a description), and then tested to see if some given object had n or more fillers of the r role that were instances of D. This would allow the application to recognize with the description

(**test** atLeastTest 1 `speaker SUBWOOFER`)

objects satisfying the non-CLASSIC description (**at-least-C** 1 `speaker SUBWOOFER`).

2.2 Formal Syntax and Semantics of CLASSIC Concepts

A more formal treatment of the meaning of DL constructors relies on denotational semantics. We begin with an interpretation I that consists of a domain of values Δ^I and an interpretation function $.^I$. The function $.^I$ maps every concept to a subset of Δ^I and maps every role and attribute to a subset of $\Delta^I \times \Delta^I$. In CLASSIC, the domain is separated into Δ^I_H, for the host realm and Δ^I_C, for the CLASSIC realm objects.

The extensions are defined as follows:

[5]In implementation, this constructor is broken into **testc** and **testh** for tests on the CLASSIC and host realm. For simplicity of presentation, **test** will cover both constructors.

CONCEPT/ROLE	DENOTATION
THING	$\Delta^{\mathcal{I}}$
CLASSIC-THING	$\Delta^{\mathcal{I}}_C$
HOST-THING	$\Delta^{\mathcal{I}}_H$
NOTHING	$\emptyset$
(**prim** ID)	$\mathtt{ID}^{\mathcal{I}}$
(**and** C D)	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
(**all** p C)	$\{d \in \Delta^{\mathcal{I}}_C \mid p^{\mathcal{I}}(d) \subseteq C^{\mathcal{I}}\}$
(**at-least** n p)	$\{d \in \Delta^{\mathcal{I}}_C \mid \lvert p^{\mathcal{I}}(d)\rvert \geq n\}$
(**at-most** n p)	$\{d \in \Delta^{\mathcal{I}}_C \mid \lvert p^{\mathcal{I}}(d)\rvert \leq n\}$
(**same-as** p_1 ... p_n q_1 ... q_m)	$\{d \in \Delta^{\mathcal{I}}_C \mid p^{\mathcal{I}}_n ... p^{\mathcal{I}}_1(d) = q^{\mathcal{I}}_m ... q^{\mathcal{I}}_1(d)\}$
(**fills** p B_1,...,B_m)	$\{d \in \Delta^{\mathcal{I}}_C \mid B^{\mathcal{I}}_j \in p^{\mathcal{I}}(d)\}$
(**test** fn arg_1,...,arg_m)	$\{d \in \Delta^{\mathcal{I}} \mid fn(d, arg_1, ..., arg_m) = True\}$
(**one-of** B_1,...,B_m)	$\{B^{\mathcal{I}}_1, \ldots, B^{\mathcal{I}}_m\}$
(**max** n)	$\{d \in \Delta^{\mathcal{I}}_H \mid d \leq n\}$
(**min** n)	$\{d \in \Delta^{\mathcal{I}}_H \mid d \geq n\}$

Table 2.1: Syntax and Denotational Semantics of Description Constructors

- the extension of a role r is a subset r^I of $\Delta^I_C \times \Delta^I$.

- the extension of an attribute a is a function a^I from Δ^I_C to Δ^I.

- the extension of a concept named D, is a subset D^I of Δ^I.

The denotation of composite descriptions is obtained by extending the interpretation function according to Table 2.1, which summarizes the syntax and semantics of CLASSIC. In the table, n represents a positive integer, C and D represent concepts, p and q represent roles, B represents an individual, and fn represents a function.

Given a knowledge base, normally one considers only those interpretations which satisfy the condition that $\mathtt{ID}^I = \mathtt{D}^I$ if $\mathtt{ID} \equiv \mathtt{D}$ is present, and $\mathtt{ID}^I \subseteq \mathtt{D}^I$ if $\mathtt{ID} \sqsubset \mathtt{D}$ is present. In CLASSIC, since there are no recursive definitions, it is sufficient to first expand names from the KB as described earlier, and then give denotations only for primitive concept names.

2.3 Concept Reasoning in CLASSIC

The main inference that DLs perform with respect to concepts is subsumption. One concept, A, *subsumes* another concept, B, if and only if A's extension is a superset of B's extension, i.e, if $A^I \supseteq B^I$, for every possible interpretation I. We will use $\Longrightarrow$ to represent "is subsumed by"[6] and write B $\Longrightarrow$ A stating that B is subsumed by A. For example, SYSTEM subsumes STEREO-SYS, and we write this as STEREO-SYS $\Longrightarrow$ SYSTEM.

DLs also perform other deductions, which in our case are reduced to subsumption:

1. *Incoherence*: A concept A is incoherent if it is impossible to create an individual that is an instance of A. This is equivalent to determining if A $\Longrightarrow$ NOTHING. For example, if A is defined as (**and** (**at-least** 4 speaker) (**at-most** 2 speaker)), then A is incoherent since it is impossible to find integer n such that $4 \leq n \leq 2$.

2. *Disjointness*: Two concepts, A and B, are disjoint if it impossible to create an individual that is an instance of both A and B. This is equivalent to determining that (**and** A B) $\Longrightarrow$ NOTHING. For example, if A = (**at-least** 4 speaker) and B = (**at-most** 2 speaker), then A and B are disjoint.

3. *Equivalence*: Two concepts A and B are equivalent if every individual that is an instance of A must be an instance of B and every individual that is an instance of B must be an instance of A. This is equivalent to determining A $\Longrightarrow$ B and B $\Longrightarrow$ A. For example, if A $\equiv$ (**fills** speaker S1 S2) and B $\equiv$ (**and** (**at-least** 2 speaker) (**fills** speaker S1 S2)), then A and B are equivalent.

4. *Classification*: Given a KB of concept declarations, these are organized in a hierarchy where for each concept A, we identify the most specific subsumers and the most general subsumees (and, inter alia, recognize synonyms).

It should be noted that all DL reasoners aim to make reasoning sound, but not necessarily complete. This is because the complexity of even the less expressive DLs

[6]Is subsumed by is also called "Isa" in some semantic networks and description logic literature.

is not polynomial time. (A detailed discussion of complexity of DL languages can be found in many papers in the literature, e.g. [48, 31]). CLASSIC, for example, has chosen to be more efficient and to avoid certain potentially exponential computations and is thus not complete with respect to the standard semantics[20]. Although individuals are allowed in concept declarations (using the **fills** or **one-of** constructor) properties of individuals are not taken into account during concept reasoning.[7] For example, if a concept `C` is defined as:

$$\texttt{C} \equiv (\textbf{and}\ (\textbf{at-most}\ 2\ \texttt{r})\ (\textbf{fills}\ \texttt{r}\ \ \texttt{I1}\ \ \texttt{I2}))$$

and if it is known that both `I1` and `I2` are instances of `D`, CLASSIC's normalization will not derive a value restriction on `C` of (**all** `r D`), and so the subsumption algorithm will miss that $\texttt{C} \implies (\textbf{all}\ \texttt{r}\ \ \texttt{D})$.

2.4 Individuals

Individuals represent the domain of discourse. Individuals may be asserted to be instances of concepts and have roles filled with individuals. For example, `T1` may be asserted to be an instance of a `TELEVISION` and it may have its `manufacturer` role filled by `Sony`. Formally, this means that the domain is extended so that $\texttt{T1}, \texttt{Sony} \in \Delta^I$ and $\{(\texttt{T1,Sony})\} \in \texttt{manufacturer}^I$. The conjunction of things asserted about the individual forms the description of the individual, which we will refer to as descr(Ind). For example, descr(`T1`) = (**and** `TELEVISION` (**fills** `manufacturer Sony`)).

Minimally, a DL must allow the user to specify that an individual is an instance of a primitive concept. However, one of the strengths of DLs is that they also allow assertions of "instancehood" in arbitrary descriptions, thus easily expressing incomplete information. For example, `SS1` can be asserted to be an instance of (**at-least** 2 `speaker`), without the system knowing the specific individuals related to `SS1` by the `speaker` role. To express the fact that an individual is an instance of some description, we will use the notation:

[7] The computational reason for this is to avoid reasoning by cases. The justification is that individuals can change in the world as time progresses but terminological definitions should not change with time.

< Individual-name > $\rightarrow$ < description >.

Asserting that individuals are instances of arbitrary descriptions requires the DL to move beyond the "closed world reasoning" used in databases, where one assumes that all fillers of a particular role are known. Thus, if `Alex` has his `child` role filled with `Michelle` and `Anna`, a system that uses closed world reasoning will deduce that `Alex` has exactly two children. If a DL was only told that `SS1` $\rightarrow$ (**at-least** 2 `speaker`) and then tried to use closed world reasoning, it would determine that since there are currently no known fillers for `SS1`'s `speaker` role, then `SS1` must have 0 `speakers`. This would violate the **at-least** restriction that has been asserted and the system would now have inconsistent information about `SS1`. Thus, DLs normally do not make the closed world assumption, assuming that additional fillers may be added to roles that do not already have the maximum number of fillers allowed. Therefore, `SS1` may obtain `speakers` in the future, so the system does not deduce that `SS1` has 0 `speakers`. Likewise, if all that is known about `Alex` is that he is an instance of (**fills** `child Anna Michelle`), the DL would not automatically deduce that he is an instance of (**at-most** 2 `child`).

Of course, in some situations, we will want to say that we do indeed know all the fillers for some role on an individual – there are no more to come. To do so, we can explicitly assert that `Alex` $\rightarrow$ (**at-most** 2 `child`), when `Alex` has 2 fillers for the `child` role. In this case, we will say that `Alex`'s `child` role is *closed*[8] since no additional fillers can be added to this role without causing a contradiction. Description logics may also include a close operator which takes an individual and a role and "closes" the role on the individual, by first counting the known fillers for the role on the individual and then asserting (**at-most** <count> <role>) on the individual.

2.5 Individual Reasoning

The analog of subsumption for individuals is *recognition*. An individual, `Ind`, is recognized to be an instance of a concept `C` if and only if `Ind`$\in$ $\mathtt{C}^I$ for all interpretations, I, that satisfy the facts in the knowledge base.

[8]Some DL literature refers to this role as being *full* since it has all the fillers that it can hold.

For example, consider the individual SS2, which is known to have S1 and S2 as fillers for its speaker role. Then it must have at least two fillers for the speaker role in all possible worlds, so SS2 would be recognized to be an instance of (**at-least** 2 speaker). The details of recognizing membership for concepts will be covered in Section 2.8.

In addition to recognition, there are a number of other inferences that are part of individual processing, including propagation, rule firing, test application, and inconsistency checking.

Propagation: Information is propagated onto an individual Ind if Ind is an r-filler for B (i.e., Ind fills B's r role) and B has a value restriction indicating that all r-fillers are in C; in this case we can deduce that Ind $\rightarrow$ C. For example, if SS3 $\rightarrow$ (**and** (**all** speaker SPEAKER) (**fills** speaker S1)), then S1 $\rightarrow$ SPEAKER because of a propagation through the speaker role on individual SS3.

Trigger Rules: Many DLs have some form of forward chaining rule. CLASSIC allows rules to be attached to concepts that state that anything that is an instance of the concept must be an instance of the consequent of the rule.

Trigger rules are typically used when some property is not needed for recognition (or in CLASSIC, when necessary conditions involve recursion). For example, our previous definition of a STEREO-SYS required satisfying instances to have 2 fillers for their speaker role, *and* for these fillers all to be instances of the concept SPEAKER. A knowledge engineer may only want to require an individual to be a SYSTEM and something that has **at-least** 2 speakers before it is recognized to be a STEREO-S.

STEREO-S $\equiv$ (**and** SYSTEM
(**at-least** 2 speaker)).

A value restriction could then appear on fillers of the speaker role *as a result of* recognizing that an individual is a STEREO-S. To accomplish this, a rule can be attached to STEREO-S with the consequent (**all** speaker SPEAKER). Thus, once an object is recognized to be an instance of STEREO-S, it will also obtain the restriction that its speakers must be instances of the concept SPEAKER. Similarly, PERSON could have a rule attached to it with the consequent (**all** parent PERSON), thus having the effect of making all individual PERSONs have parents who are instances of PERSON.

It should be noted that rules are considered as "epistemic statements" about individuals, and are never run on concepts, and hence are not considered during subsumption reasoning. Thus, if GOOD-STEREO-SYSTEM is a subclass of STEREO-S, the DL will not determine that all of the fillers of the speaker role on GOOD-STEREO-SYSTEM are instances of SPEAKER. Any individual instance of GOOD-STEREO-SYSTEM, however, will have this restriction on it since the rule will fire on every instance of STEREO-S (though not on its subclasses).

Tests: As mentioned earlier, tests provide a way of extending the language of concept constructors, albeit only for the purposes of individual reasoning. Tests are run on individuals to check to see if the individual passes the test and can thus be classified as an instance of the test concept, i.e., tests define concepts procedurally. For example, if an individual, Ind, is being checked to see if it is an instance of (**test** Debs-Favorite), then the Debs-Favorite function must be run on Ind. If it returns true, then Ind is definitely an instance, if it returns false, it is not an instance, and if it returns unknown, the DL will not determine that Ind is an instance now but it will re-run the test if something changes on Ind to see if the test will return true after the change.

Inconsistency Checking: Individuals can be locally inconsistent, i.e., they can be asserted or derived to be instances of unsatisfiable descriptions. For example, an individual that is asserted to be an instance of (**and** (**at-most** 0 r) (**fills** r A)) is inconsistent. In addition, an individual can look locally consistent but it can be inconsistent with the rest of the knowledge base. For example, consider a knowledge base, K, that contains

A $\rightarrow$ (**at-most** 0 r).

A new individual may be asserted with the following statement:

B $\rightarrow$ (**and** (**fills** s A) (**all** s (**at-least** 1 r))).

Just inspecting the description for B would not yield problems but when a DL merges that information with the rest of the knowledge base, it would discover that the knowledge base is inconsistent since now A must both have (**at-most** 0 r) and (**at-least** 1 r).

2.6 Knowledge Base Operations

We already stated that the terminological knowledge base contains concept declarations (either definitions or assertions of necessary conditions for primitives). Every concept appears once on the left hand side of $\sqsubseteq$ or $\equiv$. CLASSIC does not provide operations for changing the declaration of concept names.

Rules can also be introduced into the knowledge base. (Syntactically, CLASSIC considers them as additions to concept declarations.) CLASSIC includes a knowledge base operation that allows the deletion of a rule. The deleted rule is removed from the knowledge base and any individual that had that rule fire on it is re-normalized and in effect has the consequent of that rule removed from it.[9]

In addition to introducing individuals, knowledge base updates may modify their properties. In particular, information may be added to or deleted from individuals, and roles may be closed or un-closed on an individual in an incremental manner. Thus, in CLASSIC one can assert first that `Ind` $\rightarrow$ `SYSTEM` and then add that `Ind` is an instance of (**fills** `speaker S1`) and also an instance of (**at-most** 2 `speaker`). Later, additional fillers could be added to the `speaker` role. If, however, one tried to add a third filler to the `speaker` role, a contradiction would be detected. CLASSIC identifies the inconsistency, disallows the last update that caused the inconsistency, and reverts to the last consistent state of the knowledge base.

Most DLs also allow some form of deletion. CLASSIC allows removal of descriptions that have been told to the system about an individual. For example, if someone has created:

`Ind` $\rightarrow$ (**and** (**fills** `r A`) (**at-most** `3 r`)),

CLASSIC would allow the deletion of (**fills** `r A`) or (**at-most** `3 r`) from `Ind` but it would not allow the deletion of information that has been derived, such as (**at-least** `1 r`). Removing information from an individual may cause it to be reclassified, and if other individuals refer to it, it can cause other individuals to change as well. For example,

[9] This rule retraction is more extensive than the retraction in many rule-based systems which will not "undo" the effects of rule firings.

consider the following assertion:

SS2 $\rightarrow$ (**and** (**fills** speaker S1) (**all** speaker SPEAKER)).

From this, a DL will derive that:

S1 $\rightarrow$ SPEAKER.

Later, if the value restriction on speaker is removed on SS2, then S1 should no longer be known to be an instance of SPEAKER.

One last kind of deletion is "unclosing" a role on an individual, which has the effect of removing the explicit closed assertion introduced earlier and retracting any (**at-most** n r) restrictions that were derived as a result of the previous role closure.

2.7 Techniques for Computing Concept Subsumption

There are two main strategies for calculating subsumption in DLs.

The first approach, illustrated in Section 2.7.1, essentially starts by *normalizing* each description into a canonical form where all implicit information is made explicit (i.e., new components are deduced as necessary) but redundant information is eliminated. The resulting normal form contains the same categories of information for every concept concerning the various possible constructors used in the language. The second phase of subsumption determination is then essentially a quick *structural comparison*, where for every constructor in the candidate subsumer normal form we check the corresponding component of the candidate subsumee, and compare the corresponding entries, to make sure that the subsumee implies the subsumer. Further details of this structural subsumption technique are provided in Section 2.7.1, and this material will be useful mostly for reading Sections 3.5, 3.6, and Chapter 5, particularly Section 5.1.

A second approach to computing subsumption, based on tableaux-like theorem-proving techniques, and used in certain European DLs, will be presented in Section 2.7.2; this material will be useful, but not required, for reading Chapter 5.

2.7.1 Structural Subsumption

Structural subsumption is implemented in a two phase normalize/compare process. In order to determine if `A` subsumes `B`, `B` is first normalized. During normalization, a structured form is completed using information that has been told to the system about `B`. This told information is used along with rules of inference to try to derive additional information, and to eliminate redundant parts. After `B` is normalized, the syntactic form of `B` is compared to the syntactic form of `A` to check subsumption.

For example, consider declarations

`A` ≡ (**at-least** 2 `speaker`)

and

`B` ≡ (**fills** `speaker S1 S2 S3 S4`).

Normalization will gather all information told to the system about `B`. In this case, it is only information about the `speaker` role. The DL will need to fill in the following form:

---

concept: `B`

---

superconcepts:

one-of :

same-as :

test:

restrictions

on roles: `speaker`

at-least:

at-most:

value

restrictions:

fillers:

First, it will fill in any explicitly told information - in this case, it will put the fillers in for the `speaker` role. Then it will iteratively expand any defined concepts with their defining expression. The concept identifiers in the resulting expression will all be primitive, and marked with **prim**. The process terminates since CLASSIC knowledge bases are acyclic. All the primitive concepts are augmented by their necessary conditions. (In this example, there are no primitive concepts.)

Following this, normalization will try to deduce more information. For example:

1. Every concept has a superconcept of `THING`.
2. If a concept has role restrictions, then it is in the CLASSIC realm and it has the primitive (**prim** `CLASSIC-THING`)
3. Everything has a top level **one-of** restriction that is equal to or more specific than all the elements in the KB, which we will call `TOP-ONEOF`.
4. Every description begins with no required **same-as** restrictions.
5. Every description begins with an empty list of **test** restrictions.
6. Every object has at least 0 fillers for every role.
7. If there are n known fillers for a role, r, then the **at-least** restriction on r must be at least n. (In this example, there are 4 known fillers for the `speaker` role, so there is an **at-least** 4 restriction on `speaker`. Since an **at-least** restriction of 4 is more specific than an **at-least** restriction of 0, 4 will replace 0 in the normalized structure.)
8. Every role has some maximum number of fillers - we will call the greatest such maximum `MAXINT`.
9. Every role has a value restriction that is equal to or more specific than `THING`.

These rules allow the system to fill out the form below:

---

concept: `B`

---

primitive

superconcepts: `CLASSIC-THING`, `THING`

one-of: `TOP-ONEOF`

same-as: $\emptyset$

test: $\emptyset$

restrictions

on roles: `speaker`

at-least: 4

at-most: `MAXINT`

value

restrictions: `THING`

fillers: `S1`, `S2`, `S3`, `S4`

After normalization is completed, one can compare this structure to the structure of the candidate subsumer concept.[10] The normalized form of `A` contains the following information:

---

concept: `A`

---

primitive

superconcepts: `CLASSIC-THING`, `THING`

one-of: `TOP-ONEOF`

[10]It is possible, and in fact, more efficient to use the un-normalized form of `A` since everything in the normalized form is derivable from the un-normalized form. We only use two normalized forms for pedagogical simplicity.

same-as:	∅		
test:	∅		
restrictions			
on roles:	`speaker`		
		at-least:	2
		at-most:	`MAXINT`
		value	
		restrictions:	`THING`
		fillers:	∅

The goal of the comparison is to make sure that everything that is in `A` is implied by `B`, and thus any individual that is an instance of `B`, must be an instance of `A`. For example, in order for `A` to subsume `B`, (**at-least** 2 `speaker`) must be implied by `B`. Since `B` has an (**at-least** 4 `speaker`) restriction, this holds. Once a DL has a normal form for the objects, it is fairly simple to define a syntactic structural comparison that will guarantee subsumption. `A` will subsume `B` in CLASSIC by making the structural comparisons in Table 2.2.[11]

In our example, `A` subsumes `B` because of *all* of the following specific relationships:

1. {`CLASSIC-THING, THING` } is a subset of itself. (This is needed for primitive subconcept subsumption.)

2. `TOP-ONEOF` is a superset of itself. (This is needed for **one-of** subsumption.)

3. ∅ is a subset of itself. (This is needed for **same-as** and **test** subsumption.)

4. $2 \leq 4$. (This is needed for **at-least** subsumption on the `speaker` role.)

5. `MAXINT` $\geq$ `MAXINT`. (This is needed for **at-most** subsumption on the `speaker` role.)

---

[11]The **same-as** constructor is not in this list since it requires some special processing and a graph-based implementation. The graph structures and the normalization process dealing with **same-as** are detailed in sections 2.2 and 2.3 of [20].

1. `A`'s primitive superconcepts are a subset of `B`'s primitive superconcepts. (For example, if `A` has to be a `CLASSIC-THING` and a `THING` and `B` has to be a `STEREO-SYS`, `CLASSIC-THING` and a `THING`.)

2. `A`'s top level **one-of** restriction is a superset of `B`'s top level **one-of** restriction. (For example, if instances of `A` must be `Louise`, `Dick`, or `Helen`, and instances of `B` must be `Louise` or `Dick`.)

3. If `A` is a host concept, `A`'s minimum restriction is less than or equal to `B`'s minimum restriction and `A`'s maximum restriction is greater than or equal to `B`'s maximum restriction. (For example, if instances of `A` must be between 0 and 100 and instances of `B` must be between 0 and 20.)

4. `A`'s **test** restrictions are a subset of `B`'s **test** restrictions. (For example, if instances of `A` must pass the Deb-Favorites test and instances of `B` must pass the Deb-Favorites and Alex-Favorites test.)

5. For every role on `A`:
   - `A`'s **at-least** restriction is $\leq$ `B`'s **at-least** restriction. (For example, if `A`s must have (**at-least** 2 speaker) and `B`s must have (**at-least** 4 `speaker`).)
   - `A`'s **at-most** restriction is $\geq$ `B`'s **at-most** restriction. (For example, if `A`s must have (**at-most** 1 `subwoofer`) and `B`s must have (**at-most** 0 `subwoofer`).)
   - `A`'s fillers are a subset of `B`'s fillers. (For example, if `A`s speaker fillers include `S1` and `S2` and `B`s `speaker` fillers include `S1`, `S2`, `S3`, and `S4`.)
   - `A`'s value restriction recursively subsumes `B`'s value restriction using this same structural comparison function. (For example, if `A`'s `speakers` must be instances of `SPEAKER` and `B`'s `speakers` must be instances of `HIGH-QUAL-SPEAKER`.)

Table 2.2: CLASSIC Structural Subsumption Requirements

6. `THING` $\Longrightarrow$ `THING`. (This is needed for value restriction subsumption on the `speaker` role.)

7. $\emptyset$ is a subset of { `S1`, `S2`, `S3`, `S4` }. (This is needed for filler subsumption on the `speaker` role).

This was a particularly simple example. The rules for structural comparison in Table 2.2 handle all of CLASSIC. The complete set of subsumption inferences that we report in our implemented version for CLASSIC 2.0 (and beyond) can be found in Appendix B. (The complete set of normalization inferences is also found in Appendix A.) It is worth noting that normalization deductions about one part of the structure may force a change to another part of the structure. For example, if

`B` $\equiv$ (**all** r (**and** (**at-least** 2 s) (**at-most** 1 s))),

`B`'s told information would look as follows:

---

concept: `B`

---

restrictions
on roles: r

restrictions
on roles: s

at-least: 2
at-most: 1

Since it is impossible to have 2 or more and 1 or less of anything, a new value restriction of `NOTHING` is derived for `r`. This information is used to deduce that `r` can have no fillers and thus it will have an **at-most** restriction of 0 in its final structure. This information is shown below:

```
concept:        B
------------------------------------------------------------
primitive
superconcepts:  CLASSIC-THING, THING
one-of:         TOP-ONEOF
same-as:        ∅
test:           ∅
restrictions
 on roles:      r
                  at-least:       0
                  at-most:        0
                  value
                   restrictions:  NOTHING
                  restrictions
                   on roles:      s
                                    at-least:      2
                                    at-most:       1
                                    value
                                     restriction:  THING
                                    fillers:       ∅
                  fillers:        ∅
------------------------------------------------------------
```

It is also worth noting that these structures could be viewed as infinite since every role has a value restriction of THING on it. Thus, every object has an (**all** r THING) restriction on it for every role in the knowledge base. In order to support termination, implementation and presentation mechanisms will need to generate a value restriction on a role only if there is information about this role that is more specific than THING. Declarations in CLASSIC are not cyclic so structures will not be cyclic.

For normalize/compare approaches, subsumption will be sound if the structural comparison algorithm is correct and if the normalization algorithm only adds correct information to the structures. Subsumption will be complete if the comparison algorithm checks all parts of the structure and the normalization algorithm performs *all* the inferences that it should. It is not hard to make these algorithms sound but it typically is computationally expensive to make them complete. In fact, most implemented normalize/compare DLs are incomplete.

2.7.2 Tableau Method

Another method for determining subsumption has been proposed and developed in [107, 65, 48, 31], among others. This method is driven by the desire to find complete algorithms for subsumption. The basic inference in this case is determining if a concept is inconsistent or not, and relies on the observation that `C` is subsumed by `D` if and only if (**and** `C` (**not** `D`)) is *not* satisfiable. Thus, there exists a model for `Descr` $\equiv$ (**and** `C` (**not** `D`)), if and only if `C` is *not* subsumed by `D`. The algorithms try to generate such a finite model for `Descr` using an approach similar to first-order tableaux calculus with a guaranteed termination.

If the algorithm succeeds, then the subsumption relationship does not hold and the algorithm has a counter example to prove it. Since the algorithms are complete, if the algorithm fails to find a model, then the subsumption relationship holds. In the case where the subsumption relationship holds, the algorithm looks for a "clash" among the constraints, which would preclude a model from existing. We will illustrate such an algorithm by two examples — one where the subsumption relationship holds and the algorithm discovers a constraint clash and another where the relationship does not hold and the algorithm generates a counter example. For a complete theoretical introduction, see [107, 31].

Example 1: Attempt to prove the true statement (**at-least** `4 r`) $\Longrightarrow$ (**at-least** `2 r`).

To do this, check whether

(1) `Descr` = (**and** (**at-least** `4 r`) (**not** (**at-least** `2 r`)))

is unsatisfiable. First, push negations into the expression as far as possible using an extended form of de Morgan's rules, with the usual rules for quantifiers and the extensions:

(**not** (**at-least** `n r`)) = (**at-most** `(n-1) r`) and

(**not** (**at-most** `n r`)) = (**at-least** `(n+1) r`).

As a result, we obtain:

(2) $\texttt{Descr}_1$ = (**and** (**at-least** `4 r`) (**at-most** `1 r`))

Next, the algorithm tries to generate a finite interpretation I such that ${\texttt{Descr}_1}^I \neq \emptyset$. Thus, there must exist an individual in Δ^I that is an element of ${\texttt{Descr}_1}^I$. The system will generate an individual b and state that $b \in {\texttt{Descr}_1}^I$. Since $\texttt{Descr}_1$ is a conjunction, that means that the individual, b, is an instance of each of the conjuncts:

(3) $b \in$ (**at-least** `4 r`)^I and

(4) $b \in$ (**at-most** `1 r`)^I.

From (3) we can infer that there must be four individuals e_1, e_2, e_3, and e_4 that all fill the `r` role on b. One of the known clashes that this algorithm looks for is a statement of the form $b \in$ (**at-most** `n r`)^I along with the knowledge that b has more than `n` `r`-fillers. The algorithm identifies the clash, thus proving that no model can exist, and concludes that

(**at-least** `4 r`) $\Longrightarrow$ (**at-least** `2 r`).

Example 2: Attempt to prove the false statement (**at-least** `2 r`) $\Longrightarrow$ (**at-least** `4 r`).

As before, try to determine if

(1) `Descr` = (**and** (**at-least** `2 r`) (**not** (**at-least** `4 r`)))

is unsatisfiable. Using the extended de Morgan's laws, we get:

(2) $\texttt{Descr}_1$ = (**and** (**at-least** `2 r`) (**at-most** `3 r`)))

When the algorithm tries to generate the interpretation, it will generate b, which must be an instance of ${\texttt{Descr}_1}^I$. Thus,

(3) $b \in$ (**at-least** `2 r`)^I and

(4) $b \in (\textbf{at-most } 3 \;\; \texttt{r})^I$.

From (3), we can infer that there must be two individuals e_1 and e_2 that both fill the r role on b. At this point, no further rewrite rules apply, and we have not found a clash. We therefore have a model – an interpretation that satisfies the constraints: $\Delta^I = \{b, e_1, e_2\}$, and $r^I = \{ (b, e_1), (b, e_2) \}$.

Since the algorithm has generated a model, it is possible to be an instance of (**at-least** 2 r) and not an instance of (**at-least** 4 r) and thus, the subsumption relationship does not hold.

2.8 Techniques for Computing Individual Membership in Concepts

This material will be important for Chapter 4 particularly in section 4.2 where individual recognition is discussed. It will also be the useful for determining *non*-membership in Chapter 5.

One way of recognizing that an individual is an instance of (**at-least** n r) is to identify n (distinct) fillers of the r role. The rules of so-called "structural recognition" are based squarely on the denotational semantics of the concept constructors, which determine the individuals that belong to the denotation $D^{\mathcal{I}}$ of a description D. They work when the part of the knowledge base we need to consider for membership in D can be viewed as a single, "closed world" interpretation I. The rules appear in Figure 2.1.

Structural recognition will handle the recognition task when information about an individual is complete in the sense that we know all the possible fillers for the roles of interest. There will be times however when individuals are not complete yet we can still recognize concept membership. For example, suppose an individual is asserted as follows:

Ind $\rightarrow$ (**at-least** 3 r)

and the system wants to decide if Ind $\rightarrow$ (**at-least** 2 r). The DL does not have two known fillers for the r role that would allow it to structurally recognize that Ind $\rightarrow$ (**at-least** 2 r). Thus, structural recognition needs to be augmented in order to find instance relationships between an individual with incomplete information and a description. In

`Ind` $\rightarrow$ `B` if:

1. `Ind`'s primitive superconcepts are a superset of `B`'s primitive superconcepts. (For example, if: `Ind` $\rightarrow$ (**prim** `STEREO-SYS CLASSIC-THING THING`) and `B`'s **prim**s are `CLASSIC-THING` and `THING`).

2. `Ind` is an element of `B`'s top level **one-of** restriction. (For example, if `B` $\equiv$ (**one-of** `Sony Toshiba`) and `Ind` is `Toshiba`.)

3. If `Ind` is a host individual, and `Ind` is $\geq$ `B`'s minimum restriction and $\leq$ `B`'s maximum restriction. (For example, if `Ind` is 400 and `B` $\equiv$ (**and** (**min** 200) (**max** 500)).)

4. `Ind` satisfies `B`'s **test** restrictions in the sense that the test functions return true when run on the individual `Ind`.

5. If `Ind` satisfies all the **same-as** restrictions on `B`. (For example, if `B`'s **same-as** restriction was:
   (**same-as** (`tuner manufacturer`) (`preamplifier manufacturer`)) and
   `Ind` $\rightarrow$ (**and** (**fills** `tuner Denon680NAB`) (**fills** `preamplifier Denon5000`)) and
   Denon680NAB $\rightarrow$ (**fills** `manufacturer Denon`) and
   Denon5000 $\rightarrow$ (**fills** `manufacturer Denon`).)

6. For every role `r` on `B`:
   - There are n distinct fillers for role `r` and n $\geq$ the **at-least** restriction on `B`. (For example, if `SS3` has `S1` and `S2` as `speaker` fillers, then `SS3` will be recognized to be an instance of (**at-least** n `speaker`) for n $\leq$ 2.)
   - `Ind`'s `r` role has n fillers, it is closed and n $\leq$ m where m is the **at-most** restriction on `B`. (For example, `Ind` has 4 `speakers` and the `speaker` role is closed, then it will be recognized to be an instance of (**at-most** m `speaker`) for m $\geq$ 4.)
   - `Ind`'s `r`-fillers are a superset of `B`'s `r`-fillers. (For example, if `SS3`'s `speaker` fillers include `S1`, `S2`, `S3`, and `S4`, then `SS3` will be recognized to be an instance of (**fills** `r S1 S2`).)
   - `Ind`'s `r` role is closed, and every `r`-filler is an instance of `B`'s value restriction on `r`. (For example, if `SS4` has exactly 2 `speakers` and those `speakers` are both manufactured by `Boss`, then `Ind` is an instance of (**all** `speaker` (**fills** `manufacturer Boss`)).)

Figure 2.1: CLASSIC Structural Recognition Requirements

fact, structural recognition is augmented by concept reasoning. If the description of `Ind` is subsumed by C, then `Ind` is an instance of C. In this case, since (**at-least** 3 r) $\Longrightarrow$ (**at-least** 2 r), then `Ind` $\rightarrow$ (**at-least** 2 r).

Structural recognition is used first since it is sound, complete and efficient in the presence of full information (as is the case with database reasoning). If structural recognition does not find an instance relationship, then the system reverts to concept reasoning (which may be incomplete). In order for structural recognition to be most likely to work, the individuals should be normalized so that they are as complete as possible. For example, if `Ind1` is defined as follows in CLASSIC:

`Ind1` $\rightarrow$ (**and** (**all** r (**one-of** `A B`)) (**at-least** `2 r`)),

the system may need to decide if `Ind1` $\rightarrow$ (**fills** `r A`). The only structural rule it has to decide this involves finding that `Ind` has `A` as a filler of `r`. The told information does not explicitly say that `Ind1` $\rightarrow$ (**fills** `r A`), however this can be deduced. Normalization can determine that `Ind` must have 2 fillers and those fillers must be chosen from the set {`A`,`B`}, thus, `r` must be filled with `A` and `B`. All of the normalization inferences for concept descriptions will need to be used in individual normalization along with some special inferences that are unique to individuals. A complete set of the normalization inference rules appears in Appendix A.

Chapter 3

Explaining Subsumption

Subsumption (section 2.3) is the core inference in description logic-based knowledge representation and reasoning systems, thus explaining subsumption forms the core of the explanation module. An explanation requires a justification of the system's beliefs. In rule-based and PROLOG systems, an explanation facility is often based on a trace of rule firings since rule invocation mimics modus ponens, which is the fundamental logical inference rule of the system.

In most efficiently implemented DLKRSs, an execution trace is inappropriate since the procedural implementation is too far removed from the underlying logic. For example, the system may determine concept equivalence (i.e., two concepts subsume each other) because a hash function assigns them to the same "bucket", yet it would be inappropriate to refer to a hash function in an explanation. Another example that we referred to in [90] shows that co-reference constraints are checked in CLASSIC by traversing a graph structure, yet it would be undesirable to include such a graph in most explanations. This procedural implementation style is the natural side effect of the DL's implementor's need to be efficient. Sacrificing efficiency for the sake of explanation is not appropriate, so our goal is not to require a rewrite of the core inference system, but instead to provide a structure that can interface to the existing implementation. These issues are similar to the problems faced by deductive databases such as LDL [110], where efficient compilation hides the original program's rule structure. As in LDL, explanation benefits from a declarative view of the reasoning, and we turn to the proof theory of description logics to obtain this.

Given that we are forced to augment the system to provide a declarative presentation of inference, we have two options. We could take an approach similar to Wick and

Thompson [126] (described in Section 7.1) and provide a reconstructive explanation from a system specially designed to provide explanations. Alternatively, we could attempt to make minimal additions to the core inference system and add an explanation module that can use the output of the minimally altered deduction engine to provide explanations. Many of the basic ideas provided in the thesis can be incorporated into either approach although we believe that explanation for an evolving system needs to be tightly coupled with the core inference engine, otherwise the two systems will get out of sync. Thus, we have investigated the issues concerning the integration of an explanation system with an implemented description logic and the following presentation describes an approach that can be used to augment the core inference engine. This explanation module is implemented in CLASSIC. The integration is discussed in Section 3.6.4.

This chapter lays the foundation for explanation in description logics implemented using normalization methods. It considers the problem of explaining subsumption from the perspective of knowledge engineers. It first observes that the deductions performed in description logics can be captured in the form of proof rules. The notion of explanation is then specified using a proof-theoretic framework. Proof rules are presented for an example description logic and are used as the basis of explanations. The problem of overly long explanations is addressed by decomposing them into smaller, independent steps, using the notions of "atomic description" and "atomic justification". Explanation fragments lead to the notion of a single step explanation and automatic follow-up, where follow-ups are generated from the form of inference rules. Implementation aspects are explored by considering the design space and some desiderata for explanation modules.

3.1 Proof Theory

Intuitively, an explanation is a convincing argument. Arguments may be presented formally in logic using the notion of a proof. A proof is a sequence of sentences, where each sentence is either a given statement, an axiom, or something that is derived from the previous sentences using a rule of inference for the logic. For example, modus ponens is a rule of inference, which can be expressed as

$$\frac{\vdash \sigma \supset \rho \qquad \vdash \sigma}{\vdash \rho}$$

indicating that if the two *antecedent assertions*, $\sigma \supset \rho$ and σ, on top can be deduced, then the *consequent assertion* ρ is deducible in the next step of the proof.

Modus ponens can be used to give an explanation of deductions. For example, consider the knowledge base KB consisting of the statements:

$$p$$
$$p \supset q$$
$$q \supset r.$$

Using a proof format where each line of a proof is numbered and lines are "justified" by either "told" or "modus ponens" applied to previous lines in the proof, we can prove r as follows:

1	KB $\vdash$ p	told statement
2	KB $\vdash$ p $\supset$ q	told statement
3	KB $\vdash$ q	modus ponens 1,2
4	KB $\vdash$ q $\supset$ r	told statement
5	KB $\vdash$ r	modus ponens 3,4

Note the use of the "turnstile" notation

$\{p, p \supset q, q \supset r\} \vdash r$

to indicate that r is deducible from the told statements in KB.

Observe that the form of the inference rules can make a difference in the form of proof. For example, consider a slightly different set of statements KB':

p

p $\supset$ q

(p & q) $\supset$ r.

Although KB' is logically equivalent to KB, and it is still the case that r can be proven from the assumption set KB', the proof will differ. If there is another inference rule for "introducing conjunction", of the form

$$\text{and introduction} \quad \frac{\vdash \sigma \qquad \vdash \rho}{\vdash (\sigma \ \& \ \rho)}$$

we can provide a simple proof:

1	KB' $\vdash$ p	told statement
2	KB' $\vdash$ p $\supset$ q	told statement
3	KB' $\vdash$ q	modus ponens 1,2
4	KB' $\vdash$ (p & q)	and introduction 1,3
5	KB' $\vdash$ (p & q) $\supset$ r	told statement
6	KB' $\vdash$ r	modus ponens 4,5

A different axiomatization of the logic might eschew AND-introduction, and instead have rules corresponding to de Morgan's laws, such as,

$$\frac{\vdash \sim(p \ \& \ q)}{\vdash \sim p \vee \sim q}$$

as well as inference rules for arguing by contradiction, and discharging disjunction. In this case, one would have to first assume $\sim$(p & q), and then infer that either $\sim$p or $\sim$q must hold; $\sim$p can not hold since that contradicts the told information; likewise, $\sim$q can not hold since we can deduce q using modus ponens; thus, our assumption is wrong and (p & q) must be true.

Finally, if we used a more standard Frege/Hilbert-style axiomatization (e.g., [102] Section 2.5.4), with the only additional inference rules being axioms (rules with no antecedent) such as $\vdash \sigma \supset (\rho \supset \sigma)$, then the proof would be much, much longer, involving a whole series of lemmas.

Our point is that depending upon the different rules of inference and the form of the statements and axioms, the proof can vary in length and, potentially, understandability.

3.2 Proof Theory for Subsumption in Description Logics

In standard logic, one is interested in the "truth" of formulas, so formulas are the appropriate units for proofs. In description logics, one is interested in relationships between descriptions and/or individuals. We will refer to these relationships as "judgements". Judgements can be "proven" using a set of inference rules and some given statements. Kahn first used this approach in programming languages [71] and Borgida used the approach in description logics [12]. Subsumption judgements like C $\Longrightarrow$ D can be proven using a set of inferences rules and some given statements – the declarations of concept identifiers in a knowledge base KB. For example, we can deduce that (**all** `r C`) $\Longrightarrow$ (**all** `r D`) when `C` is more specific than `D`, i.e., when `C` $\Longrightarrow$ `D`. We can write this as the following natural semantics style proof rule:

$$\textsf{All}\quad \frac{KB \vdash \beta \Longrightarrow \delta}{KB \vdash (\textbf{all}\ \pi\ \ \beta) \Longrightarrow (\textbf{all}\ \pi\ \ \delta)}$$

which can be paraphrased "If it is deducible, possibly using declarations from the knowledge base KB, that some description β is subsumed by another description δ, then it is deducible from the knowledge base KB that the description (**all** π β) is subsumed by the description (**all** π δ)".

Since we always consider only a single knowledge base, for convenience we will henceforth drop the explicit KB from the left-hand side of $\vdash$, while in proofs we omit the turnstile as well.

The set of inference rules for deducing subsumption relationships provides an alternate characterization of the semantics of a DL. For pedagogical reasoning, we will introduce next the inference rules needed for a miniature DL.[1] We will call the small language Mini-CLASSIC. It includes the top concept `THING`, the bottom concept `NOTHING`, and the constructors **and**, **all**, **at-least**, **at-most**, and **prim**. This grammar is shown in Figure 3.1.

The All inference rule that was mentioned previously was particular to the **all** constructor. There will be one or more inference rules associated with each constructor that address how expressions built with that constructor relate to other expressions built with the same constructor. For example, how does (**at-least** `4 speaker`) relate to (**at-least** `2 speaker`) or (**at-least** `2 turntable`)? An **at-least** expression (**at-least**

[1]The entire set of inference rules that we report for CLASSIC 2.3 is included in appendices A to C.

<Descr> := THING | NOTHING |
(**and** <Descr>$^+$) |
(**all** <role-name> <Descr>) |
(**at-least** <positive-integer> <role-name>) |
(**at-most** <non-negative-integer> <role-name>) |
(**prim** <concept-name>$^+$)

Figure 3.1: Mini-CLASSIC Grammar

n r) is more specific than another **at-least** expression (**at-least** m p) when n $\geq$ m and p=r. This can be written:

$$\text{AtLst}\quad \frac{n > m}{\vdash (\textbf{at-least}\ n\ \pi) \Longrightarrow (\textbf{at-least}\ m\ \pi)}$$

Similarly, another rule compares two **at-most** restrictions, with (**at-most** n r) more specific than (**at-most** m r) when n $\leq$ m (the opposite relationship to that for **at-least**). This can be written as the inference rule

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

A description containing one set[2] of primitives will be more specific than an expression containing another set of primitives when the first set is a superset of the second set. For example, something that is both a STEREO-SYSTEM and GOOD must be GOOD, i.e., (**prim** STEREO-SYSTEM GOOD) $\Longrightarrow$ (**prim** GOOD).

$$\text{Prim}\quad \frac{\tilde{\alpha} \subset \tilde{\theta}}{\vdash (\textbf{prim}\ \tilde{\theta}) \Longrightarrow (\textbf{prim}\ \tilde{\alpha})}$$

The above inference rules essentially perform the structural subsumption comparison mentioned in Chapter 2.7. Corresponding to normalization, we have inference rules that translate between syntactically distinct expressions that have the same semantics. For example, it is always the case that there is an **at-least** 0 restriction on every role, thus (**at-least** 0 r) is identical to the top concept. Additional equivalence rules usually arise from conjoining two descriptions built using the same constructor (e.g., All-and and PrimAnd from Figure 3.2).

[2]The tilde superscript above a variable (as in $\tilde{\theta}$) indicates that the variable represents a set.

Usually, there are also rules describing ways in which one can arrive at incoherent concepts: those expressions are equivalent to the bottom concept. In this simple language, there is only one way to generate an incoherency: the rule BdsCnflct says, for example, that (**and** (**at-least** 4 `speaker`) (**at-most** 2 `speaker`)) is incoherent. Of course, more generally it says that any description with an **at-least** restriction on a role that is greater than the **at-most** restriction on the same role is incoherent.

In addition, all DLs have some more fundamental inference rules. For example, consider the transitivity rule that says if a description, say `HIGH-QUAL-STEREO-SYSTEM`, is subsumed by a description, say `STEREO-SYSTEM`, and in turn `STEREO-SYSTEM` is subsumed by (**at-least** 2 `speaker`), then `HIGH-QUAL-STEREO-SYSTEM` is subsumed by (**at-least** 2 `speaker`). This deduction is captured by the rule

$$\frac{\vdash \beta \Longrightarrow \delta\,, \quad \vdash \delta \Longrightarrow \lambda}{\vdash \beta \Longrightarrow \lambda}$$

These rules are included in the set of rules for mini-CLASSIC which are summarized in Figure 3.2. Rules are named for later reference in proofs. The figure contains additional basic rules like reflexivity (every description is subsumed by itself), the top concept `THING` subsumes everything, and the bottom concept `NOTHING` is subsumed by everything.

Another set of basic rules involve the **and** constructor. If a description is subsumed by two different descriptions, then it should be subsumed by the conjunction of those two descriptions. This rule is named AndR. Similarly, if a concept β is already more specific than another concept, then adding a restriction to β will not change the subsumption relationship.

Concerning the knowledge base, we have two options. One is to expand class identifier declarations, as described in Chapter 2, before performing any subsumption reasoning, and consider this a preprocessing step that is outside the proof theory (and hence outside of explanation). Alternately, we can add a couple of inference rules for reiterating told facts from the knowledge base concerning the necessary and/or sufficient conditions of declared identifiers:

$$\text{ToldDef} \qquad \{ID \equiv C, ...\} \vdash ID \equiv C$$

$$\text{ToldPrim} \quad \{ID \sqsubset C, ...\} \vdash (\textbf{prim}\ ID) \Longrightarrow C$$

together with a rule Eq allowing "equals to be substituted for equals".

When constructing the set of inference rules for a particular DL, it is important to consider the soundness and completeness of the rules. First, of course the rules should be sound, i.e., applying a rule to a knowledge base should produce only valid judgements with respect to the semantics specified (see Table 2.1). Second, and usually more difficult, is determining that the rules can be used to deduce everything that is logically implied by the semantics of the constructors. A particular implementation may however not detect all subsumptions, in which case an incomplete set of inference rules may characterize the entire implementation of a system.

3.3 Subsumption Explanations as Proofs

Consider a concept A, defined as

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

and let us consider the explanation of the subsumption

$$\texttt{A} \Longrightarrow (\textbf{and}\ (\textbf{at-least}\ 2\ \text{grape})\ (\textbf{prim}\ \text{WINE})).$$

Since subsumption can be captured by inference rules, we start by making the explanation of a subsumption relationship be its proof. We justify each line by the name of the description logic rule used to deduce it, and the line numbers on which the antecedents, if any, of the rule appear in Figure 3.3. If such a proof is to be treated as an explanation, we must resolve several problems: First, the proof is much longer than one might have expected for such a simple case. It includes some lines that are obvious to most readers, e.g., applications of AndEq (creating an equivalent expression by adding an enclosing **and**) and Eq (creating an equivalent expression by substituting equivalent subexpressions). These should be avoided if we are to explain more complex DLs, which have many more rules. In the example above, the final proof should contain

All	$\dfrac{\vdash \beta \Longrightarrow \delta}{\vdash (\textbf{all}\ \pi\ \beta) \Longrightarrow (\textbf{all}\ \pi\ \delta)}$	If value restrictions are subsumed then the ALL restriction is subsumed
AtLst	$\dfrac{n>m}{\vdash (\textbf{at-least}\ n\ \pi) \Longrightarrow (\textbf{at-least}\ m\ \pi)}$	Higher lower bounds are more restrictive, hence subsumed by lower bounds
AtMst	$\dfrac{m>n}{\vdash (\textbf{at-most}\ n\ \pi) \Longrightarrow (\textbf{at-most}\ m\ \pi)}$	Lower higher bounds are more restrictive, hence subsumed by lower bounds
Prim	$\dfrac{\tilde{\alpha} \subset \tilde{\theta}}{\vdash (\textbf{prim}\ \tilde{\theta}) \Longrightarrow (\textbf{prim}\ \tilde{\alpha})}$	If a concept's primitive set contains another set of primitives, then this concept is subsumed by the smaller set of primitives
Trans	$\dfrac{\vdash \beta \Longrightarrow \delta\ ,\ \ \vdash \delta \Longrightarrow \lambda}{\vdash \beta \Longrightarrow \lambda}$	If a concept β is subsumed by δ which is subsumed by λ, then β inherits λ s properties and thus is subsumed by λ
Ref	$\vdash \beta \Longrightarrow \beta$	Concepts subsume themselves
Equal	$\dfrac{\vdash \beta \Longrightarrow \delta \qquad \vdash \delta \Longrightarrow \beta}{\vdash \beta \equiv \delta}$	Two concepts are equivalent if they subsume each other.
Eq	$\dfrac{\vdash \lambda \equiv \eta \qquad \vdash \beta \Longrightarrow \delta}{\vdash \beta' \Longrightarrow \delta'}$	β' and δ' are β and δ with zero or more occurrences of λ replaced by η
Ref	$\vdash \beta \Longrightarrow \beta$	Concepts subsume themselves
Thing	$\vdash \beta \Longrightarrow \texttt{THING}$	THING is the topmost concept and subsumes all others
Nothing	$\vdash \texttt{NOTHING} \Longrightarrow \beta$	The bottom concept NOTHING is subsumed by everything
AndR	$\dfrac{\vdash \beta \Longrightarrow \delta,\ \ \vdash \beta \Longrightarrow (\textbf{and}\ \tilde{\lambda})}{\vdash \beta \Longrightarrow (\textbf{and}\ \delta\ \tilde{\lambda})}$	Concepts subsumed by two concepts are subsumed by the conjunction of those two concepts
AndL	$\dfrac{\vdash \beta \Longrightarrow \delta}{\vdash (\textbf{and}\ \lambda\ \beta) \Longrightarrow \delta}$	A more specific concept created by ANDing the old concept with new information will still be subsumed by the old concept's parents
AndEq	$\vdash \beta \equiv (\textbf{and}\ \ \beta)$	**and** concept equals concept
AtLs0	$\vdash (\textbf{at-least}\ 0\ \pi) \equiv \texttt{THING}$	0 is the atleast restriction on THING
AtMsMax	$\vdash (\textbf{at-most}\ \texttt{MAXINT}\ \pi) \equiv \texttt{THING}$	MAXINT is the atmost restriction on THING
AllThing	$\vdash (\textbf{all}\ \pi\ \texttt{THING}) \equiv \texttt{THING}$	THING is the value restriction on THING
AllNothing	$\vdash (\textbf{all}\ \pi\ \texttt{NOTHING}) \equiv (\textbf{at-most}\ 0\ \pi)$	Inconsistent all restriction means no fillers for that role
All-and	$\vdash (\textbf{and}\ (\textbf{all}\ \pi\ \beta\ )\ (\textbf{all}\ \pi\ \delta\ )\ \ldots\ ) \equiv (\textbf{and}\ (\textbf{all}\ \pi\ (\textbf{and}\ \beta\ \delta))\ \ldots)$	π all restrictions may be combined
PrimAnd	$\vdash (\textbf{and}\ (\textbf{prim}\ \tilde{\lambda})\ \ (\textbf{prim}\ \tilde{\eta})\ \ \ldots\ ) \equiv (\textbf{and}\ (\textbf{prim}\ \tilde{\lambda}\ \ \tilde{\eta})\ \ \ldots)$	Primitives may be combined
BdsCnflct	$\dfrac{n>m}{\vdash (\textbf{and}\ (\textbf{at-most}\ m\ \pi)\ (\textbf{at-least}\ n\ \pi)) \equiv \texttt{NOTHING}}$	Lower higher bounds are more restrictive, hence subsumed by lower bounds

Figure 3.2: Inference Rules for Mini-CLASSIC

1.	**(at-least** 3 grape) $\Longrightarrow$ **(at-least** 2 grape)	AtLst
2.	**(and (at-least** 3 grape) **(prim** GOOD WINE))	
	$\Longrightarrow$ **(at-least** 2 grape)	AndL,1
3.	**(prim** GOOD WINE) $\Longrightarrow$ **(prim** WINE)	Prim
4.	**(and (at-least** 3 grape) **(prim** GOOD WINE))	
	$\Longrightarrow$ **(prim** WINE)	AndL,3
5.	$A \equiv$ **(and (at-least** 3 grape) **(prim** GOOD WINE))	Told
6.	$A \Longrightarrow$ **(prim** *WINE)*	Eq,4,5
7.	**(prim** WINE) $\equiv$ **(and (prim** WINE))	AndEq
8.	$A \Longrightarrow$ **(and (prim** WINE))	Eq,7,6
9.	$A \Longrightarrow$ **(at-least** 2 grape)	Eq,5,2
10.	$A \Longrightarrow$ **(and (at-least** 2 grape) **(prim** WINE))	AndR,9,8

Figure 3.3: Simple Subsumption Proof

just those steps that seem most critical: AtLst and Prim. Thus, we should only generate those proof lines or some filtering mechanism should know how to take a longer proof and present only meaningful fragments.

Second, we note that the proof has more than one focus — one portion concerns reasoning with **at-least** and another concerns reasoning with **prim**. To deal with large, complex proofs, we propose to *decompose* them into parts that can be presented independently. This decomposition will be based on the use of AndR and AndL to break the concept into its component conjuncts, and then proceed with separate, smaller proofs of each part.

Finally, note that the current proof needs to be presented in its entirety since it contains 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. For example, a proof step might have the form

(all wines RED-WINE) $\Longrightarrow$ **(all** wines WINE)

because All(π=wines, β=RED-WINE, δ=WINE)

This form can be generated simply by stating that some subsumption relationship holds "because" a rule was applied with a particular set of bindings for its variables. (Remember the All rule was

$$\frac{\vdash \beta \Longrightarrow \delta}{\vdash (\mathbf{all}\ \pi\ \beta) \Longrightarrow (\mathbf{all}\ \pi\ \delta)}$$

so β, δ, and π are the variables that need bindings reported.)

3.4 Explanations as Proof Fragments

According to Hempel and Oppenheim [64], the "basic pattern of scientific explanation" consists of (i) the *explanandum* — a sentence describing the phenomenon to be explained, and (ii) the *explanans* — the class of those sentences which are adduced to account for the phenomenon. The latter fall into two classes: one containing sentences that state certain antecedent conditions; the other is a set of sentences representing general laws.

Our explanandum is a subsumption 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.

One way to prove that $B \Longrightarrow C$ uses the following two-step approach:

1. Find a description (**and** C_1 ... C_n) that is equivalent to C.

2. Show that $B \Longrightarrow C_i$ for i=1,...,n.

In this approach, the first step of an explanation for $B \Longrightarrow C$ offers a list of descriptions C_i, whose conjunction is equivalent to C, and then explains why B entails each C_i, for every i. In typical knowledge bases, however, descriptions may be very complex, thus there may be very many C_i. While we support allowing the user to ask why B entails *every* C_i, we also provide methods for the user to ask why B entails any particular C_i. In fact one of our contributions is in breaking up explanations into more manageable pieces by allowing the user and the knowledge engineer to limit the amount of information he or she would like to see.

In Normalize-Compare algorithms, introduced in Chapter 2.7.1, B is put into a normal form so that it is itself a conjunction (**and** B_1 ... B_m), and then $B \Longrightarrow C_i$ is proven by finding some component B_j such that $B_j \Longrightarrow C_i$. For example, if

C_i = (**at-least** 2 r) and

B = (**and** (**fills** `r Ind1 Ind2 Ind3`)

(**all** `r C`)),

B would be normalized to generate

B = (**and** (**fills** `r Ind1 Ind2 Ind3`)

(**at-least** `3 r`)

(**all** `r C`)).

Thus, B_j would be (**at-least** `3 r`) which implies (**at-least** `2 r`) and thus, B $\implies$ C_i. In this case, one must also explain how the conjuncts B_i of the normalized form were deduced. So, in the example, we would also need to explain how (**at-least** `3 r`) was deduced.

3.4.1 Atomic Descriptions

Components like C_j above should, ideally, be as simple as possible, so let us call them *atomic*. Consider a description, D_1, such as (**fills** `speaks Italian English`), which can also be written as

D_2 = (**and** (**fills** `speaks Italian`) (**fills** `speaks English`)).

Both of these appear to be normalized; but is one form more atomic than another? Consider a third concept, D_3, defined below, which is equivalent to D_1 and D_2:

$F \equiv$ (**fills** `speaks English`)

$D_3 \equiv$ (**and** F (**fills** `speaks Italian`)).

Now, if an explanation is to be given for the normalization of D_3, there should be two components to the answer: one showing that `English` is a filler of the `speaker` role because of inheritance from F, and another showing that `Italian` is a filler of `speaker` role because of told information. Since each conjunct may be explained differently, we choose to rewrite (implicitly) conjunctive descriptions into explicit conjunctions of atomic descriptions and then to explain each atomic description separately. Therefore, form D_2 is preferred. We will rewrite (or "atomize") a non-atomic description `D` into an equivalent description of the form (**and** E_1 ... E_n) where each of the E_i are atomic. During atomization, we should eliminate redundant conjuncts, i.e., E_i should not subsume E_j for $i \neq j$.

The specification of a *general and absolute* criterion of atomicity that will work for all DLs is unfortunately not likely to succeed. If we ask for components D that do not have the property that $D \equiv$ (**and** E_1 E_2), then no description is atomic, since every description satisfies $D \equiv$ (**and** D E) where E subsumes D. An attempt to follow this line of thought is the notion of "reduced clauses" introduced by Teege [118], which requires that the conjuncts E_i be distinct from D and `THING`. This definition, however, excludes some descriptions that should be atomic, as shown in the following examples:

`NOTHING` $\equiv$ (**and** (**at-least** 2 p) (**at-most** 1 p))

`NOTHING` $\equiv$ (**and** (**prim** CAT) (**prim** DOG))

(**one-of** 4) $\equiv$ (**and** (**one-of** 2 4) (**one-of** 1 4))

Although we would want both `NOTHING` and (**one-of** 4) to be atomic, Teege's definition would *not* include them since they can be rewritten as a conjunction of terms, neither of which are equivalent to `THING` or the expression being rewritten. In a definition that is augmented to handle the CLASSIC language, we distinguish "spurious" conjunctions (which should not be used as signs of non-atomicity) by the fact that they allow an unbounded number of distinct ways of rewriting the description D. For example, in the above two cases involving integers 2, one can use any integers k and l such that $4 < k < l$, not just 2 and 1. So, there is an unlimited number of ways that we can write (**one-of** 4) $\equiv$ (**and** (**one-of** k 4) (**one-of** l 4)) and `NOTHING` $\equiv$ (**and** (**at-least** l r) (**at-most** k r)). Similarly, for the case concerning primitives, one can use any two disjoint primitives. Other research has also utilized a notion similar to our atomic descriptions. Cohen and Hirsh [39], for example, "factor" descriptions into a set of simpler descriptions for use in their learning work. They use an algorithm to define their factors for the CLASSIC DL.

Note that our general approach to explanation does not rely on any particular property of "atomic descriptions", and so the absence of a general definition for this notion is not a problem. Moreover, we can not always avoid atomic justifications involving a form of conjunction: sometimes NOTHING is obtained from combining two conflicting bounds on some role, and this is an implicit conjunction of the bounds. As we shall see later (in Section 3.6.1), candidates for atomic descriptions should be found among the components of normalized concepts used in the implementation. The grammar for atomic descriptions for mini-CLASSIC and full CLASSIC are presented later in Figures 3.7 and 3.8.

3.4.2 Atomic Justifications

Once the subsuming concept is broken up into its atomic descriptions, we use *atomic justifications* to explain each subsumption relationship. An atomic justification has the form

A $\Longrightarrow$ B **because** *ruleId(<argument list>).*

where B is an atomic description, *ruleId* is the name of an inference rule, and *argument list* is a set of bindings for variables in the inference rule.

The simplest atomic justification is *told information*: a relationship holds because it was explicitly asserted by the KB builder as part of a declaration:

A $\Longrightarrow$ (**and** (**prim** GOOD WINE) (**at-least** 3 grape)) **because** ToldInfo

We also report ToldInfo as the reason why A is subsumed by each of the syntactically occurring conjuncts[3] of its definition:

A $\Longrightarrow$ (**at-least** 3 grape) **because** ToldInfo.

Concept definitions and the necessary conditions on primitives are expanded by the "inheritance" inference: If a concept B is defined as (**and** A ...), then one reason why Bs have at least 3 grapes is because of inheritance through A:

B $\Longrightarrow$ (**at-least** 3 grape) **because** inheritance(δ=A).

Inheritance is a special case of the inference we introduced as transitivity, where the intermediate description must be a named concept. This can be written as:

$$\text{Inheritance} \quad \frac{\vdash \beta \Longrightarrow \delta\ ,\ \vdash \delta \Longrightarrow \lambda}{\vdash \beta \Longrightarrow \lambda} \quad \delta \text{ is a named concept.}$$

[3] Syntactically occurring parts of conjuncts work similarly. If B = (**all r** (**and** C D)), then B $\Longrightarrow$ (**all r** C) because ToldInfo.

The restriction that δ is a named concept is written as a "side condition" of the rule. It must hold before the rule can be applied. (Details of what is appropriate to relegate to side conditions are provided in Section 3.5.)

Continuing with the example above, suppose an additional concept, C, is introduced as

C $\equiv$ (**and** B ...).

Then C $\Longrightarrow$ B $\Longrightarrow$ A, so C would have (**at-least** 3 grape) by inheritance through both B and A, whether or not B stated anything explicitly about the grape role. We may therefore limit inheritance to report only from the concept that contains the description as "told information":

C $\Longrightarrow$ (**at-least** 3 grape) **because** Inheritance(δ=A).

Another atomic justification is based on the All rule:

(**all** wines RED-WINE) $\Longrightarrow$ (**all** wines WINE)

because All(π=wines, β=RED-WINE, δ=WINE).

Observe that this is not a complete explanation since there is an intermediate explanandum: why was RED-WINE subsumed by WINE? To answer this, the user may request a separate atomic justification or may enter another mode where the system will automatically ask all appropriate follow-up questions. (See Section 3.5.3 for more details.) From the above example we can see that atomic justifications naturally *chain* backward, until one stops at:

1. inference rules without antecedents (e.g., told information),

2. rules whose antecedents are theorems of mathematics (e.g., AtLst),

3. user-specified rules that are deemed unnecessary or stopping points (e.g., in CLASSIC explanation chains can be stopped when a user-defined rule fires).

There may be cases where there are multiple justifications (e.g., inheritance from several ancestors). For this, one might allow multiple (disjunctively branching) explanation chains. Of course, if an inference rule has multiple antecedents, the explanation "chain" needs to branch conjunctively. Because it is possible for explanation chains to become long and branching, we provide user control of chaining.

To conclude, we review the example from Section 3.3. Explaining 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:

`A` $\Longrightarrow$ (**prim** `WINE`)

because Prim($\tilde{\theta}$={`WINE`,`GOOD`}, $\tilde{\alpha}$={`WINE`})

`A` $\Longrightarrow$ (**at-least** `2 grape`)

because AtLst(n=3,m=2,π=`grape`).

For readability, in applications such as [91] we associate templates with the various inference rules (and patterns of inference rule applications), so that the above justification is actually reported as

"AtLeast-Ordering" on role `grape`: `3` is greater than `2`.

Our work on such "surface" presentation is still preliminary.

3.5 Discussion of Inference Rules for Explanation

As mentioned earlier, the form of the inference rules affects the brevity and naturalness of proofs and explanations. In this section, we will discuss the form of the rules of inference actually used for explanations as a way of providing a more general approach to developing explanation systems for normalize-compare DLKRS.

Since subsumption is implemented as a normalization process followed by a structural comparison process (Section 2.3), it is appropriate that proofs of subsumption be done in two phases. If we approach a proof as first presenting the last step in the deduction chain, a subsumption question may be answered by stating what structural comparison rule was applied as the last step, optionally, followed by the explanations of the steps that allowed the system to deduce the values of the arguments used. For example, the reason that (**fills** `r A B`) $\Longrightarrow$ (**at-least** 1 r) is **because** AtLst(n=2,m=1,π=r). It may be obvious to the user why an (**at-least** 2 r) was inferred on the subsumed description (which was the result of a normalization inference), or it may be necessary to explain that inference too. In this section, we will consider the form of the structural comparison rules, the normalization rules, and their interactions. This is important since the *form* of the inference rules determine the automatic generation of the explanation structures and of the appropriate follow-up questions.

3.5.1 Structural Subsumption Rules

Structural subsumption rules, as defined in [40], usually determine when an expression of the form (K α) subsumes an expression of the form (K θ), where K is one constructor, while α and θ range over appropriate arguments for the K constructor. In Figure 3.2, the structural subsumption rules include: AtLst, AtMst, All, and Prim. The general form of a structural subsumption rule is:

$$\textsf{Structural-Subsump} \quad \frac{\vdash \{\mathit{relationships\ containing\ Args1\ and/or\ Args2}\}}{\vdash (\mathit{Constructor}\ \{\mathit{Args1}\}) \Longrightarrow (\mathit{Constructor}\ \{\mathit{Args2}\})} \quad \text{(side conditions)}$$

We require the relationships in the numerator to be subsumption relationships and relegate mathematical relationships to "side conditions". Side conditions must be satisfied in order for the rule to be applicable, but we distinguish them because the side conditions will normally not have any follow-up within our explanation system. A typical side condition is exemplified in the **at-least** and **at-most** rules where there is an ordering relationship between two arguments, for example, given that $2 > 1$, (**at-least** 2 r) $\Longrightarrow$ (**at-least** 1 r). The structural subsumption rules for Mini-CLASSIC in the new format appear in Figure 3.4.

As we noted, all subsumption queries can be answered referring initially to only structural subsumption rules. For example, consider asking why `D-MERITAGE` (`DMER`) is subsumed by `AMERICAN-BLENDED-WINE (ABW)` when:

`ABW` ≡ (**and** (**prim** `WINE`)
(**all** `location AMERICAN-REGION`)
(**at-least** 2 `grape`))

`DMER` ≡ (**and** (**prim** `WINE`)

$$\text{All} \qquad \frac{\vdash \beta \Longrightarrow \delta}{\vdash (\textbf{all}\ \pi\ \beta) \Longrightarrow (\textbf{all}\ \pi\ \delta)}$$

$$\text{Prim} \qquad \frac{}{\vdash (\textbf{prim}\ \tilde{\theta}) \Longrightarrow (\textbf{prim}\ \tilde{\alpha})} \quad \tilde{\alpha} \subseteq \tilde{\theta}$$

$$\text{AtLst} \qquad \frac{}{\vdash (\textbf{at-least}\ n\ \pi) \Longrightarrow (\textbf{at-least}\ m\ \pi)} \quad \mathrm{n} \geq \mathrm{m}$$

$$\text{AtMst} \qquad \frac{}{\vdash (\textbf{at-most}\ m\ \pi) \Longrightarrow (\textbf{at-most}\ n\ \pi)} \quad \mathrm{n} \geq \mathrm{m}$$

Figure 3.4: Structural Subsumption Rules for Mini-CLASSIC

(**all** `location CALIFORNIA-REGION`)

(**at-least** 3 `grape`)

(**all** `grape`

(**one-of** `Cab-Sauvignon Cab-Franc Malbec Petite-Verdot Merlot`))).

There are three atomic descriptions in the subsuming concept; thus, there are three parts to the subsumption answer:

`DMER` $\Longrightarrow$ (**prim** `WINE`)

because Prim($\tilde{\theta}$={`WINE`}, $\tilde{\alpha}$={`WINE`})

`DMER` $\Longrightarrow$ (**all** `location AMERICAN-REGION`)

because All(δ=`AMERICAN-REGION`,β=`CALIFORNIA-REGION`, π=`location`)

`DMER` $\Longrightarrow$ (**at-least** `2 grape`)

because AtLst(n=3,m=2,π=`grape`)

assuming the obvious definitions for `AMERICAN-REGION` and `CALIFORNIA-REGION`.

3.5.2 Normalization Rules

Normalization involves making explicit all implicit information. For example, having a filler for a role implies an **at-least** 1 restriction on that role. Many times normalization rules involve more than one constructor, as in the case of the "`FILLS-IMPLIES-ATLEAST`" (FIA) rule mentioned above:

$$\text{FIA} \qquad \frac{\vdash \beta \Rightarrow (\textbf{fills}\ \pi\ \tilde{\theta})}{\vdash \beta \Rightarrow (\textbf{at-least}\ n\ \pi)} \quad |\tilde{\theta}| = \mathrm{n}$$

$$\text{Trans} \quad \frac{\vdash \beta \Longrightarrow \delta\,, \;\; \vdash \delta \Longrightarrow \lambda}{\vdash \beta \Longrightarrow \lambda}$$

$$\text{BdsCnflct} \quad \frac{\vdash \beta \Longrightarrow (\textbf{at-most}\ m\ \pi)\ \vdash \beta \Longrightarrow (\textbf{at-least}\ n\ \pi))}{\beta \Longrightarrow NOTHING} \quad n > m$$

$$\text{AllNothing} \quad \frac{\vdash \beta \Longrightarrow (all\ \pi\ NOTHING)}{\vdash \beta \Longrightarrow (\textbf{at-most}\ 0\ \pi)}$$

$$\text{AllNothing} \quad \frac{\vdash \beta \Longrightarrow (\textbf{at-most}\ 0\ \pi)}{\vdash \beta \Longrightarrow (\textbf{all}\ \pi\ NOTHING)}$$

$$\text{All-and} \quad \frac{\vdash \beta \Longrightarrow (\textbf{and}\ (\textbf{all}\ \pi\ \beta)\ (\textbf{all}\ \pi\ \delta)...)}{\vdash \beta \Longrightarrow (\textbf{all}\ \pi\ (\textbf{and}\ \beta\ \delta)}$$

$$\text{PrimAnd} \quad \frac{\vdash \beta \Longrightarrow (\textbf{and}\ (\textbf{prim}\ \tilde{\lambda})\ (\textbf{prim}\ \tilde{\eta})\ ...)}{\vdash \beta \Longrightarrow (\textbf{prim}\ \tilde{\lambda}\ \tilde{\eta})}$$

Figure 3.5: Normalization Rules for Mini-CLASSIC

Another CLASSIC normalization rule says that if the **at-least** restriction on a role equals the number of elements in a **one-of** restriction on that role, then the role must be filled with *all* the elements of the **one-of**. We call this `ATLEAST-AND-ONEOF-IMPLIES-FILLERS` (`AOIF`), and write it as follows:

$$\text{AOIF} \quad \frac{\vdash \beta\Rightarrow\ (\textbf{at-least}\ n\ \pi), \quad \vdash \beta\Rightarrow\ (\textbf{all}\ \pi\ (\textbf{one-of}\ \tilde{\theta}))}{\vdash \beta\Rightarrow\ (\textbf{fills}\ \pi\ \tilde{\theta})} \quad |\tilde{\theta}| = n$$

Another inference rule involving **one-of** can be called `ONEOF-IMPLIES-ATMOST` (OIA). It says if there is a **one-of** restriction on a role, and that **one-of** has n elements, then there should be an **at-most** restriction of n on the same role. This can be written:

$$\text{OIA} \quad \frac{\vdash \beta\Rightarrow\ (\textbf{all}\ \pi\ (\textbf{one-of}\ \tilde{\theta}))}{\vdash \beta\Rightarrow\ (\textbf{at-most}\ n\ \pi)} \quad |\tilde{\theta}| = n$$

The general form of a normalization rule is:

$$\text{Normalization Rule} \quad \frac{\vdash \beta\Rightarrow\theta,\ ...,\ \beta\Rightarrow\alpha,\ additional\ relationships}{\vdash \beta\Rightarrow\delta} \quad \text{side conditions}$$

The previous normalization rules were examples drawn from the full CLASSIC system. The normalization rules for Mini-CLASSIC appear in Figure 3.5. Mini-CLASSIC does not need many normalization rules because of the simplicity of its grammar. However, the rules do show how the **all** and **at-most** constructors interact when the **at-most** restriction is 0. In a more complex DL, typically many more constructors interact (as we saw with the FIA and AOF rules), and for full CLASSIC, there are many more normalization rules than subsumption rules.

To review matters: asking for an explanation of the subsumption relationship between `DMER` and the expression (**at-least** 2 `grape`) really has two parts:

1. Does the correct ordering relationship exist between the **at-least** restrictions? (i.e., is `DMER`'s **at-least** restriction greater than or equal to 2?)

2. How did `DMER` get its **at-least** restriction?

The first question uses the structural subsumption rules, while the second uses normalization rules. In this case, the system was told that `DMER` has (**at-least** 3 `grape`) so there is a trivial normalization reason – "told information". More generally, suppose we add the following new concept definition for five varietal meritage:

`5VARMER` $\equiv$ (**and** `DMER` (**at-least** 5 `grape`)).

Now, if one asks how `5VARMER` got its fillers for the `grape` role, a DL could answer

`5VARMER` $\Longrightarrow$ (**fills** `grape Cab-Sauvignon Cab-Franc Malbec Petite-Verdot Merlot`)

because

`AOIF`(n=5, π=grape, $\tilde{\theta}$={Cab-Sauvignon Cab-Franc Malbec Petite-Verdot Merlot}).

3.5.3 Follow-up Questions

In an incremental approach, our explanations first present the last rule used in the deductive process and then answer appropriate automatically generated or user initiated follow-up questions. We have noted that all subsumption questions can be answered with an initial structural subsumption rule, thus we should look first at structural subsumption follow-up questions.

Following-up Structural Subsumption Rules

Let us continue with the example in Section 3.5.1, which explains why `D-MERITAGE` is an `AMERICAN-BLENDED-WINE`. The follow-up questions can be generated from the form of the inference rule used. For each rule used, we consider asking why all of the relationships in the *numerator* (or the side condition) of the inference rule hold. In this example,

(1) why {WINE} is a subset of {WINE} in the Prim rule, (2) why CALIFORNIA-REGION is subsumed by AMERICAN-REGION in the All rule, or (3) why $3 \geq 2$ in the AtLst rule. As seen in questions (1) and (3), side condition follow-ups refer to partial orders in mathematics like subset or $\geq$, and thus may be viewed as not needing further explanation. Follow-up question (2) is another subsumption question.

The denominator of the inference rule also needs to be considered in follow-up questions since it is always the source of one *normalization question*: why did the object being subsumed obtain the property mentioned? In the example above, this results in follow-up questions about how DMER obtained its:

1. set of primitives,
2. value restriction on the `location` role,
3. **at-least** restriction of 3 on the `grape` role.

These are all questions about how the subsumed concept was normalized. (In this simple example, the answer to all three questions is "told-info".) Similarly, questions about how the subsumer concept was normalized could be considered.[4] In practice, we found it useful to be able to automatically explain the normalization (or "origin" in CLASSIC terms) of each property mentioned on the subsumed object in a subsumption rule. Thus, the CLASSIC implementation supports an option on subsumption explanation that will answer the appropriate normalization questions too. In one demonstration system containing a graphical interface[91], we provide explanations and object presentations with mouse-sensitivity for all "interesting"[5] properties that have been derived. Pop up menus associated with the mouse-sensitive region include an option for explaining how the object got that property.

[4]We do not suggest doing this follow-up automatically since subsumption questions can be answered by asking why the subsumed concept is subsumed by each atomic description in the *told* information on the subsuming concept, thus making normalization questions on the subsumer concept redundant.

[5]Interestingness is determined by using a limited version of the meta-language presented in Chapter 6.

Following-up Normalization Rules

Follow-up questions for normalization are generated from asking why the object, β, has each of the properties mentioned in the numerator (and asking why the other relationships hold if they are not just mathematical statements). At the end of section 3.5.2, we saw the AOIF inference rule used to explain why 5VARMER had particular fillers for its grape role. The specific application was **because**

AOIF(n=5, π=grape, $\tilde{\theta}$={Cab-Sauvignon Cab-Franc Malbec Petite-Verdot Merlot}).

In this case, the follow-up questions are (i) how did 5VARMER get its **at-least** restriction on grape, and (ii) how did it get its **all** restriction on the same role. The answers are that it got the **at-least** restriction because that was part of its told definition and it got the **all** restriction because of inheritance from DMER. Another follow-up question then is how did DMER get its **all** restriction. As in this case, follow-up questions may be generated continually until the answer "bottoms out" because the information was told to the system. Our implementation provides a "complete explanation" function that will ask and answer all the follow-up questions.

3.5.4 Special Inference Rule Handling

Some inference rules deserve special handling. The first set involve constructs that are present in all description logics, such as conjunction, primitives, THING, NOTHING[6] . Concerning conjunction, we break up subsuming concepts into their component parts and explain the component subsumptions, thus eliminating the need for explaining AndR. Also, we identified a class of inference rules that we believed always added superfluous information to explanations, and classified AndL and AndEq in this group. We never use any of the **and** rules in our explanations. Thus, we do not include explanations of like B $\Longrightarrow$ (**and** B anything) **because** AndL.[7]

Another class of rules that may be omitted in explanation reporting are subsumption

[6] Most of the NOTHING rules remain but the one that explains why NOTHING is more specific than any arbitrary other concept is removed.

[7] If some class of users needed the **and** rules included, it would be simple to add them back in. In fact, our implementation in NeoClassic [99] includes optional explanations of these rules just to provide a more complete set of inferences.

relationships with THING and NOTHING. Since stating that a concept is subsumed by THING, is equivalent to saying nothing new about the object, we do not explain the inference that THING subsumes every concept. We also do not explain why it is a primitive on every object. Similarly, we do not explain the inference that NOTHING is subsumed by every other concept.

We also found it useful to provide special handling for named concepts. One approach to handling named concepts is to first expand all named concepts before explaining anything. This is a correct and general approach, however, named concepts may provide a natural grouping of restrictions that may be useful when kept intact. We suggest not decomposing named concepts when the concept appears alone in a value restriction. For example, consider the following KB:

A ≡ (**and** (**at-least** 2 r) (**at-most** 5 r))

B ≡ (**all** s A)

C ≡ (**and** B ...)

If the system is asked to explain C's value restriction on role s, we could simply ask why C $\Longrightarrow$ (**all** s A), and a simple explanation could be given:

C $\Longrightarrow$ (**all** s A) **because** of inheritance from B.

If A was always expanded, this question itself would have to rephrased as asking why C $\Longrightarrow$ (**all** s (**at-least** 2 r)) and why C $\Longrightarrow$ (**all** s (**at-most** 5 r)).

Additional rules, such as reflexivity (Ref) and substitution (Eq) (but not transitivity, which corresponds to inheritance) have been placed in this class of unexplained inferences since we considered these inferences to be obvious in all applications.

In addition to the above, there may be a number of special rules dealing with the constructors specific to the DL at hand. For example, there may be a number of ways of expressing THING. In Mini-CLASSIC,

THING ≡ (**at-least** 0 r) ≡ (**at-most** MAXINT r) ≡ (**all** r THING) ≡ (**prim** THING) ≡ (**and**) ≡ (**and** THING).

Based on practical experience, we chose not to explain any of the THING equivalences.

Similarly, there are a number of inferences describing incoherent concepts (e.g., the bounds conflict inference in mini-CLASSIC), which are equivalences to `NOTHING`. Since a derivation of `NOTHING`, unlike a derivation of `THING`, is adding new information to an object, these inference are not omitted from an explanation. There are 11 such conflict inferences in full CLASSIC, listed in Appendix C. Conflicts will be covered in much more detail in our presentation of errors in section 4.4.

There may be additional system–dependent concepts worth adding to the set of inferences not explained (e.g., the primitive `CLASSIC-THING` in CLASSIC) or domain–specific concepts that users should be able to add. The CLASSIC implementation provides user control over adding the system-dependent concepts and control using meta-information for adding domain–specific concepts that should not be used in explanations. This topic will be covered in more detail in Chapter 6.

If a system has non-structural subsumption deductions, such as CLASSIC's **same-as** deductions, then explanations may be less precise. Our explanation implementation does not recreate the graph-based CLASSIC **same-as** algorithms and thus does not differentiate between some **same-as** restrictions which were told versus others which were derived. See Appendix B for the implemented **same-as** inference rules. Our work presented in [90] includes a discussion of a more general approach to the problem.

The rules that are handled specially for Mini-CLASSIC appear in Figure 3.6.

3.6 Developing an Explanation System

Suppose we wish to build an explanation system for some other DL, based on the preceding theory. To do so, we need to accomplish the following tasks:

- Identify a (sub)language of atomic descriptions.
- Present "atomization" rules for normalizing a description into atomic conjuncts.
- Identify rules of inference for subsumption.
- Develop algorithms for (re-)constructing subsumption proofs.

We will address these issues, assuming that there is already an implementation of subsumption reasoning for the DL, along the lines described for CLASSIC in Section 2.7.1.

AndR	$\dfrac{\vdash \beta \Longrightarrow \delta, \ \vdash \beta \Longrightarrow (\textbf{and}\ \tilde{\lambda})}{\vdash \beta \Longrightarrow (\textbf{and}\ \delta\ \tilde{\lambda})}$	
AndL	$\dfrac{\vdash \beta \Longrightarrow \delta}{\vdash (\textbf{and}\ \ldots\ \beta \ldots) \Longrightarrow \delta}$	
AndEq	$\vdash (\textbf{and}\ \ \beta) \equiv\ \beta$	
Ref	$\vdash \beta \Longrightarrow \beta$	
Eq	$\dfrac{\vdash \lambda{\equiv}\eta \qquad \vdash \beta \Longrightarrow \delta}{\vdash \beta' \ \ \Longrightarrow \delta'}$	β',δ' = β,δ with one or more occurrences of λ replaced by η.
EmptyAnd	$\vdash \texttt{THING} \equiv\ (\textbf{and})$	
AllThing	$\vdash \texttt{THING}\ \equiv\ (\textbf{all}\ \pi\ \texttt{THING})$	
AtLs0	$\vdash \texttt{THING}\ \equiv\ (\textbf{at}-\textbf{least}\ 0\ \pi)$	
AtMsMax	$\vdash \texttt{THING} \equiv (\textbf{at}-\textbf{most}\ \texttt{MAXINT}\ \ \pi)$	
Nothing	$\vdash \texttt{NOTHING}\ \Longrightarrow \beta$	
Thing	$\vdash \beta \Longrightarrow \texttt{THING}$	

Figure 3.6: Other Rules

It is also useful to have an initial set of inference rules for the proof theory of the description logic in question. Royer and Quantz [106] describe an interesting systematic technique for obtaining inference rules for a DL from its translation to first order logic. One can also analyze the DLKRS implementation, looking for all updates to data structures, and expressing these as inference rules. This is the approach we have followed for CLASSIC, thus describing the actual (incomplete) implementation (see Section 3.6.3 for a discussion of these issues.)

3.6.1 Atomic Descriptions and Atomization

To define the grammar of atomic descriptions, one may begin with the grammar of the concept language, and eliminate those constructs that can be written as conjunctions of other, more general descriptions. Such constructs might be signaled by the presence of inference rules involving **and**, or having the form

$$\frac{\vdash C \Longrightarrow \ldots \qquad \vdash C \Longrightarrow \ldots}{\vdash C \Longrightarrow \ldots}.$$

For example, in a system that allows any number of individuals in a **fills** expression, there will be an inference rule of the form:

$$\frac{\vdash C \Longrightarrow (\textbf{fills}\ r\ A) \qquad \vdash C \Longrightarrow (\textbf{fills}\ r\ B)}{\vdash C \Longrightarrow (\textbf{fills}\ r\ A\ B)}$$

or of the form:

$$\frac{\vdash C \Longrightarrow \textbf{(and (fills } r\ A) \textbf{ (fills } r\ B) \ ...\ )}{\vdash C \Longrightarrow \textbf{(fills } r\ A\ B)}.$$

Both of these forms are warnings that the expression in the rule consequent may not be an appropriate atomic description form.

Figure 3.7 contains a grammar of atomic descriptions for Mini-CLASSIC. The limitations introduced by this grammar are:

1. no **and** constructor,
2. **prim** only takes one argument, and
3. **all** allows only atomic descriptions instead of general descriptions.

<atomic-descr> ::= `THING` | `NOTHING` |
(**at-least** <integer> <role-name>) |
(**at-most** <integer> <role-name>) |
(**prim** <identifier>) |
(**all** <role-name> <atomic-descr>)

Figure 3.7: Atomic Description Grammar for Mini-CLASSIC

One can verify that this grammar has the property that every description is equivalent to the conjunction of some set of atomic descriptions. If such a proof is constructive, it will usually identify a subset of inference rules that are needed for converting to normal form (see [90]). For example, the rule that breaks apart **fills** expressions:

$$\frac{\vdash C \Longrightarrow \textbf{(fills } r\ A\ B...)}{\vdash C \Longrightarrow \textbf{(fills } r\ A)}$$

will be needed.

For DLKRSs using a Normalize-Compare algorithm for subsumption, the normal form of a concept can be mapped onto a set of atomic descriptions relatively easily, and hence forms a good basis for the atomization process. For example, CLASSIC's normal form is not quite atomic because it contains nested conjunctions, as in (**all** `r` (**and** `C` `D`)), and implicit conjunctions such as (**fills** `r` B_1 B_2). However, a simple routine can be written to break apart such conjunctions to yield atomic descriptions.

Figure 3.8 contains the additional expressions necessary to define atomic descriptions for full CLASSIC. The only additional limitations it introduces over the additions needed for the full CLASSIC is that **fills** only takes one argument.

<atomic-descr'> ::= {CLASSIC | HOST}-THING |
(**one-of** <ind-name>*) |
(**test** <test-name>) |
(**fills** <role-name> <ind-name>) |
(**same-as** (<att-name>+) (<att-name>+)) |
(**min** <number>) |
(**max** <number>)

Figure 3.8: Additions to the Atomic Description Grammar for CLASSIC.

3.6.2 Recipes for Identifying Atomic Descriptions

The designer of an explanation facility needs to define the form of an atomic description for his or her particular knowledge representation language. We offer the following heuristics for identifying this, using the grammar of the language, its semantic specification (including inference rules) and/or an implementation.

We shall call *raw atomic descriptions* ones which have at most one concept constructor at the top level, and have no embedded non-atomic identifiers. To get such descriptions, we must therefore eliminate defined concepts and the **and** construct. Raw atomic description form will typically be encoded in the data structures of the DLKRS, since it provides an obvious way of organizing the internal data structures.

Once conjunctive constructors and defined concepts are removed, the explicit sources of conjunction have been removed. In order to make raw atomic descriptions into atomic descriptions, we need to try to eliminate implicit sources of conjunction. First, consider constructors whose syntax does not have nested concepts; in full CLASSIC, this includes **one-of**, **testh**, **min**, **max**, **testc**, **at-least**, **at-most**, **fills**, and **same-as**. If such a constructor takes at most one argument, which is itself indivisible, then the raw atomic description is an atomic description.[8] In full CLASSIC, constructors in this category

[8] Macros which expand to a conjunction are of course omitted. For example, (**exactly** 2 **r**) = (**and**

include **min**, **max**, **at-least**, and **at-most**.

A constructor taking more than one argument or taking a composite argument needs to be checked for implicit conjunctions. The **fills** constructor is implicitly conjunctive since (**fills** `r A B`) = (**and** (**fills** `r A`) (**fills** `r B`)), but **one-of**, **testh**, **testc**, and **same-as** are not since their arguments are indivisible semantically.

Next consider any recursive constructors. In full CLASSIC, after eliminating non-nested **and**s, the only construct left is **all**. In order to make **all**'s arguments conjunction-free, <descr> in the DL grammar needs to be changed to the non-conjunctive <atomic-descr>.

3.6.3 Finding Subsumption Rules and Atomic Justifications

An explanation designer needs to identify the inference rules and the appropriate arguments that will be *reported* in explanations. It is wise to begin with a complete set of the inference rules required to derive all subsumptions.

In order to control the verbosity of explanations, the developer may choose to limit the set of inferences that will actually be *reported* in explanations. This may mean skipping certain inferences, modifying inferences, or merging two or more inferences together for the purposes of reporting. In CLASSIC, for example, we reduced the set of inference rules from the 100 or so original ones, to about half that number used in (the default settings for) our explanations. We also gave the user the option to add or delete some inferences to the set that will be reported. For example, we have already mentioned that we eliminated the rules associated with conjunction, and reflexivity, and some of the rules associated with `THING` and `NOTHING`. We also modified inferences, such as limiting transitivity by requiring a named intermediate description (calling the inference inheritance).

In reporting an atomic justification, the implementor may choose to eliminate arguments that appear obvious. In CLASSIC we generally do not report arguments that appear in the denominator of the inference rule. For example, the Prim rule is reported

---

(**at-least** 2 **r**) (**at-most** 2 **r**)), so **exactly** should be eliminated and the grammar should contain **at-least** and **at-most**.

only by name since all of its arguments appear in the denominator. Additionally, we do not report arguments if their value is unambiguous. For example, if one concept has role fillers `A, B,` and `C` and another has role fillers `A` and `B`, then the subsumption relationship holds because of the filler-subset relationship. A verbose mode could go on to state that {`A,B`} is a subset of {`A,B,C`}. It is unambiguous as to the fillers of the roles - the entire set fills the role. We do report argument values if there is one particular object from a set that allows an inference to be triggered. For example, an object may have inherited a property from one of its parents. It is important to identify which of the many parents the information was inherited from. Additionally filtering of what is interesting to print or explain is controlled by the meta-language described in detail in Chapter 6.

3.6.4 The Explanation Construction Process

Typically, it requires too much space and effort to maintain enough information to allow a reconstruction of a formal subsumption or normalization proof for every deduction in a DLKRS. Thus, we expect that any extensive explanation facility will need to reconstruct proofs on demand, by "replaying" the normalization and subsumption algorithms. To do this, we need to choose between augmenting the core system code or writing separate explanation modules. In a tradeoff between impact on the core system and extra code that must be maintained, we agree with Swartout and Moore [117] and choose to minimize dual algorithm development. Thus, we have instrumented the main code to store more information when it is *replayed in explanation mode*.

Our implementation allows explanations to be requested that are either structural subsumption only, normalization only, or a combination of the two. If a single structural subsumption question is asked, only the subsumption algorithm needs to be rerun so that it can identify which structural subsumption rules were applied. If a normalization question is asked, the normalization code will need to be rerun[9] so that it can identify the (potentially many) normalization inferences that have been applied. In order to

[9]There are some cases where the algorithm is smart enough to deduce the answer from the current structures, so an optimization can be done to avoid rerunning normalization for some questions.

rerun the normalization code, the system first destroys certain derived information so that it can be rederived in a more verbose mode. This normalization code has been modified to include a rerun mode that calls an explanation update function whenever the system applies an inference rule that should be reported. The explanation update functions build structures that record the proof structure, and it is from these structures that the final explanations are generated. The only modifications to the core system are an extra pointer in the main data structures (for the explanation structure) and the rerun mode calls to explanation functions.

Let us illustrate this with a simple example. Given the definition

`A` ≡ (**and** (**at-least** 2 `favorites`)
(**all** `favorites` (**one-of** `Sony Toshiba`))

the told information can be depicted as

concept:	`A`	
restrictions		
on roles:	`favorites`	
	at-least:	2
	one-of:	{ `Sony Toshiba` }

The derived structure would have additional information now with a pointer to an explanation structure:

concept:	`A`
explanation structure:	pointer to the explanation structure
primitive	

superconcepts:	`CLASSIC-THING`	
restrictions		
on roles:	`favorites`	
	at-least:	2
	at-most:	2
	fillers:	`Sony`
		`Toshiba`
	primitive	
	superconcepts:	`CLASSIC-THING`
	one-of:	{ `Sony Toshiba` }

---

and the explanation structure mirrors the derived structure but with (one or more) inference(s) justifying each piece of information:

---

concept:	`A`		
primitive			
superconcepts:	`CLASSIC-THING`		
			ROLE-RESTR-IMPLIES-CLASSIC-THING
restrictions			
on roles:	`favorites`		
	at-least:	2	
			TOLD
	at-most:	2	
			ONE-OF-IMPLIES-AT-MOST
	fillers:	`Sony`	
			ATLEAST-AND-ONEOF-IMPLIES-FILLERS

	`Toshiba`
	ATLEAST-AND-ONEOF-IMPLIES-FILLERS
primitive superconcepts:	`CLASSIC-THING`
	ONEOF-IMPLIES-CLASSIC-THING
one-of:	{ `Sony Toshiba` }
	TOLD

From these structures, presentations of the explanations may be generated. In our applications, we have experimented with simple pretty printing of the structures, natural language, and interactive mouse-sensitive graphical presentations.

3.7 Summary

As a foundation for explanations of subsumption, we propose the use of a *deductive framework* based on proof rules in the "natural semantics" style. As part of this plan, we presented the rules of inference for a simple description logic, while the complete set of inference rules for CLASSIC is relegated to Appendices A– C.

We also showed how formal proofs of subsumptions are like explanations, albeit ones that are too onerous for users to read because of a surfeit of details. The advantage of the natural deduction style is that the resulting proofs have the familiar form of backward chaining trees.

In order to simplify and break up explanations, we decompose concepts into atomic descriptions, for which subsumption is explained independently. Furthermore, rather than presenting complete proofs trees, we explain individual nodes in the proof tree through atomic justifications, which make symbolic references to the antecedents, thereby allowing the user to choose if and when proofs for these should be presented.

We provided a discussion of the form of the classes of inference rules. Using the logical form of applied rules, we showed how follow-up questions may be generated automatically. Finally, we drew on our implementation and application experience to provide some general guidelines for developing explanation components for . In particular, rather than developing an entirely separate "theorem prover" for finding proofs, we suggest augmenting the DL implementation with facilities for reconstructing proofs, when these are needed.

In the next chapters, we will introduce explanations of individuals, errors, and negative deductions, followed by a presentation of our filtering methods.

Chapter 4

Explaining Recognition

The main judgement that description logics perform concerning individuals is recognition. When processing a new individual, the DL will need to determine if it is recognized to be an instance of existing concepts, just as when processing a new concept one needs to determine if it is subsumed by existing concepts. This analogy between recognition and subsumption is in fact supported by a technique that implements recognition by appropriately augmenting the concept subsumption algorithms. Therefore, an approach to explaining recognition algorithms could begin with the foundation for explaining subsumption and provide extensions necessary to explain the additional individual reasoning.

In this chapter we extend the foundation presented in Chapter 3 in order to handle explanations of individual processing, introducing the notion of membership proofs to parallel subsumption proofs as explanations. For this purpose, we will present inference rules necessary to allow the concept rules to work with individuals, as well as additional inference rules needed to handle individual reasoning, paying particular attention to individual closed world reasoning. In order to deal with overly-long proofs, we will use the same techniques of atomic descriptions and atomic justifications to support our explanations.

One issue that is much more important in individual processing than in concept handling is error explanations. Many DLs allow user-initiated modifications to individuals (while most DLs do not allow modifications to concepts), thus opening the possibility for more ways for users to create inconsistencies. In this chapter, we show how error explanation fits easily into our framework with the support of “intermediate objects” used to capture the state objects were in when a contradiction was detected. We also note that in error explanations, follow-up questions may be more important and we provide additional support for automatic follow-up question and answer generation.

4.1 The Knowledge Base for Individuals

As described in Section 2.4, once individuals are introduced, the knowledge base contains more than just concept declarations. In particular, there is information about individual membership in primitive and defined concepts, fillers of roles, and roles on particular individuals that are closed. Additionally, trigger rules may need to be fired on individuals.

In order to construct proofs about the recognition judgement, the DL will need to refer to the knowledge base. This will include the declarations of concepts plus additional information related to individuals, which will be captured using judgements of the form

I $\rightarrow$ <concept description>

that state that I is an instance of a certain description.

This can represent, among others,

- whether an individual belongs to some primitive concept, by having the description be the name of a primitive concept;
- whether an individual has certain fillers for a role, through the use of the **fills** concept constructors as in

  `Deb` $\rightarrow$ (**fills** `pet Panda Monium`)

- whether some role is closed on an individual, by using an **at-most** n role restriction where n is the current number of fillers.

  `Deb` $\rightarrow$ (**and** (**fills** `pet Panda Monium`) (**at-most** 2 `pet`)).

Unlike CLASSIC, some description logics do not include the **fills** constructor. In such cases, one must either introduce a new judgement, concerning role fillers, or simply allow **fills** in the knowledge base for individual membership assertions, though not for concept definitions. Similarly, if a DL does not support an **at-most** concept constructor, one needs to have a way to represent in the knowledge base that the role is closed.

4.2 Proof Theory for Individual Recognition Explanation

There are many cases where determining if an individual is an instance of a concept is almost identical to determining if a concept is subsumed by another concept. Recognizing when an individual is an instance of a description can also be done in the familiar normalize/compare paradigm. Thus, we would like the rules for structural recognition and concept normalization to be applicable to the descriptions of individuals. This could be done by making an individual rule for every concept rule. For example, we can start with the concept rule for structural subsumption of **at-least** descriptions:

$$\mathsf{AtLst}\quad \frac{}{\vdash (\mathbf{at{-}least}\ n\ \pi) \Longrightarrow (\mathbf{at{-}least}\ m\ \pi)}\quad \mathrm{n} \geq \mathrm{m}$$

and make an individual version as follows:

$$\mathsf{AtLstI}\quad \frac{\vdash \iota \longrightarrow (\mathbf{at{-}least}\ n\ \pi)}{\vdash \iota \longrightarrow (\mathbf{at{-}least}\ m\ \pi)}\quad \mathrm{n} \geq \mathrm{m}$$

Instead of making an individual version for all of the concept rules, we add a single rule that says if an individual is an instance of a description and that description is subsumed by another description, θ, then the individual is an instance of θ. This rule is actually just transitivity for individuals, and can be written as follows:

$$\mathsf{Transitivity}\quad \frac{\vdash \iota \longrightarrow \beta \quad \vdash \beta \Longrightarrow \theta}{\vdash \iota \longrightarrow \theta}$$

So, for example, if:

I $\rightarrow$ (**at-least** 2 r) and

(**at-least** 2 r) $\Longrightarrow$ (**at-least** 1 r),

then one can conclude

I $\rightarrow$ (**at-least** 1 r).

In addition, we will need individual analogs for AndR and Eq (previously defined in Figure 3.2):

$$\mathsf{AndR}\quad \frac{\vdash \iota \longrightarrow \delta,\ \ \vdash \iota \longrightarrow (\textbf{and}\ \tilde{\lambda})}{\vdash \iota \longrightarrow (\textbf{and}\ \delta\ \tilde{\lambda})}$$

$$\mathsf{Eq}\quad \frac{\vdash \lambda \equiv \eta \qquad \vdash \iota \longrightarrow \delta}{\vdash \iota \longrightarrow \delta\{\lambda/\eta\}}$$

So, if a knowledge base contained:

`I` $\rightarrow$ `A`

`I` $\rightarrow$ (**and** `B`)

`C` $\equiv$ (**and** `A B`),

we can conclude `I` $\rightarrow$ (**and** `A B`) using the AndR rule and `I` $\rightarrow$ `C` using the Eq rule.

4.2.1 Closed World Reasoning in Recognition

Individual reasoning includes new deduction rules not considered in concept reasoning. One source of such rules comes from the reasoning used to determine individual membership in concepts when complete filler information is known, as introduced in Section 2.8. Consider an individual stereo system `Ss1` with exactly two speakers, `Sp1` and `Sp2`. If it is also known that `Sp1` and `Sp2` are instances of `AMERICAN-MADE`, and that there can be no additional speakers for `Ss1`, then one could expect that:

`Ss1` $\rightarrow$ (**all** `speaker AMERICAN-MADE`).

Therefore, an inference rule is needed that looks at roles that are closed on individuals and checks to see if all their fillers satisfy a value restriction. For example, in the above case we need the `speaker` role to be closed, i.e., it needs to have as many fillers as its **at-most** restriction states it should, and we need the fillers all to be instances of `AMERICAN-MADE`. This rule can be written as follows:

$$\mathsf{FillersSatisfyAll}\quad \frac{\vdash \iota \longrightarrow (\textbf{fills}\ \pi\ \ \tilde{\kappa})\ \ \vdash \iota \longrightarrow (\textbf{at-most}\ n\ \pi)\ \ \forall \kappa \in \tilde{\kappa}\ \ \vdash \kappa \longrightarrow \eta}{\vdash \iota \longrightarrow (\textbf{all}\ \pi\ \ \eta)}\quad |\ \tilde{\kappa}\ | = \mathrm{n}.$$

Note that CLASSIC has chosen to avoid all such considerations of individual properties for the purposes of computing subsumption, though this decision may be made differently for other languages (in which case the definition of subsumption and recognition become inextricably interwoven).

The complete set of structural recognition rules for full CLASSIC is stated (in English) in Figure 2.1. As it happens for a simple language such as Mini-CLASSIC, there are very few rules and almost all of these rules are already covered in the concept subsumption rules. For example, there is a structural recognition rule that says that an individual is an instance of (**prim** Aset) if the individual is an instance of (**prim** Bset) and Bset $\supseteq$ Aset. Since we already have the Prim rule for concepts along with transitivity for individuals, there is no need to add a special primitive structural recognition rule. In fact, for Mini-CLASSIC, the FillersSatisfyAll rule is the only rule that needs to be added in order to implement the structural recognition rules given the concept subsumption rules. In full CLASSIC, there are additional rules concerning the **one-of**, **same-as**, **test**, **min**, and **max** constructors. For example, the **one-of** inference rule is as follows:

$$\text{OneofI} \quad \frac{}{db \vdash \iota \longrightarrow (\mathbf{one\!-\!of}\ \tilde{\kappa})} \quad \iota \in \tilde{\kappa}$$

The other inferences are included in Appendix B.

In those cases where the DL, unlike CLASSIC, does not include constructors that can represent information about role fillers or role closure, then inference rules involving them must be added. For example, full CLASSIC has a concept normalization rule (FIA, mentioned in Section 3.5.2) that counts the number of known fillers, n, on a role, `r` and deduces an (**at-least** n `r`) restriction on the concept. If there had been no **fills** construct allowed in concept descriptions, there would not have been a FIA rule for concepts, and it would have been needed to be added in order to process individuals.

4.2.2 Propagation

From an individual assertion, the DL can deduce information about other individuals using the notion of propagation (as introduced in Section 2.5). For example, if individual `Ss1` is known to be an instance of (**all** `speaker SPEAKER`), and it is known that `Sp89` fills the `speaker` role on `Ss1`, then the propagation rule can be used to deduce that `Sp89` is an instance of the `SPEAKER` concept.

This inference is captured by the following rule:

$$\text{Propagation}\quad \frac{\vdash \iota \longrightarrow (\mathbf{all}\ \pi\ \eta) \qquad \vdash \iota \longrightarrow (\mathbf{fills}\ \pi\ \kappa)}{\vdash \kappa \longrightarrow \eta}$$

4.2.3 Trigger Rules

As mentioned in Section 2.5, CLASSIC and other DLs allow "trigger rule" or "productions" to be attached to a concept. Once an individual is discovered to be an instance of such a concept, the trigger "fires" on the it, and the individual is asserted to be an instance of the consequent of the rule.

For example, if an individual stereo system, `SS3`, is known to be an instance of a `HIGH-QUAL-HOME-THEATER-SYS`, which in turn has the following trigger rule, called `TV-RULE`, attached to it

`HIGH-QUAL-HOME-THEATER-SYS` ≈>(**all** `tv BIG-TV`),

then the trigger rule can be used to deduce that `SS3` is an instance of (**all** `tv BIG-TV`).

The corresponding inference rule is:

$$\text{Trigger}\quad \frac{\vdash \iota \longrightarrow \delta\ ,\ \ \vdash \mathit{identifier}\!: \delta \approx\!> \lambda}{\vdash \iota \longrightarrow \lambda}$$

4.2.4 Closing Roles

As mentioned in Section 2.6, some DLs allow users to state that a role is closed on a particular individual. In CLASSIC, the function (cl-ind-close-role <I> <r>) can be used to state that all of the fillers are known for the role `r` on individual `I`. Operationally, this means that the system should count the current fillers of `r`, and assert that count as an **at-most** restriction for the role. For example, if `SS4` $\rightarrow$ (**fills** `speaker A B`) and the `speaker` role is closed on `SS4`, then the DL should assert (**at-most** 2 `speaker`) on `SS4`. We will report this inference as "ToldClosed" and write it as follows:

$$\text{ToldClosed}\quad \frac{\vdash \iota \longrightarrow (\mathbf{fills}\ \pi\ \tilde{\kappa})}{\vdash \iota \longrightarrow (\mathbf{at\!-\!most}\ n\ \pi)} \quad \begin{array}{l}\mid \tilde{\kappa} \mid = n,\ \tilde{\kappa} \text{ is the largest set such}\\ \text{that } \iota \rightarrow (\mathbf{fills}\ \pi\ \tilde{\kappa}) \text{ is true, and}\\ \text{the role } \pi \text{ is told to be closed on}\\ \iota.\end{array}$$

The simple thing to do with this rule is to determine when an individual is an instance of some **at-most** restriction: Consider an application with:

I $\rightarrow$ (**fills** r A B C).

If information is given that r is closed on i, then a DL can deduce

I $\rightarrow$ (**at-most** 3 r) using the ToldClosed rule.

Another (indirect) use of closed is in determining when an individual is an instance of an **all** restriction. For example, if A, B, and C are all instances of SPEAKER in the example above, then the FillersSatisfyAll rule can be applied because the **at-most** restriction was deduced, and the DL can deduce:

I $\rightarrow$ (**all** r SPEAKER).

4.3 Recognition Explanations as Proofs

Once the additional rules covering recognition have been added, proofs can be provided as explanations of recognition. Let us look at some typical examples of individual reasoning. Consider the following knowledge base with the objects: HTS (Home Theater System), SS1 (an individual HTS), and HHTS (High Home Theater System):

HTS $\equiv$ (**and** SS (**at-least** 1 tv) (**all** tv TELEVISION))

SS1 $\rightarrow$ (**and** HTS (**fills** tv Mytv))

HHTS $\equiv$ (**and** HTS (**all** price (**min** 6000))).

Additionally, let there be a "HIGH-TV-RULE" associated with HHTS that says that instances of HHTSs have televisions whose diagonals are at least 27 inches.

HIGH-TV-RULE :

antecedent: HHTS

consequent: (**all** tv (**all** diagonal (**min** 27)))

This example was chosen because it can make use of typical individual inferences – i.e., rule firings and propagations. One typical pattern of inference used in individual processing involves:

- Recognizing that an individual is an instance of a concept (which is the antecedent of a rule).
- Firing rules on that individual.
- Propagating information onto other individual(s) as a result of information newly discovered about the initial individual.

For example, if we are also told that `SS1` costs at least 8000, CLASSIC will discover that `Mytv` has a `diagonal` that is at least 27 inches. The following is a proof of why `Mytv`'s `diagonal` has that restriction, using numbered lines and inference rule justifications:

1 `SS1` $\rightarrow$	(**all** `price` (**min** 8000))	Told
2 (**min** 8000) $\Longrightarrow$ (**min** 6000)		MinOrdering
3 (**all** `price` (**min** 8000)) $\Longrightarrow$ (**all** `price` (**min** 6000))		All 2
4 `SS1` $\rightarrow$	(**all** `price` (**min** 6000))	Trans 1, 3
5 `SS1` $\rightarrow$	(**and** `HTS` (**fills** `tv` `Mytv`))	Told
6 `HTS` $\Longrightarrow$	`HTS`	Ref
7 (**and** `HTS` (**fills** `tv` `Mytv`)) $\Longrightarrow$ `HTS`		AndL
8 `SS1` $\rightarrow$	`HTS`	Trans 5,7
9 (**all** `price`	(**min** 6000)) $\equiv$ (**and** (**all** `price` (**min** 6000)))	AndEq
10 (**all** `price`	(**min** 6000)) $\Longrightarrow$ (**all** `price` (**min** 6000))	Ref
11 (**all** `price`	(**min** 6000)) $\Longrightarrow$ (**and** (**all** `price` (**min** 6000)))	Eq 9,10
12 `SS1` $\rightarrow$	(**and** (**all** `price` (**min** 6000)))	Trans 4,11
13 `SS1` $\rightarrow$	(**and** `HTS` (**all** `price` (**min** 6000)))	AndR 8,12
14 `HHTS` $\equiv$	(**and** `HTS` (**all** `price` (**min** 6000)))	Told
15 `SS1` $\rightarrow$	`HHTS`	Eq 13,14
16 `HHTS` $\approx>$	(**all** `tv` (**all** `diagonal` (**min** 27)))	Told
17 `SS1` $\rightarrow$	(**all** `tv` (**all** `diagonal` (**min** 27)))	Rule 15, 16
18 (**fills** `tv` `Mytv`) $\Longrightarrow$ (**fills** `tv` `Mytv`)		Ref
19 (**and** `HTS` (**fills** `tv` `Mytv`)) $\Longrightarrow$ (**fills** `tv` `Mytv`)		AndL 18
20 `SS1` $\rightarrow$	(**fills** `tv` `Mytv`)	Trans 5, 19
21 `Mytv` $\rightarrow$	(**all** `diagonal` (**min** 27))	Prop 20, 17

Just as in the case of concepts, this line numbered proof is cumbersome and atomic descriptions and atomic justifications can be used to simplify matters. There are a few additional simplifications that we find useful for individual handling. It is worth noting steps 5-8 and 18-20 are all included in the proof just to be able to state that an individual is an instance of one of the conjuncts of its defining expression. We simplify matters by introducing the rule below which will collapse this deduction to one step:

$$\textsf{ToldInd} \quad \frac{}{\vdash \iota \longrightarrow \delta_i} \quad \text{ι is asserted (not derived) to be an instance of } (\textbf{and } \delta_1 \ldots \delta_i \ldots \delta_n)$$

Then, lines 5-7 and 18-19 can be eliminated, and lines 8 and 20 can be justified by ToldInd. In CLASSIC, this inference and the analogous inference for concept reasoning are both reported by the same inference rule name – Told.

Additionally, just as in the case of concept explanations, we choose not to report transitivity unless the intermediate concept is a named concept (in which case this inference is named Inheritance). So, for example, if we needed a justification of why `SS1` $\rightarrow$ (**at-least** 1 tv), this would be reported as Inheritance from `HTS` (i.e., δ =`HTS`). None of the applications of transitivity in the previous proof will be reported since none of them use an intermediate named concept.

Now, using atomic descriptions, atomic justifications, and our simplifications, an explanation of why

Mytv $\rightarrow$ (**all** `diagonal` (**min** 27)) is because

Propagation(ι=`SS1`, π=`tv`,

η= (**all** `diagonal` (**min** 27)), κ=`Mytv`)[1].

4.3.1 Individual Follow-Up Questions

Since new inferences have been added to the reasoning system, a follow-up question for these inferences need to be considered. We will address follow-up questions in the context of the previous example.

[1] Of course, η and κ are part of the question that we are asking, so we could choose only to report the bindings for ι and π.

The one-step explanation used the propagation rule. The appropriate follow-up questions to an explanation of propagation are:

- How did the individual, ι, obtain its restriction on the role π? In this case, how did `SS1` obtain the restriction on `tv` of (**all** `diagonal` (**min** 27))?
- How did the individual ι get κ as a filler for π? In this case, how did `SS1` obtain `Mytv` as a filler of the `tv` role?

Both are simply normalization questions on the individual ι. In this case,

`SS1` $\rightarrow$ (**fills** `tv` `Mytv`)

because ToldInd

so there will be no follow-up to this question. Additionally,

`SS1` $\rightarrow$ (**all** `tv` (**all** `diagonal` (**min** 27)))

because TriggerRule(δ=`HHTS`, identifier=HIGH-TV-RULE,

λ=(**all** `tv` (**all** `diagonal` (**min** 27))), ι=`SS1`)[2].

Looking at the form of the trigger inference rule, the appropriate follow-up questions is:

Why is ι an instance of the antecedent of the rule, δ? In this case, why is `SS1` $\rightarrow$ `HHTS`? This is just a recognition question, so a complete explanation mechanism can automatically ask the follow-up recognition question any time a rule fires. In practice, we found that users would prefer to do their own follow-up when user-defined rules fired, so we provided a way for them to choose if they would like rule firings to be automatically explained. If they chose the automatic follow-up, the next question would by why `SS1` $\rightarrow$ `HHTS`. In this case `SS1` has been told to be an `HTS` so the only remaining issue is why `SS1` $\rightarrow$ (**all** `price` (**min** 6000)), which is because MinOrdering(n=6000,m=8000). Users typically did *not* automatically follow-up rule firings.

To recap, our previous 21 line proof can be explained by one application of the

[2]Once again, ι, and λ are in the question, so their bindings may not be reported.

propagation rule, succeeded by two follow-up questions — one explained by the HIGH-TV-RULE firing and the other explained by told information. The HIGH-TV-RULE firing would have an optional follow-up which is justified by one application of the MinOrdering inference rule.

The example showed the appropriate follow-up questions for all the new individual processing rules except for the rule concerning closure of roles. For this, the follow-up questions are how did the individual in question obtain all of its role fillers for the role mentioned in the **at-most** restriction. So, for example, if a DL is told: `I` $\rightarrow$ (**fills** `r A B`) and the system has been told that the role `r` is closed on `I`, then `I` $\rightarrow$ (**at-most** 2 `r`) because ToldClosed(π=`r`, $\tilde{\kappa}$=\{ `A, B`\}). The follow-up questions are why does `I` have `A` and `B` as fillers for `r` (and the answer is because the information has been told to the system).

4.3.2 Discussion

There are a few issues that are worth noting in regards to individuals. First, it is typical for information about one individual to impact information about another individual. The most prominent example of this is the propagation rule. Thus, follow-up questions that began centering on one object, such as `Mytv` in our example, will lead to questions about properties of another object, such as the restriction on the `tv` role on `SS1`. It would be possible for a follow-up question concerning `SS1` to then come back and ask a question about some other individual, or perhaps even the original individual.

Consider the following pathological example (using the primitive concept `C`):

`I` $\rightarrow$ (**fills** `r J`)

`J` $\rightarrow$ (**and** (**all** `s` (**all** `r C`)) (**fills** `s I`))

After normalization, `I` and `J` look as follows:

---

individual: `I`

---

restrictions
 on roles: `r`
  fillers: `J`
  value
   restriction: `C`

---

individual: `J`

---

primitive
 superconcepts: `C`
restrictions
 on roles: `s`
  fillers: `I`
  restrictions
   on roles: `r`
    value
     restriction: `C`

If one asks why J $\rightarrow$ C, it is because of a propagation from I on the role r. The follow-up question is how did I obtain its value restriction on r, and that is because of a propagation from J on the role s with the information propagated: (**all** r C). Finally, asking how J obtained its restrictions on roles on s, there is no follow-up, since it is because of told information.

In this example, the follow-up questions move back and forth between objects but the actual questions do not cycle. The questions are why J $\rightarrow$ `C`?

why I $\rightarrow$ (**all** `r` `C`)?

why J $\rightarrow$ (**all** s (**all** `r` `C`))?

In cases involving multiple objects, there are two presentation styles which might be used to provide explanations. One would follow each chain of reasoning, asking follow-up questions and moving between individuals, another sets up all the explanations that are necessary and then presents all the information about each individual that is pertinent to the explanation. As mentioned in Chapter 3, we chose the second approach. We found this particularly useful when individuals refer to other individuals multiple times. There are other motivations for this as well which will become evident after the next example.

With rules and individuals, it becomes possible for the identical question to be posed multiple times in one series of follow-up questions. Consider a knowledge base below which will have a rule whose consequent contains a concept that is identical (or subsumed by) the concept to which the rule is attached.

This example begins with the primitive concept D. Attach a rule to D that says that anything that is a D has all of its r fillers being Ds. Then create an individual that is an instance of D and has its r role filled with itself.

Rs-are-Ds: `D` $\approx>$ (**all** `r` `D`)

`I1` $\rightarrow$ (**and** D (**fills** `r` `I1`))

After normalization and rule firing, `I1` looks as follows:

---

individual: `I1`

---

primitive
 superconcepts: `D`

restrictions
 on roles: `r`

fillers:	`I1`
value restrictions:	`D`

Now, we may consider asking why `I1` has the primitive `D` or

Question1: why `I1` $\rightarrow$ `D`?

The answer is twofold:

because Told *and* because Propagation from `I1` through role `r`.

There is no follow-up to the Told answer. The follow-up question to the propagation answer is why does `I1` have itself as the filler of the role `r`, i.e.,

Question2: why `I1` $\rightarrow$ (**fills** `r I1`)?

and why does `I1` have its value restriction on the `r` role, i.e.,

Question3: why `I1` $\rightarrow$ (**all** `r D`)?

The answer to Question2 is because of Told information so there is no follow-up question. The answer to Question3 is because the Trigger Rule Rs-are-Ds fired, and its follow-up is asking why the individual is an instance of the antecedent of the rule, i.e.,

Question4: why `I1` $\rightarrow$ `D`?

Question4 is identical to Question1 so there is a cycle in the follow-up question series which could cause an infinite process of question generation and answering if an explanation designer is not careful.

Our two pass approach provides a solution to the problem of looping follow-up questions. If some piece of information has already been requested in a series of follow-up questions, then an explanation structure will exist for it. A new follow-up question will only be generated if there is no existing structure for the appropriate piece of information. Thus, cycles will be broken. In this example, Question 4 will not be posed to the explanation system since an explanation structure already exists explaining this information. Additionally, implementors may choose, as we did in the CLASSIC implementation, not to follow-up alternative reasons for something if one of the reasons was Told information.

4.4 Recognizing and Explaining Errors

Properties of individuals change in the real world, and similarly, most DLs allow changes to individuals (though possibly not to concepts, which may be considered "eternal definitions"). Updating an individual, might (monotonically) add consistent information to this individual, it might (monotonically) add consistent information to another individual, or it might create an inconsistency in the local individual, or a remote individual, by over-constraining it. Although our definition of the concept `NOTHING` states that it has no instances, one could conceptualize inconsistent individuals as being instances of `NOTHING`. Thus detecting an inconsistency could be thought of as just another case of recognizing an instance relationship, specifically looking for instances of `NOTHING`.

A simple example of an error begins with a knowledge base containing:

`I` $\rightarrow$ (**at-most** 2 `r`);

updating `I` such that

`I` $\rightarrow$ (fills `r A B C`)

generates a bounds conflict since `r` would need to have at least 3 and at most 2 fillers. In most knowledge-base management systems, such an inconsistency will cause the DL to reject the most recent update, and revert to the previous stable state where `I` has at most 2 `r`-fillers. The filler information has now been lost. But experience shows that it is very important to explain to users what the inconsistency was, and to do an adequate job of reporting the error, one needs to access objects in their *intermediate error state*. CLASSIC stores such intermediate error objects.

4.4.1 Intermediate Objects

Intermediate objects record the state objects were in when a contradiction was detected. If a system does *not* draw any additional conclusions from the presence of inconsistency (e.g., as in various forms of relevance logic), we could let the system complete its deductions, and then temporarily store and explore the complete state of the (inconsistent) knowledge base.

Reasoning to completion may, however, require numerous additional deductions, thus implemented DLs concerned with efficiency may try to avoid the extra computation. Additionally, it is easier for a user if an intermediate state of an object only contains information that resulted from the processing involved in detecting only *one* contradiction. It is possible for one description to lead to more than one contradiction and if a DL was allowed to process to completion, then the intermediate state of objects would contain information concerning all the contradictions, causing potential confusion. CLASSIC, for example, after detecting an initial contradiction, halts all additional deduction, and reverts to a previous stable state. Intermediate objects in CLASSIC are used to record the state that the object was in when a contradiction was first detected.

For example, there will be an intermediate object associated with `I` after adding 3 fillers. Someone who is debugging the knowledge base may choose to print `I` in its stable state:

---

individual: `I`

---

restrictions

on roles: `r`

at-most: 2

It may be more informative to print `I` in its intermediate state:

---

individual: `I-*intermediate*`

---

restrictions

on roles: `r`

at-least: 3

at-most: 2

fillers: A, B, C

It is appropriate for explanation functions to work on intermediate objects as well as regular objects, so that one can determine, for example, where the **at-least** 3 restriction came from. If one asked for an explanation of why I $\rightarrow$ (**at-least** 3 r), there would be no explanation since the stable state of the knowledge base does not have that information. If instead, one asks why `I-*intermediate*` $\rightarrow$ (**at-least** 3 r), we would find that it is because of the FillersImpliesAtLeast inference.

Consider another example where one object impacts another. Start with

I1 $\rightarrow$ (**all** r (**at-most** 2 r))

B $\rightarrow$ (**at-least** 3 r)

which are unrelated and consistent. Consider however what happens when I1 is further augmented with the information

I1 $\rightarrow$ (**fills** r B).

This does not cause any contradiction on I1, but a propagation onto B will cause the DL to detect a contradiction due to a bounds conflict on B; therefore the DL will disallow the update. Before or after the update, the stable state of I1 and B are:

---

individual: I1

---

restrictions
on roles: r
restrictions
on roles: r
at-most: 2

---

individual: `B`

---

restrictions

on roles: `r`

at-least: 3

There would be no bounds conflict to explain on `B`. However, `B-*intermediate*`, looks as follows:

---

individual: `B-*intermediate*`

---

restrictions

on roles: `r`

at-least: 3

at-most: 2

Thus, an explanation of interest might be to ask why `B-*intermediate*` $\rightarrow$ (**at-most** 2 `r`)? The answer will not refer to `I1` since that does not have either `B` or `B-*intermediate*` as a filler of any role and, thus, no propagations will occur using it. Instead, the answer will need to refer to the intermediate state of `I1`, after the addition of `B` as a filler of `r` but before the DL has reverted to a stable state. Thus, we also need an intermediate object for `I1`, which looks as follows:

---

individual: `I1-*intermediate*`

---

restrictions
 on roles: `r`
  fillers: `B-*intermediate*`
  restrictions
   on roles: `r`
    at-most: 2

Now, one can discover that `B-*intermediate*` $\rightarrow$ (**at-most** `2 r`) because of a propagation from `I1-*intermediate*` on role `r`. This example shows that a DL requires access to the intermediate state of more than just the object on which the error was discovered. In fact, the DL potentially requires access to the intermediate state of all objects that were touched in an update in order to be prepared to give adequate information for printing and explaining error states. The DL does not need these objects to be in their completely normalized form however. The only thing that is needed is for the DL to have updated the intermediate objects with the information that was used in a contradiction detection. So for example, if it deduced that there was a bounds conflict on some object, the bound and the information that caused the bound to be deduced must be reflected in the intermediate object, but information not having to do with this deduction need not be completely normalized in order to give an informative error explanation just concerning the contradiction.

4.4.2 Automatic Error Follow-up Questions

Intermediate objects have proven to be invaluable in diagnosing error situations.

However they do add some complexity to any explanation or printing interface since the user may now ask questions about the intermediate state of the object as well as its stable state. After interviewing subjects who used explanations to debug knowledge bases, we concluded that automatic support for error question generation would be valuable. Someone familiar with the possible error conditions in a DL can easily determine the appropriate follow-up questions that should be asked for each error. For example, in the case of mini-CLASSIC, there is only one kind of (semantic) error – InconsistentBoundsConflict. In this case, the appropriate follow-up questions are:

How did the object obtain its **at-least** restriction?

How did the object obtain its **at-most** restriction?

In our implementation, we have provided a special function that takes the intermediate state of an object, identifies which error inference was used, and then asks the appropriate follow-up questions for that inference, thus setting up explanation structures on this object or on other (potentially intermediate) objects that contributed to the deduction.

For example, a special function, (called cl-exp-error in CLASSIC), could be used to explain `B-*intermediate*` above. It would identify the bounds conflict on role `r`, and then explain where the **at-least** and **at-most** restrictions came from. Conceptually, one could think of this as having the system determine what questions make sense to ask about an object, and then automatically answering those questions. In this simple example, where `B-*intermediate*` only contains information that contributed to the error, it is not quite as critical to choose a small number of things to explain about the object. However, in real knowledge bases, intermediate objects may contain a large amount of data, so simply explaining where every piece of information came from can be overwhelming. Choosing the "important" things to explain becomes critical in such cases.

We should note that error handling is also useful on concepts, however it is much more prevalent in individual processing. Intermediate objects would become more important to an implementation of concepts if concepts are allowed to be modified or if concept reasoning is complete.

4.5 Summary

In this chapter, we have shown how the foundation presented in Chapter 3 can be extended to explain individual processing. The additions required involved a small number of additional inferences concerning propagations, closed world reasoning, trigger rule firing, and role closure. We showed how these inferences can be used to explain typical patterns of individual reasoning. We presented a discussion of follow-up questions, noting some potential problems and showed how our two phase approach avoids the problems. Finally, we addressed the area of changing knowledge bases and explaining error situations and showed how our approach naturally handles such explanations with the aid of intermediate objects.

Chapter 5

Non-Subsumption

While explaining positive determinations of subsumption relationships is important, it can be equally important to explain negative determinations of subsumption. This chapter explores explanations of why one concept, D, has *not* been found to be subsumed by another concept, E. One unsatisfying explanation of non-subsumption is that a "black box" algorithm, which is believed to be correct and complete, tried everything that it was supposed to try but failed to prove subsumption. Even if this was extended to include a trace of what was tried and why it failed, these traces may include many deduction paths and quickly become unwieldy. In addition, if the implementation is incomplete, then subsumption relationships could be missed, thus non-subsumption determinations based on such algorithms could be unsound.

We consider a different strategy for explaining non-subsumption, based on counter-examples inspired by the tableau techniques described in Section 2.7.2. This kind of explanation says that D is not subsumed by E because it is possible for a counter-example, I, to exist such that I is an instance of D but I is not an instance of E. If such a counter-example exists, being an instance of D does not imply being an instance of E and thus, subsumption does not hold.

In order for this approach to succeed, it must be possible to:

(1) generate a counter-example individual I

(2) explain why I $\rightarrow$ D

(3) explain why I $\not\rightarrow$ E.

In this chapter, we will first motivate the problem of non-subsumption explanation with an example. Then we will present our approach, first by showing that (2) and (3) are simple problems in a closed-world reasoning domain, then by presenting a solution to (1) that produces a "closed" counter-example individual. We will do this first by example, and then by giving an algorithm. We have not, however, implemented this algorithm and integrated it into CLASSIC. We will also discuss the algorithm's use for dealing with the incompleteness of the subsumption reasoner, and an example trace of its use.

5.1 Motivation and Approach

During certain activities such as knowledge base creation and browsing, it becomes important to be able to explain why one object is *not* subsumed by another. For example, a knowledge engineer might define a new concept `NEW-WINE` (`NW`) as:

`NW` = (**and** (**prim** `WINE`)
(**all** `location` `CALIFORNIA-REGION`)
(**fills** `grape` `Merlot`)).

We might also remember the definition for `AMERICAN-BLENDED-WINE (ABW)`:

`ABW` $\equiv$ (**and** (**prim** `WINE`)
(**all** `location` `AMERICAN-REGION`)
(**at-least** 2 `grape`)).

In our previous pictorial form[1] `NW` and `ABW` look as follows:

---

concept: `NEW-WINE (NW)`

---

primitive
superconcepts: `WINE`
restrictions
on roles: `location`

[1]For simplicity, these depictions do not include derived information that is equivalent to `THING`, i.e., restrictions such as an **at-least** restriction of 0 are not included.

	value
	restriction: `CALIFORNIA-REGION`
	`grape`
	at-least: 1
	fillers: `Merlot`

---

concept:	`AMERICAN-BLENDED-WINE (ABW)`
primitive superconcepts:	`WINE`
restrictions on roles:	`location`
	value restriction: `AMERICAN-REGION`
	`grape`
	at-least: 2

A DL will *not* deduce that `NW` is subsumed by `ABW`. Thus, a knowledge base designer might need to ask why. One justification for the non-subsumption is the existence of an individual that is an instance of `NW` but is not an instance of `ABW`. Consider the following individual, `I1`, which is told to be an instance of:

(**and** (**prim** `WINE`)
    (**at-most** 0 `location`)
    (**fills** `grape Merlot`)
    (**at-most** 1 `grape`))

Using a new pictorial description for individuals that just includes their primitive superconcepts and role filler information, I1 looks as follows:

individual:	I1	
primitive superconcepts:	WINE	
closed role fillers:		
	location:	$\emptyset$
	grape:	Merlot

It is a simple matter to say why I1 $\rightarrow$ NW: I1 is an instance of all of the primitives of NW (since it is an instance of WINE), I1 can never have any fillers for its location role, thus all of those (nonexistent) fillers are instances of any concept, including CALIFORNIA-REGION, and Merlot is a filler of I1's grape role. It is equally simple to show why I1 $\not\rightarrow$ ABW: ABW requires 2 fillers for the grape role and I1 has exactly one filler.

In this case, we reduced explaining why NW $\Longrightarrow$ ABW to producing a counter-example I1, and explaining why I1 $\rightarrow$ NW and I1 $\not\rightarrow$ ABW. The two recognition explanations were very simple in our example. In general, however, recognition can include subsumption reasoning (as shown in Chapter 4 when individuals are not completely specified) and hence non-recognition might involve non-subsumption, so we would be caught in a circular trap. If, however, we appropriately restrict the form of the counter-example, it is possible to guarantee that recognition will not require subsumption reasoning, and in fact, be straight forward. The next two paragraphs will motivate this restriction and demonstrate it.

In predicate calculus, given a finite model, it is much easier to explain why a formula is true or false than it is to explain why the formula is valid or invalid. This is because checking for truth or falsehood only involves calculating a truth value by using the definition of the logical connectives and the "knowledge base" corresponding to the model, from which truth values of predicates can be obtained. Similarly, determining that an individual is an instance of a concept can be much easier than proving why its defining expression is a subconcept of another expression. The DL knowledge base must contain the analog of the predicate interpretations for DLs. There are two kinds of information required about each individual – membership in primitive concepts and role fillers. We will call a knowledge base closed if for every individual `I`:

- for every primitive concept `P` in the knowledge base, it is known whether or not `I` is an instance of `P`.
- for every role `r` in the knowledge base, `r` is closed on `I`. (Thus, all fillers of `r` are known for `I`.)

Given such a closed knowledge base, KB, it is a simple matter to determine if an individual is an instance of a description – one just uses the definition of each concept constructor. For example:

An individual `I` is an instance of (**at-least** n r), if the number of r-fillers of `I` is $\geq$ n.

An individual `I` is an instance of (**prim** C_1 ... C_i ... C_n), if `I` is an instance of *every* C_i.

An individual `I` is an instance of (**and** C_1 ... C_i ... C_n) if `I` can be shown, recursively, to be an instance of *every* C_i.

An individual `I` is an instance of (**all** `r C`) if *every* r-filler of `I` is an instance of `C`. A complete set of these simple "lookup" rules for CLASSIC appears in Figure 2.1. These are the *structural recognition rules* for closed world reasoning and these are assumed to be obvious to the user,[2] from the meaning of the concept constructors. Note that none

[2]If we needed to include explanations for these rules in our implementation, it would be a simple matter to add them.

of these rules uses subsumption, and the rules only require information on primitives and/or role fillers. If an individual `I` is recognized to be an instance of a concept `C` using these rules, we will write that `I` $-s \rightarrow$ `C`.

The only recursive rule for CLASSIC are associated with the **all** and **and** constructors, and those rules make a call to the same procedure with a simpler description. For example, to determine if `I` $\rightarrow$ (**all** r (**at-least** 2 s)), there needs to be a determination if all of `I`'s `r`-fillers are instances of (**at-least** 2 s). This process terminates.

Just as significantly, the rules for determining *non-recognition* given a closed knowledge base are equally simple. For example the rules governing the constructors used in the previous paragraph are shown below:

An individual `I` is *not* an instance of (**at-least** n r), if the number of r-fillers of `I` is $<$ n.

An individual `I` is *not* an instance of (**prim** C_1 ... C_i ... C_n), if `I` is *not* an instance of *some* C_i.

An individual `I` is *not* an instance of (**and** C_1 ... C_i ... C_n) if `I` is *not* an instance of *some* C_i.

An individual `I` is *not* an instance of (**all** `r` `C`) if *some* r-filler of `I` is *not* an instance of `C`.

If we add the rules below, we have the complete set of rules needed for CLASSIC[3] .

An individual `I` is *not* an instance of (**at-most** n r), if the number of r-fillers of `I` is $>$ n.

An individual `I` is *not* an instance of (**one-of** I_1 ... I_j ... I_n) if `I` $\neq I_j$ for *every* j $\leq$n.

An individual `I` is *not* an instance of (**test** T_1 ... T_j ... T_n) if T_j(`I`) $\neq$ true for *some* j $\leq$ n.

An individual `I` is *not* an instance of (**fills** `r` I_1 ... I_j ... I_n) if *some* I_j is not an r-filler of `i`.

[3]In this presentation, we will not include the **same-as** constructor. We discuss the necessary extension for handling **same-as** in Section 5.7.

An individual `I` is *not* an instance of (**min** n) if `I` $<$ n.

An individual `I` is *not* an instance of (**max** n) if `I` $>$ n.

If an individual, `I`, is recognized *not* to be an instance of a concept, `C`, using these rules, we will write `I` $-s \not\rightarrow$ `C`. Given a closed knowledge base, it is simple to determine recognition and non-recognition of an individual and a concept and the correctness and completeness of the algorithms follows immediately from the *denotational semantics* of the DL.

Note that given some individual `I`, we only need to have "complete" information about those individuals that play some part in determining (non)recognition. In the case of CLASSIC, these are only the individuals reachable from `I` via chains of roles. Moreover, given a concept C, with respect to which we are trying to determine (non)membership, the chains of roles of interest are also restricted to be those appearing in the concept definition.

Observe also that if the DL language has a concept constructor **not**, corresponding to negation, then showing that `D` is not subsumed by `E` is equivalent to finding an example instance of the concept (**and** `D` (**not** `E`)), so only complete example instance generation is needed and non-membership explanation is not even required.

5.2 Counter-Example Generation

We will consider the issues involved with generating an individual `I1`, that is closed and is a counter-example to the alleged subsumption `D` $\Longrightarrow$ `E`. Rather than writing a separate algorithm for generating this individual, our strategy will be to take advantage of the fact that we have an implemented performance system which already does individual reasoning (though possibly not all such reasoning, due to incompleteness).

In the example in the previous section, one simple method of generating a counter-example was to:

(1) tell the DL system that `I1` was an instance of the alleged subsumed concept, i.e., `I1` $\rightarrow$ `NW`.

(2) close all the roles mentioned in `NW` on `I1`, and repeat the role closing recursively for fillers of these roles on `I1`.

From (1), the DL reasoner would conclude `I1` belongs to the primitive concept `WINE`, and has filler `Merlot` for role `grape`. Closing `location` and `grape` on `I1` (and possibly `Merlot`, for safety), yields a closed instance of `NW`, which turns out not to be an instance of `ABW`. Thus, `I1` is a counter-example to the subsumption statement.

This simple procedure may not work in general for one of two reasons: (i) closing the roles may not necessarily lead to an instance that is recognized according to the rules of structural recognition (e.g., because there may be a problem in closing the roles), and (ii) the resulting individual might not be counter-example because it is still a member of the alleged subsumer.

Consider the following example using:

`NAPA-WINE` (`NAW`) $\equiv$ (**and** (**prim** `WINE`)
(**fills** `location Napa`)
(**at-least** 1 `grape`)
(**all** `grape GRAPE`))

and `ABW` defined in the previous section. Classic will *not* deduce that `NAW` $\Longrightarrow$ `ABW`, and a counter-example does exist. However, if one asserts `I1` $\rightarrow$ `NAW` and then attempts to close the `grape` and `location` roles, an inconsistency will be detected in the knowledge base, since `I1` now must have **at-least** 1 and **at-most** 0 `grape`-fillers. However, by first adding a new, anonymous filler for the `grape` role, we can then close the roles and get the desired instance. The final individual is shown below:

---

individual:	`I1`
primitive superconcepts:	`WINE`
closed role fillers:	

location: Napa

grape: Anon-grape

It is important to note that in this example, the existing DL implementation detected the conflict for us. Also, upon adding a new filler for the grape role, the DL implementation did some additional inferences, such as propagating the information that this new individual, Anon-grape, must be an instance of the concept GRAPE. Therefore, in this example we were left only with the task of "gensym"ing the new individual filler for the appropriate role.

The next example shows that a successful closure of an individual is not enough to guarantee a counter-example. Consider the following definitions for SINGLE-VARIETAL-WINE (SVW) and BAD-MERLOT (BM).

SVW ≡ (**and** (**prim** WINE) (**at-least** 1 grape) (**at-most** 1 grape))

BM ≡ (**and** (**prim** WINE) (**fills** grape Merlot))

Given these definitions, CLASSIC will **not** deduce that BAD-MERLOT $\Longrightarrow$ SINGLE-VARIETAL-WINE because the **at-most** restriction on SINGLE-VARIETAL-WINE is not met. When we assert I2 $\rightarrow$ BM, and attempt to close all roles, we obtain a valid individual which is an instance of the alleged subsumee concept; however it is also an instance of the alleged subsumer concept, and is thus not a counter-example. In such a situation we need to try to modify the individual. We can add another filler to the grape role on I2 before closing it and obtain a counter-example.

---

individual: I2

---

primitive

superconcepts: WINE

closed role

fillers:

	`grape:`	`Merlot`
		`Something-or-other`

`I2` is a counter-example to the statement `BM` $\implies$ `SVW`.

In order to generate `I2`, our strategy could be stated as: create an individual that is an instance of the alleged subsumee concept. If closing that individual results in an instance of the alleged subsumer concept, appropriately modify the individual so that it is not an instance of the alleged subsumer concept. In order to obtain guidance on how to modify the individual, we can rely on our explanation work. In particular, if we know which atomic description of the alleged subsumer is not implied by the alleged subsumee, that tells us where to aim our modifications. In this case, (**at-most** 1 `grape`) is not implied by `BM`, thus we know to try to create an individual that has more than one `grape`-filler. So our resulting strategy for explaining why $D \implies E$ was not deduced has the following components:

- determine an atomic description, A, such that E $\implies$ A, but D $\not\Longrightarrow$ A.
- create an individual I such that I $\rightarrow$ D, deducing filler information if necessary to satisfy role restrictions.
- appropriately modify I so that I $\not\rightarrow$ A, while maintaining I $\rightarrow$ D.

Since there may be several such atomic descriptions, and since some of them may actually be incorrect hints (since the subsumption algorithm may be incomplete, see Section 5.5), we have chosen to present the more general algorithm (which may however do more work). We will present the pseudo-code for our approach below, in the `GenerateCounterExample` function.

GenerateCounterExample(D,E) {

for I in GetInd(D,E) {

I = Contradict(I,D,E)

if success

return I as the counter-example to the subsumption}

Return "No counter-example found." }

`GetInd(D,E)` returns "prototypical" instances[4] I of D, while `Contradict(I,D,E)` tries to modify I to be a non-instance of E, keeping it an instance of D.

In the following two subsections, we will discuss the two main functions in this algorithm — `GetInd` and `Contradict`.

5.3 Deriving an Example Individual in CLASSIC

Let us consider the problem of creating a closed instance of some concept D — the task of procedure `GetInd`. The general difficulty will be that the description D is likely to be indeterminate in some manner, and we will have to make some choices regarding this indeterminacy in order to get a closed individual– specifically, in order to obtain an individual with all of its filler information known. At this stage, a second problem may arise: namely that some choices would be inconsistent, so that the entire process may be more easily thought of as a non-deterministic one. The tableaux proof techniques used in the implementation of DLs such as KRIS attempt to construct exactly such an example individual (hoping to find failure in all paths, and thereby demonstrate subsumption).

We could have tried to find a complet tableaux technique for CLASSIC along the lines of [30], but chose to proceed differently because (i) we will want the already implemented DL individual reasoner to help us as much as possible in obtaining the example (e.g., detecting inconsistencies, propagating information) rather than messing around with the implementation of CLASSIC; and (ii) we desire a deterministic algorithm.

The nature of the problems that may arise when we try to simply close the roles, and the nature of the repairs needed are unfortunately specific to every DL, and its constructors. We will therefore restrict ourselves henceforth to the issues arising in the

[4]If D has certain forms of disjunction, more than one individual may be generated, in order to simulate case-by-case reasoning.

task of generating examples for CLASSIC concepts.[5]

In CLASSIC, trying to close an individual I, after successfully asserting that it is an instance of concept D can lead to only one problem:

- Some role(s), on some individual(s) may not have enough fillers to meet an **at-least** restriction.

The reason for this is that consistency checking and propagation reasoning in the CLASSIC system have already ensured that any individuals present satisfy the restrictions of the other constructors. Therefore, the task of creating a closed individual instance of some concept is just the task of adding extra fillers whenever lower bounds on roles are not met. However, in order to be able to create a counter-example, in the next subsection, we want to add only the minimal possible information in every case. In most situations, this means adding some newly created "anonymous" individual as a filler, so it will not possess any unnecessary properties. But this individual is in fact only a Skolem constant, and in some situations we may be forced to replace it by some existing individual. We will give an example of such a situation in our discussion of the contradict algorithm.

One major obstacle in the path of a deterministic algorithm is the **one-of** constructor. A description such as

(**all** r (**one-of** I_1 I_2 I_3))

does not permit the generation of anonymous role fillers, and a particular individual may have any subset of these individuals as fillers for role r. Each such alternative can be captured by a description:

(**at-most** 0 r)

(**and** (**at-most** 1 r) (**fills** r I_1))

(**and** (**at-most** 1 r) (**fills** r I_2))

(**and** (**at-most** 1 r) (**fills** r I_3))

(**and** (**at-most** 2 r) (**fills** r I_1 I_2))

(**and** (**at-most** 2 r) (**fills** r I_1 I_3))

---

[5]For simplicity of presentation, we also do not discuss CLASSIC user-defined rules.

(**and** (**at-most** 2 r) (**fills** r I_2 I_3))

(**and** (**at-most** 3 r) (**fills** r I_1 I_2 I_3))

In order to simplify our algorithms, we will take the drastic step of preprocessing some descriptions, replacing **one-of**s by a series of alternate description for each of these choices. As a result, the remainder of the processing can proceed deterministically, and will not need to consider the **one-of** constructor. In our algorithm presentation, `RemoveEmbeddedOneofs` handles this pre-processing.

The remainder of the algorithm will then treat each of these descriptions separately. Of course, this will create undesirable worst case performance since expanding the **one-of**s will produce a combinatorial explosion of descriptions to process. It is encouraging to note that in our prototype and industrial knowledge bases, it was extremely rare to find **one-of**s with more than three elements.

A more minor inconvenience is caused by a "top-level" **one-of**. In the absence of such a construct, the algorithm can generate a new individual to instantiate the concept D; but if D contains a top level (**one-of** { Iset }), then the algorithm must choose an individual from Iset. In such cases, the algorithm needs to try successively every element of Iset – which is the task of the subprocedure `BreakUpTopLevelOneof`.

This leads to the following pseudo-code for the `GetInd` procedure:

GetInd(D,E)

assumes: D is a told description

returns: a list of (individual I_i, description D_j) pairs

such that $I_i \rightarrow$ D, $D_j \Longrightarrow$ D, , {D_j} cover D and

ClosedForMemb(I_i,D) $-s\rightarrow$ D

(The list only contains more than one pair if D contains **one-of**s)

{ expand any defined concepts in D

Ilist0 = BreakUpTopLevelOneof(D)

Returnpairs = nil

For all D' in RemoveEmbeddedOneofs(D) {

```
    Normalize(D')                   ;; will fail on bad one-of breakups
    if fail or E subsumes D',
        continue loop               ;; skip D' if it can't generate a counter-example
    for all j in CreateInd(D',Ilist0) {
        Complete(j,D')
        if successful, push (j,D') on Returnpairs }}
return Returnpairs }
```

Auxiliary functions concerning **one-of** handling (`BreakUpTopLevelOneof` and `Remove-EmbeddedOneofs`) for `GetInd` appear in Figure 5.1. An additional auxiliary function that creates individuals appears in Figure 5.2. `ClosedForMemb` appears in Figure 5.3. For now, it can be thought of as an efficient way of closing the knowledge base with respect to D.

GetInd relies on a crucial function which "completes" the individual. The pseudocode for `Complete` appears in Figure 5.4.[6] The goal of `Complete` is to create an individual on which only structural recognition rules need to be used to identify if the individual is an instance of the given description. For some descriptions, `Complete` does not have any additional work to do. For example, if `D` $\equiv$ (**prim** `F`) and `I` $\rightarrow$ `D`, there is no additional information needed to use the **prim** rule for structural recognition to deduce that `I` $-s\rightarrow$ `D`. For `D`s of other forms, however, we may need to add fillers for roles which have an unmet **at-least** restriction, close roles for which the structural recognition rules need to know all fillers, or complete individuals that fill roles for which there is an **all** restriction that must be satisfied. For example if an individual has no known fillers for r but is known to be an instance of (**at-least** 1 r), `Complete` must generate a filler for `r`. Additionally, if an individual is asserted to be an instance of (**and** (**all** `r F`) (**fills** `r Ind3`)), then `Complete` needs to assure that `r` is closed on the individual. Moreover, in order to guarantee that `Ind3` is an instance of `F`, `Ind3` must also be completed.

In order to be more efficient, we will introduce two auxiliary functions which know

[6] Lines marked with *** will be discussed in Section 5.7 when we mention additional requirements for **same-as** processing.

BreakUpTopLevelOneof(D)

assumes: D is a told description with any defined concepts expanded
returns: a list containing the common elements of any top level **one-of**s
(i.e. (and (**one-of** a b c) (**one-of** a c) (**at-least** 2 r)) will
return the list (a c)
{ return the intersection of the set arguments of all **one-of**s }

RemoveEmbeddedOneofs(D)

assumes: D is a told description with all defined concepts expanded
and no top level **one-of**s left
returns: a list of descriptions {D'} such that every D' $\Longrightarrow$ D
and no D' contains a **one-of**. The set contains a D'
for every possible **one-of** combination covered in D.
{ move any **and** to left so that no **and** is embedded in an **all** restriction
;; i.e., (**all** p (**and** C (**one-of** K L))) = (**and** (**all** p C) (**all** p (**one-of** K L)))
case on the form of D:
(**and** C_1 ... C_n):
E_1 = RemoveEmbeddedOneofs(C_1)
...
E_n = RemoveEmbeddedOneofs(C_n)
result = { (**and** $A_1 ... A_n$) | $A_i \in E_i$ }
(**all** p (**one-of** O_1 ... O_n)):
result = { (**and** (**fills** p {S}) (**at-most** k p)) |
S a subset of { O_1 ... O_n } of size k }
(**all** p X):
result = { (**all** p Y) | Y $\in$ RemoveEmbeddedOneofs(X) }
anything else:
result = {D}
return result }

Figure 5.1: Auxiliary Oneof Functions for Creating an Example Instance

CreateInd(D,Ilist)

```
assumes:   D is coherent according to CLASSIC
returns:   A list of consistent individuals that are instances of D.
           If Ilist is empty, returns a single newly created
           individual. If Ilist contains inds, then
           uses CLASSIC to add the description D to each member.
           returns the individuals on which this was successful.
note:      This only fails when expanding one-ofs created an incoherence
{ newIlist = nil
  if Ilist nonempty
        { for all element I of Ilist
              cl-ind-add(I,D)
              if successful, put I on newIlist
              ;; this could fail because of propagations
        if newIlist empty, then fail }
  else generate a new name - Anon-Ind
        { cl-create-ind(Anon-Ind,D) ;; this could fail because of propagations
        if fail, return fail
        else put Anon-Ind on newIlist }
  return newIlist }
```

Figure 5.2: Auxiliary Functions for Creating an Example Instance

how to limit the roles that need to be closed and the role fillers that must be completed. Those auxiliary functions, called `CloseForMemb` and `CloseforNonMemb` appear in Figure 5.3. `CloseForMemb` realizes that if one is interested in structural recognition, then many roles do not need to be closed in order to use the structural recognition rules. For example, if `I` $\rightarrow$ (**fills** `r Ind1 Ind2`) and `D` $\equiv$ (**at-least** 1 `r`), it is not necessary to close `r` on `I` before the structural recognition rule for **at-least** can be used. However, if `D` $\equiv$ (**at-most** 2 `r`), it is necessary to close `r` before the **at-most** structural recognition rule can be used. In addition, `GetInd` minimizes the amount of work it does, creating only one anonymous filler per incomplete role, counting on `CloseForMemb` to make sufficient copies of it, if needed, to satisfy any **at-least** restrictions. This will become important later, when working on the counter-example itself.

Discussion At this point, we will provide an informal argument of the correctness and termination of critical functions. (This discussion could be skipped on an initial reading.) Since `GetInd` relies critically on `Complete`, we will consider what it means for

CloseForMemb(ind,D)

```
assume:     ind → D
effect:     ind has roles closed which have restrictions
            also, anonymous inds are duplicated as needed
   { mark ind
        for every role r with an at-most in D
             close role r on ind;
        for every role r with an at-least in D
             duplicate the anon individual x in r on ind
                  sufficient times to meet at-least bound;     ****
             close role r
        for every role r on D with all-restriction, say C
             { close role r on ind
             for every unmarked filler x of r on ind
                  CloseForMemb(x,C) }
   }
```

CloseForNonmemb(ind,E)

```
assume:     ind is consistent
effect:     all roles on ind with at-least or fills information in E are closed
   { mark ind
   for every role r with an at-least or fills in E
        close role r on ind
   for every role r on E with all restriction, say C
        { for every unmarked filler x of r on ind
             CloseForNonmemb(x,C) }
   }
```

Figure 5.3: Auxiliary Closing Functions

```
Complete(Ind,D)

Assumes:   Ind → D, Ind is consistent,
           D does not contain one-of
Effect:    attempt to modify Ind so
           CloseForMemb(Ind,D) is consistent and Ind −s→ D;
           report if successful
{ case D= (prim Aset), (fills p Bset), (min n), (max n), or (at-most k p)
        if Ind → D, then return success else report fail
  case D = (at-least k p)
        If count(cl-fillers(Ind,p)) ≥ k, then return success
        else {
           generate anonymous ind x
           add x as p-filler of ind          ;; in classic, cl-ind-add(Ind,(fills p x))
           *****
           if adding x causes classic failure then return fail else return success }
  case D = (all p C)
        for every p-filler j on ind
           { Complete(j,C)
           if fail, return failure }
        return success
  case D = (and C1 C2)
        Complete(ind,C1);
        if fail, return failure
        Complete(ind,C2);
        if fail, return failure
        else return success }
```

Figure 5.4: `Complete` pseudo-code

`Complete` to return success. We will want to show that given the assumption `Ind` $\rightarrow$ `D`, if success is returned from `Complete` then, `CloseForMemb(Ind,D)` is consistent and is structurally an instance of `D`. We will proceed by induction on the number of **all** and **and** constructors in `D`, considering the cases according to the form of `D`.

If success is returned and `D` had the form (**prim** Aset), (**fills** `r Iset`), (**min** n), or (**max** m), then `Complete` had made no changes to the individual. Additionally, `CloseforMemb` does not do anything for these constructors, so `Ind` is unchanged. The assumption was that `Ind` was consistent and `Ind` $\rightarrow$ `D` and for these constructors, that is identical to `Ind` $-s \rightarrow$ `D`.

If success is returned and `D` had the form (**at-most** n `r`), `Complete` had made no changes to the individual. It is known that `Ind` $\rightarrow$ `D`, thus it can not be the case that `Ind` has too many `r`-fillers and thus `CloseForMemb` must be able to close `r`. Thus, `CloseForMemb(Ind, D)` $-s \rightarrow$ `D`.

If success is returned and `D` had the form (**at-least** n `r`), then one of two routes is followed: (1) `Complete` added no fillers because `Ind` already had enough fillers and then `CloseForMemb` did not change `Ind`. Thus, `Ind` is unchanged and is known to have at least n fillers so `Ind` $-s \rightarrow$ `D`. (2) `Complete` successfully added an anonymous individual as a filler of `r`. If this individual can be added once, the appropriate number of copies of it may be added as `r` fillers. (The only thing that would not allow copies to be added would be conflicting bounds but that is impossible since `Ind` $\rightarrow$ `D`.) `CloseForMemb` will make the appropriate number of copies and add them as `r` fillers so `Ind` now has at least n `r`-fillers and thus `CloseForMemb(Ind, D)` $-s \rightarrow$ `D`.

If success is returned and `D` had the form (**all** r `C`), then every `r`-filler of `Ind` returned success from `Complete(Ind, C)`. There is a recursive call to `Complete` (with an argument containing one less **all** than before), so by induction all of `Ind`'s `r`-fillers are structural instances of `C` and thus, `CloseForMemb(Ind, D)` $-s \rightarrow$ `D`.

The case of `D` having the form (**and** `C1 C2`) is of course handled similarly by induction.

Next we will consider what it would mean for `Complete` to fail, and in fact argue that if CLASSIC individual reasoning is complete in the absence of **one-of**, then Complete will never return fail!

If `D` is of the form (**prim** Aset), (**fills** r Iset), (**at-most** n r), (**min** n), or (**max** m) then by the assumption stated at the beginning of the procedure, Complete cannot fail. For `D` of the form (**at-least** n r), given the assumption that individual propagations are complete if **one-of** is removed from the language, then it is not possible for errors to occur when anonymous individuals are added as fillers of roles with unmet **at-least** restrictions since CLASSIC has done a complete job of inference by way of propagations. In other words, CLASSIC had anticipated the introduction of existentially quantified role fillers. By induction, `D` of the form (**all** r `C`) or (**and** `C1 C2`) may not fail either. (Failures for bad **one-of** choices would already have been found by `cl-create-ind`.)

Next, we will consider termination. The only place where `Complete` is called recursively is in the case where `D` is an **all** restriction. The next call to `Complete` has a smaller description argument– i.e., if the first call was on arguments (`Ind`, (**all** r `C`)), the next call will be on (`Ind1`, `C`). Thus, the description will eventually contain no more nested **all**s, and the recursion will stop. Also, `Complete` uses `CloseForMemb` which is called recursively for individuals on role-paths on the original individual. We should note that `CloseForMemb` creates no new individuals and it is successively marking (and closing) more individuals so eventually it will not find any new individuals which must be closed. (In the worst case, it would close all the individuals in the knowledge base which contains a finite number of individuals.)

Note that the issue of propagation has been raised in our discussion. In our effort to exploit the underlying implementation, we are using the creation command (cl-create-ind in CLASSIC) to generate new individuals and we are using the modification command (cl-ind-add in CLASSIC) to add information to individuals. This means that the propagations are done for us by the underlying description logic. Because the implementation is complete with respect to propagations after **one-of** is removed from the language, individual creation does not need to address the additional issues raised in the next section.

5.4 Modifying the Example to be a Counter-Example in CLASSIC

Once an individual `Ind` has been generated by `GetInd` such that it is structurally an instance of the alleged subsumee, `D`, the next task is for the `Contradict` procedure to check I's relationship to the alleged subsumer concept, `E`. If it is structurally not an instance of `E`, it is truly a counter-example and we are done. Otherwise we try to modify it so that it ceases to be an instance of `E`. This modification again depends on concept constructors involved, and, as we shall see, on the DL itself. Note that because we had started to build `Ind` by asserting `Ind` $\rightarrow$ `D`, any attempt to modify `Ind` in such a way that it would stop being an instance of `D` will be detected as an inconsistency. However, adding a new filler to a role could make closed(`Ind`) be inconsistent again.

In CLASSIC, if `E` is constructed with **at-least**, **one-of**, **fills**, **prim**, **min**, or **max** and `I` is initially an instance of `E`, then there is no way to add information to I in order for it to stop being an instance of `E`.

Therefore, we need to consider the case when concepts are constructed with **and**, **at-most**, or **all**. Conjunction is dealt with by trying to successively find a contradiction to each conjunct, so we are left with two cases to consider.

If `E` has the form (**at-most** n `r`), with $n <$ MAXINT, then the algorithm will try to add `r`-fillers to the individual to violate the **at-most** constraint. While if `E` has the form (**all** `r C`), then the algorithm will try to either add an `r`-filler that is not a `C`, or add something to one of the `r`-fillers that will make it stop being a `C`.

There will be a few challenges for the algorithm. First, it will need to recognize when no effort it makes will succeed in modifying the individual so that it violates the alleged subsumer. For example, consider a situation where `I1` is an instance of `D` $\equiv$ (**at-most** 0 `r`), while `E` $\equiv$ (**at-most** 1 `r`). If we try to add further `r`-fillers to `I1`, we will contradict the assertion `I1` $\rightarrow$ `D`. This problem is relatively straight forward to solve – in this particular case, the algorithm can just check the **at-most** restriction on the alleged subsumee and if that number is $\leq$ the **at-most** on the alleged subsumer, it is not possible to succeed.

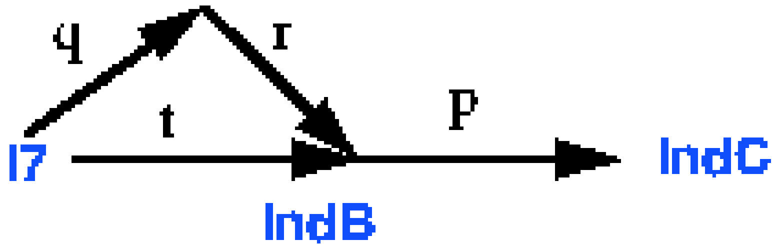

Figure 5.5: **An Object with the same filler along two role paths**

Another problematic situation will arise when the contradiction algorithm will attempt to add a filler to a role, obtaining a conflict as a result, but it will be possible to recover from this conflict. Since all other additions to the individual are forced, the repair can consist of retracting a previous filler added to the role in order to meet some constraint, and putting in a different filler for the role. In our case, this will be equivalent to merging a Skolem constant with some other individual.

Consider the following description (for the alleged subsumee concept):

$$
\begin{array}{lll}
\texttt{D7} \equiv (\textbf{and} & (\textbf{all}\ \mathrm{q}\ (\textbf{and} & (\textbf{fills}\ \mathrm{r}\ \mathrm{IndB}) \\
 & & (\textbf{all}\ \mathrm{r}\ (\textbf{fills}\ \mathrm{p}\ \mathrm{IndC})))) \\
 & (\textbf{all}\ \mathrm{t}\ (\textbf{and} & (\textbf{prim}\ \mathrm{F}) \\
 & & (\textbf{at-least}\ 1\ \mathrm{p}))) \\
 & (\textbf{fills}\ \mathrm{t}\ \mathrm{IndB}))
\end{array}
$$

An instance, `I7`, of this description is presented graphically in Figure 5.5.

When `D7` is created in the knowledge base, no information is deduced about `IndB`, because concept definitions do not affect the contingent state of the world. Once an instance, say I7, of D7 is created, propagation inference applies to fillers of roles. Since there was no requirement that there be a q-filler on I7, nothing is inferred there; in particular, no propagation caused IndC to become p-filler on IndB. On the other hand, on role t, propagation applies to filler IndB, and as a result IndB is inferred to have description (**at-least** 1 p); as a consequence of that, GetInd actually puts in an anonymous individual x as the p-filler for IndB.

Suppose now we get concept E7 ≡ (**at-most** 0 q). If we want I7 to contradict E7, then we will need a filler for q, and once this is added, IndC will show up as a p-filler on IndB. Therefore, the original use of x was superfluous, since IndC is there to do the job. If p had been declared as an attribute, this would in fact lead to an error.

If we now retracted x as the filler for p on IndB, and used IndC instead, everything would work out and it would be possible to put in an anonymous filler for q. Note that this kind of problem can occur only in the following circumstances: a branch of the concept which had been "dormant" is activated by the first individual to be added to a role, and as a result some named individual is added as an extra filler to some other named individual, causing a conflict.

For readers who desire a more complete explanation of this situation, we include an annotated CLASSIC trace of the scenario described in Appendix D. It carefully notes properties of IndB at critical points.

In the previous scenario, we discovered a problem by trying to add an anonymous individual to the individual we are trying to make into a counter-example. If a previous anonymous individual is retracted and replaced by a named individual, it is possible to put in the new anonymous individual as a filler and create the appropriate counter-example. Another scenario can be generated where a problem is discovered when an anonymous individual is added to the potential counter-example. In this scenario, the appropriate repair is to avoid adding the anonymous individual completely and instead add a particular named individual. Consider the following description for the alleged subsumee:

D2 ≡ (**and** (**fills** p Indb)

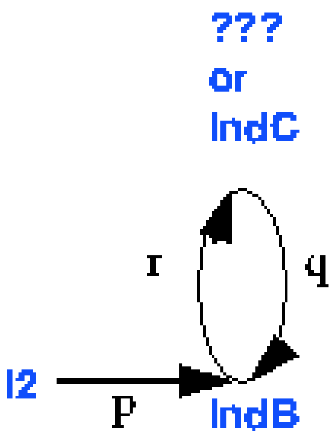

Figure 5.6: **Another object with the same filler along two role paths**

(**all** p (**and** (**all** r (**fills** q `Indb`)))
(**all** r (**all** q (**fills** r `Indc`)))))

An instance, `I2`, of this description is presented graphically in Figure 5.6.

Again, CLASSIC will not deduce anything about `IndB`'s `r`-filler from the definition of `D2`. Upon creation of the individual `I2` $\rightarrow$ `D2`, we will discover that `IndB` has restrictions on the fillers of `r` but we do not yet know that it must have any `r`-fillers. A perfectly good example instance of `D2` has no fillers for the `r` role on `IndB`. However, if we have `E2` $\equiv$ (**all** p (**at-most** 0 `r`)), then we will want to add a new, anonymous individual as an `r`-filler. This individual then triggers propagations which make `IndC` show up as a `r`-filler for `IndB` also, thus providing an embarrassment of riches (possibly even a contradiction if there had been an (**at-most** 1 r) restriction). In an error situation, we would need to take out the original anonymous filler and, instead, insert the newly found, named individual `IndC`.

These examples show that our `Contradict` function must, under certain conditions, retract appropriate role fillers. Our algorithm identifies those conditions by looking for a bounds conflict on a role that has an anonymous filler in it. Detecting the particular kind of error and taking the appropriate remedial action is handled in our supporting function called `Repair` shown in Figure 5.8.

The `Contradict` function needs to take an example individual which is structurally an instance of an alleged subsumee and appropriately modify the individual (if necessary) to make it structurally not an instance of the alleged subsumer. It will use the supporting function `Repair` to handle the problems just presented. The pseudo-code for Contradict appears in Figure 5.7. In order to simplify algorithm presentation, we will assume that both `Contradict` and `Repair` keep a current list of assertions. If they fail, before returning fail, they undo the assertions that they made.

Discussion (This discussion could be skipped on an initial reading.) At this point, we will consider what it means for `Contradict` to return success. We need to make sure that `ClosedForNonMemb(Ind, E)` is consistent and it is $-s \not\rightarrow$ `E`, while at the same time ensure that `ClosedForMemb(Ind,D)` remains consistent (and hence $-s \rightarrow$ D).

The argument is by induction on the number of **and** and **all** constructors in E.

If `E` has the form (**prim** `Aset`), (**one-of** `Oset`), (**min** n), or (**max** m), then `Contradict` made no changes to the individual thus its relationship to D is unchanged. `Contradict` returned success because `Ind` $\not\rightarrow$ `E`. In the case of these constructors, $\not\rightarrow$ is identical to $-s \not\rightarrow$.

If `E` has the form: (**fills** `r Bset`), then `Contradict` has made no changes to the individual. $\not\rightarrow$ is identical to $-s \not\rightarrow$ for **fills**. Additionally, `CloseForNonmemb` must be able to close the `r` role because of the assumption that `CloseForMemb(Ind, D)` $-s \rightarrow$ `D`.

If `E` has the form: (**at-least** k p) then there are two possible courses. (1) `Ind` had fewer than k p-fillers so there was no change to `Ind`. `Contradict` has counted the fillers for the **at-least** role so it has checked for structural recognition. Additionally, `CloseForNonmemb` must be able to close the `p` role because of the assumption that `CloseForMemb(Ind,D)` $-s \rightarrow$ `D`. (2) `Contradict` removed the anonymous filler (every individual has at most one anonymous filler per role, possibly made superfluous by some

```
Contradict(i,D,E)
assume:   i consistent and completed, CloseForMemb(i,D) -s → D,
          D has no one-of
effect:   returns success, with i incremented so CloseForMemb(i,D) -s → D
          and ClosedForNonmemb(i, E) -s ↛ E
          OR returns failure
{
  if E subsumes D then return fail
  case E=  (prim Aset), (fills p Bset), (one-of Oset), (min n), or (max m)
           if i ↛ E then return success else return failure
  case E= (at-least k p)
           if i has fewer than k p-fillers then return success
           else if no anonymous p-filler or more than k fillers then return failure
               else { retract the addition of the anonymous filler
                   if i's at-least bound is still met   ;; i.e. i -s → D
                   then return success
                   else put back the filler x and return fail };
  case E=  (at-most k p)
           if i has more than k p-fillers
               then return success ;
           if i → E then return failure   ;; no way to violate E
           if i has anonymous p-filler x,  ;; it can be duplicated at will since i ↛ E
               then return success
           generate anonymous ind x
           add x as p-filler of i    ****
           C = cl-all(i,p)
           if adding x causes a CLASSIC failure
           then Repair(error,i,p,x)
                 if success and i now has more than k fillers
                 then return success
                 else return failure
           else Complete(x,C)
               return success
```

```
case E=  (all p F)
         if some p-filler of i is  -s ↛  F then return success;
         let C= cl-all(i,p)                ;; might be THING
         for every p-filler j on i
             if Contradict(j,C,F) succeeds
             then return success else continue
         if role p is full on i or already has an anon filler, then fail
         else { generate new anonymous individual x
             cl-ind-add(i, (fills p x)), ****
             if CLASSIC failure
             then if Repair(error,i,p,x) fails, return failure
             Complete(x,C)                 ;; makes sure x  -s →  C
             if Contradict(x,C,F) successful, return success
             else return failure }
case E= (and C1 C2)
         Contradict(i,D,C1);
         if success then return success
         Contradict(i,D,C2);
         if success then return success else return fail
}
```

Figure 5.7: Contradict Pseudo-code

later addition.). Our check made sure that `CloseForMemb` would still work (since D has been asserted of Ind) in the sense that we could make enough copies of the anonymous filler to satisfy D. And of course, as a result of the removal, `Ind` is structurally an instance of `E` since `Contradict` counted k fillers just before.

If `E` has the form: (**at-most** k `p`), then one of two routes is followed. Either (1) `Ind` already had enough `p` fillers to violate the **at-most** restriction in which case things are ok as usual. Or (2) `Contradict` has successfully added either an anonymous fillers to `p` to violate the **at-most** restriction, or replaced it with a named individual in Repair. Either way, `Ind` has enough fillers to be structurally a non-instance of `E`. We have also made sure that the new filler can also be closed with respect to D.

If `E` has the form: (**all** `r F`), then several courses could have been taken. (1) There currently existed a `r` filler that was structurally not an instance of `F`. Then `Ind` is unchanged and is structurally not an instance of `E`. Since there was no change to Ind, and CloseForMemb used to work, so will CloseForNonMemb. (2) There was a p-filler

```
Repair(error,I,p,x)
;; This algorithm tries to find the one individual that could have
;; avoided the error if it was used instead of the earlier anonymous addition.
;; It will attempt to take out an anonymous ind and replace it with
;; a named individual. This, together with closing the inserted named individual
;; This is equivalent to merging the anonymous individual with the named individual.
{ if error ≠ BoundsConflictError then return failure;
  from BoundsConflictError object identify named individual B on which
        conflict occurs, as well as role *r* and the set of all its fillers F-set
        at the time of the error;
  if no element of F-set is anonymous, return fail
  else suppose y in F-set is the anonymous individual;
  let G-set be set of current fillers of *r* on B (after error recovery)
  if y ≠ x then remove y as r-filler of B
  let H = F-set − G-set −{y} ;; these are the fillers implied by the addition of x
  if H is empty, then return fail ;; the error was not fixable by merging
  add all elements of H as r-fillers to B
  if CLASSIC fails then return fail ;; no repair possible
  if B ≠ I
  then { add x as p-filler on I again ;; to restart Contradict process on I
        if fail then recursively call Repair(error,I,p,x)
        C = cl-all(I,p)
        Complete(x,C) }
  C = cl-all(B,r)
  Complete(h,C) for all h in H ;
  return success }
```

Figure 5.8: Repair Auxiliary Function

j on `Ind` that could be made to contradict `F`. Here we use the induction hypothesis (noting that all previous cases were base cases for the induction.) Therefore, the p-filler j is structurally not an instance of `F`, and hence `Ind` is structurally not an instance of `E`. In addition, j can be closed by induction, and hence so can `i`. (3) `Contradict` was successfully able to add a new filler that is structurally not an instance of `F`. After completing it with respect to D, we can safely close it for membership.

The only additional argument for this case is when Repair was used. If `Repair` succeeded, it replaced an anonymous individual with another individual. Because of propagation, this individual will have the desired properties, except possibly not being able to be closed for membership, because it just was assigned an **at-least** bound in `F` without the required number of known fillers. For this reason we invoke `Complete` on it.

If `E` has the form: (**and** C1 C2), then the argument follows by induction. C1 and C2 must have one of the forms mentioned above. If we have shown that `Contradict` correctly succeeds on each of C1 and C2, then it must correctly succeed on their conjunction.

Next we should consider what it means for `Contradict` to return failure, indicating that it we could not make `Ind` be a non-instance of `E`. We will again proceed by induction on the form of `E`.

If `E` had the form: (**prim** Aset), (**fills** r Bset), (**one-of** Oset), (**min** n), (**max** m), there is nothing `Contradict` can add to `Ind` to make it a non-instance of `E` if it is currently an instance of `E`.

If `E` had the form: (**at-least** k p), then there is nothing that `Contradict` can add to `Ind` to make it violate `E`. Since there is only one anonymous individual kept per role, if there are more than k fillers or no anonymous ones, we cannot take enough things out in order to fail to meet the **at-least** bound. Finally, even if there is an anonymous individual, it might have been really needed to make i be closeable for membership in D because of an **at-least** bound (which was asserted of i), and we discover this once we retract the filler.

If `E` had the form: (**at-most** n r), the one of two courses is followed. (1) `Ind` already has an **at-most** restriction that is imposed on it by virtue of it being an instance of `D` which causes subsumption to hold. In this case, `Contradict` must fail. (2) `Contradict` was not able to add an anonymous individual to the role, and `Repair` was not able to replace it by sufficiently many named ones, so failure results.

If `E` has the form (**all** `r C`), then `Contradict` failed because it could not make any known filler into a non-instance of `C` and it could not add a filler that was a non-instance of `C`. Once again, since there are recursive calls to `Contradict`, this is an argument by induction.

If `E` has the form: (**and** C1 C2), then we will make an argument by induction. Assume that C1 and C2 do not contain **and**. Then they must have one of the forms mentioned above. If `Ind` is an instance of all of the conjuncts, (which is what it means to fail to contradict them), then it must be an instance of the conjunction of them. If we have shown that `Contradict` correctly fails for all of the conjunctions, then it must fail on their conjunction.

Finally, consider the repair strategy. The only way that CLASSIC will deduce an inconsistency that is not implied by the logic is for our algorithm to have placed an anonymous filler somewhere where a named individual should appear. (This is because CLASSIC is sound so it will never deduce properties of objects that are not implied by the logic.) Our solution exploits the error handling mechanism in CLASSIC.

If our algorithm has generated an anonymous individual that should be merged with a named individual, then CLASSIC will detect a bounds conflict on some role, *r*, of some individual, B, where there are too many fillers now, and one of these fillers is anonymous. CLASSIC maintains error objects that capture objects at the point where an error has been detected and can thus identify *all* the fillers of B in its *intermediate* state. It is from that error object that the bounds conflict message is generated. The fillers will contain an anonymous individual that needs to be merged with one of the named individual role-fillers which showed up to cause the conflict. If the addition of the original filler x, caused several individuals to be added as *r*-fillers, all of them will have to be added instead of y later in any case, so we might as well do it now.

5.5 Dealing with Incomplete Subsumption

As noted in Section 2.3, many implementations of DLs used in practice choose efficiency over completeness (e.g., BACK, CLASSIC), or are incomplete even in principle (e.g., LOOM) because the corresponding logic is only semi-decidable. A sound but incomplete subsumption algorithm may yield unsound *non*-subsumption judgements because if C logically implies D, and an incomplete algorithm misses this, a non-subsumption determination based on the same algorithm will incorrectly judge C not to be subsumed by D.

Practical experience strongly suggests that users want to be able to ask the system to distinguish between the case where subsumption was not detected due to incompleteness of the implementation from the case where it is a consequence of the logic. A good explanation system should ideally be able to make such a distinction for its users.

As mentioned earlier, CLASSIC is one example of a DL that is incomplete in its subsumption reasoning, since it does not take into account the properties of individuals appearing in **one-of** constructs, treating them instead almost as disjoint primitive concept names. For example, the concept (**and** (**all** p D) (**fills** p I1) (**all** q (**one-of** I1))) is not subsumed by (**all** q D), even though I1 must in fact be an instance of D.

There are several approaches we may take to handle the problem of detecting cases of incomplete reasoning, when pressed for such diagnoses as part of an explanation mechanism.

The first uses a different, non-performance oriented but complete reasoner to check if a particular subsumption relationship holds. If the complete system does find a positive subsumption relationship, the explanation of the non-subsumption judgement is the incompleteness of the performance system. For example, CLASSIC might be connected to a First Order Predicate Calculus (FOPC) theorem prover, which would reason with the translation of DL concepts into FOPC formulas.

This might pose logistical problems though, related to space requirements for two implemented systems, possibly non-terminating procedures in the theorem prover, or simply multiple license fees or too many systems required to do one task.

If the explanation designer can not connect to some existing complete reasoner, she may try to generate the additional inference rules that would make the inference set complete, implement the additional rules, and retry subsumption using the extended implementation. Note that the additional rules will be normalization rules since the missing inferences are not due to missing structural comparisons, but instead missing implications from the normal form.

Another approach uses heuristics to determine which specific set of missing inferences is responsible for the failure to derive subsumption. For example, in CLASSIC, properties of individuals are not taken into account when determining concept subsumption. A non-subsumption explanation module using heuristics appropriate for CLASSIC could first check if an individual is involved and then check if the current properties of that individual would make the subsumption relationship hold. Given the primitive concept `AMERICAN-REGION` and an individual `Napa` which is an `AMERICAN-REGION`, and definitions for the concepts `AMERICAN-THING (AT)` and `NAPA-THING (NT)` as:

`AT` = (**all** `location AMERICAN-REGION`)

`NT` = (**and** (**at-most** 1 `location`) (**fills** `location Napa`)),

CLASSIC would not determine that the concept `NT` is subsumed by `AT`. It would not deduce a value restriction on the `location` role of `NT` based on properties of the individual `Napa`, and thus `NT` would not imply the all restriction on `AT`. The explanation system could do additional reasoning to see if `AT` would subsume `NT` if properties of Napa were considered[7] and if so, state that the non-subsumption is due to CLASSIC's decision not to use individual properties in concept subsumption. If a system designer can find a small number of heuristics that can be implemented efficiently and that cover most of the types of incompleteness of the performance system, then the heuristic approach should be considered.

Finally, the technique introduced earlier to explain non-subsumption can be used

[7]This might be done simply by using the individual reasoner to see if `NT` would be an instance of `AT`. This simple solution works only when the individual reasoner is complete.

as follows: if an individual counter-example is found, then we know that we do not have a case of incomplete reasoning since the subsumption is semantically impossible. If the technique for finding counter-examples can be proven to be complete, then in fact the absence of a counter-example is a proof that the subsumption relationship does in fact hold, and we must be facing an incompleteness. We must however point out that with this particular approach we do not get a very acceptable explanation of why the subsumption *does* hold because the proof is essentially non-constructive: it rests again on not being able to find something.

Note that in KRIS, the absence of a counter-example is correlated with the presence of a "clash"/inconsistency, which can then be traced back to obtain an argument by contradiction of why E must subsume C.

5.6 Example of Nonsubsumption Using `GenerateCounterExample`

In order to make this process more concrete, let us look at a specific scenario and see how the algorithm will function. We will choose a moderately challenging example – one in which CLASSIC's incompleteness causes a subsumption relationship to be missed. What will happen is that the algorithm will generate an instance of the alleged subsumee, it will then try to modify this individual to make it a non-instance of the alleged subsumer and find that this is impossible. The problem will not be one that our repair strategy can fix (appropriately so since this is a "real" incoherence forced by the logic) and thus, the algorithm will show that the subsumption relationship really should have held.

We will provide this example in the form of a CLASSIC trace. An instance of `D`, below is shown in Figure 5.9.

```
(cl-define-roles '(p s q r))
(cl-define-concept 'D
                  '(and (fills p IndB)
                        (all p (atmost 0 s))
                        (all p (all r (and (fills q IndB)
                                           (all q (fills s IndC))))))
```

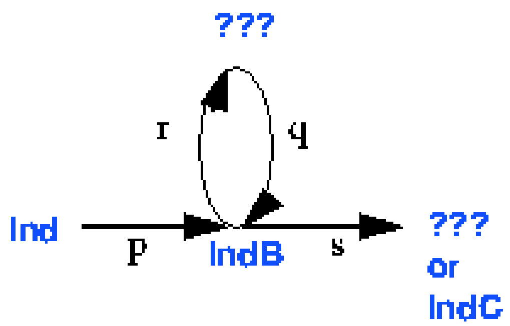

Figure 5.9: **A Non-subsumption Example**

```
(cl-define-concept 'E '(and (fills p IndB)
                            (all p (and (atmost 0 s)
                                        (atmost 0 r)))))
USER(111): (cl-subsumes? @E @D)
NIL
```

This shows that CLASSIC does *not* derive that `E` subsumes `D`. We can use CLASSIC to identify the reason - CLASSIC has not deduced an **at-most** restriction for p's `r`s. Thus, the current **at-most** restriction is T which is CLASSIC's way of saying there is no more specific **at-most** restriction than MAXINT. Thus, the explanation says that there is a bad **at-most** ordering which has caused the subsumption relationship not to hold.

```
USER(112): (cl-exp-not-subsumes-concept @e @d)
D ->
 Role Restrictions:
  P
   All: :
```

```
    Role Restrictions:
     R
      at-most: (T) :
        Subsumption:
          BAD-AT-MOST-ORDERING: At-most: T doesn't satisfy at-most: 0
@c{D}
```

The `GenerateCounterExample` routine will first call `GetInd` to produce an individual instance of `D`. There are no **one-of**s in this example so `GetInd` just normalizes `D` and uses `CreateInd` which only calls the CLASSIC function to create a new individual instance of `D`.

```
USER(113): (cl-create-ind 'i1 'd)
@i{I1}
USER(114): (cl-print-ind @i1)
I1 ->
 Derived Information:
  Parents: D
  Ancestors: THING CLASSIC-THING
  Role Fillers and Restrictions:
   P[1 ; INF] -> IndB
     => Role Restrictions:
     S[0 ; 0]
     R =>
      Role Restrictions:
       Q[1 ; INF] -> IndB
         => Role Restrictions:
         S[1 ; INF] -> IndC
@i{I1}
```

Next I1 needs to be completed. This will mean that IndB also has to be completed since IndB is a p-filler for I1, which has an **all** restriction on p. Neither I1 nor IndB however have any unmet **at-least** restrictions, so Complete will not generate any anonymous fillers. Also, CloseForMemb on I1 and IndB would be able to close any open role and I1 is structurally an instance of D (and IndB is structurally an instance of (cl-all(I1, p))). Thus, GetInd has successfully produced an instance of D.

Next, it is time to let Contradict work to make I1 $-s \not\rightarrow$ E. The critical part of E that we need to contradict is: (**all** p (**at-most** 0 r). Thus, we need to get the p-filler, IndB and see if it currently is not an instance of (**at-most** 0 r) or if it can be made to be a non-instance. This will mean that we will call Contradict on IndB making sure that it remains an instance of the **all** restriction on D's p role yet trying to violate (**at-most** 0 r). Contradict will attempt to add a new (anonymous) filler for r. This is about to cause an error in CLASSIC so we surround the call with a call that will bind all the error return values.

```
USER(115): (multiple-value-setq (v1 v2 v3 v4)
     (cl-ind-add @IndB '(fills r newind)))
*WARNING*: Inconsistent number restriction for role @r{S} - at-least: 1;
  at-most: 0. Found while processing object @i{IndB}.
*WARNING*: Retracting addition of expression.
*CLASSIC ERROR* while processing
  (CL-IND-ADD @i{IndB} (FILLS R NEWIND))
    occurred on object @i{IndB-*INCOHERENT*}
    along role-path (@r{S}):
  Inconsistent number restriction for role @r{S} - at-least: 1; at-most: 0.
CLASSIC-ERROR
USER(116): (cl-print-ind v4)
IndB-*Incoherent* -> : (INCONSISTENT-BOUNDS-CONFLICT @r{S} 1 0)
 Derived Information:
  Parents: CLASSIC-THING
  Ancestors: THING
```

```
  Role Fillers and Restrictions:
   R[1 ; INF] -> Newind-*Copy*
     => Role Restrictions:
     Q[1 ; INF] -> IndB-*Incoherent*
       => Role Restrictions:
       S[1 ; INF] -> IndC
   S[1 ; 0] -> IndC
@i{IndB-*INCOHERENT*}
```

This showed that CLASSIC discovered an error when it tried to add an anonymous filler for `r`. The error that it discovered is a bounds conflict but it does not contain an anonymous individual. The error is that `IndC` has been deduced to be a filler for a role that can have **at-most** 0 fillers. It is impossible to repair this problem, thus it is impossible to create a counter-example individual. Thus, the algorithm will conclude that the subsumption relationship would have held in a complete implementation.

5.7 Some Non-Subsumption Uses and Extensions

We will mention some of the ways that our work on non-subsumption can be used and extended.

Non-Normalization: A user may be interested in explaining non-normalization, i.e.,, why is `AD` not part of the description of `O` where `AD` uses constructor *con* on rolepath rr ? A particularly simple approach just explains the normalization along rolepath rr with respect to constructor *con*. If `O` is a concept, we can use our algorithm to explain why `O` $\not\Longrightarrow$ `AD`. If `O` is an individual, we can get the normalized description of `O`[8], call it `ON`, and explain why `ON` $\not\Longrightarrow$ `AD`.

General Non-Recognition: We may need to explain a general non-recognition issue, i.e., explain why `Ind` $\not\rightarrow$ `E` when `Ind` does not necessarily contain *all* of its filler and primitive information. In this case, we can first try to use the closed world structural

[8]In CLASSIC, this can be obtained by using cl-ind-descr(O).

non-recognition rules if `Ind` happens to be closed with respect to the knowledge base. If information on `Ind` is missing, obtain the description of the normalized form of `Ind`, which we will call `IN` and then explain why `IN` $\not\Longrightarrow$ `E`.

Atomic Descriptions: `GenerateCounterExample` could save work if it chose an atomic description `AD` such that `E` $\Longrightarrow$ `AD` that it would pass to `Contradict`. The key would be to choose the `AD`s such that they are easier –i.e., we should only choose an **all** restriction if we had to, and only choose an **at-most** restriction if there are no other options other than **all** restrictions. The other restrictions are all simply checked so `Contradict` would do much less work when it tried to generate a counter-example. In a complete implementation, if the algorithm fails to produce a counter-example for *any* `AD`, then the subsumption relationship should hold. However, given that an implementation may be incomplete, then it could be the case that no counter-example exists for this `AD` because of the incompleteness, but another a counter-example may exist for another `AD` of `E`, thus *all* `AD`s must be tried before the algorithm can conclude subsumption should have held.

Extensions: There are many ways that non-subsumption could be extended. One may be motivated by a user who would like to have suggestions about ways to deduce a particular subsumption relationship. Since there are an an infinite number of such suggestions, we must rely on a heuristic solution. In concept reasoning, we may retrieve the parents of the subsumed concept and see if any of those concept's children have the atomic descriptions we are missing. If so, the system could suggest that the user add that child concept to the description of the concept in question. In the case of individuals, it may be useful to look for rules that conclude the missing atomic descriptions and then note that if the individual were an instance of the antecedent of the rule, then the subsumption relationship would hold.

Sameas Considerations: As we mentioned, our algorithm did not handle the **same-as** constructor. Adding **same-as** violates one of our assumptions; now it is possible for an anonymous individual to be *derived* as a filler of a role instead of only being *told* to fill a role. Thus, our repair strategy could not assume that it is always possible to directly retract that some anonymous individual fills a role. Now the repair strategy must be able to figure out that the anonymous individual was derived as the result of an assertion that it fills some other role on another individual. One solution to this problem is to maintain a history list of anonymous individual assertions. Then when an anonymous individual must be retracted, the repair strategy can look through the history list and determine which (unique) assertion should be retracted. Since we always generate new anonymous individuals, there will only be one entry containing any particular anonymous individual. The **** lines in the algorithm show the places where there would need to be a statement "add the cl-ind-add of the anonymous individual to the history list". Also, repair now needs to distinguish between `B` in the algorithm – the individual on which the error occurred – and the individual on which the anonymous individual needs to be retracted.

5.8 Summary

DLs require an explanation service that handles both positive and negative deductions. In this chapter, we have proposed a counter-example-based approach to explanations of negative subsumption determinations. In this approach, we generate a counter-example, i.e., an individual `Ind` such that `Ind` $\rightarrow$ alleged subsumee and `Ind` $\not\rightarrow$ alleged subsumer. The counter-example is restricted in such a way that it contains all of its filler and primitive information and all of its fillers are restricted the same way. Given this restriction, it is a simple matter to deduce and explain why `Ind` is or is not recognized to be an instance of a description. We have presented an algorithm for the CLASSIC knowledge representation system.

If such an algorithm is complete, as we believe ours to be[9], it solves the problem of identifying when a subsumption relationship has not been detected only because of an incomplete implementation. In our solution, if a counter-example can not be generated, then a subsumption relationship should have held.

This approach could be used to augment CLASSIC's subsumption checking. (The

[9]We have provided only relatively informal correctness arguments for key procedures. Of course, we can claim that our algorithm is more complete than CLASSIC since it augments CLASSIC's subsumption algorithm by taking more properties of individuals into account in the reasoning process.

simplest integration would involve calling the explanation function after *not* detecting a subsumption when individuals are involved in the descriptions. A more substantial integration effort would be needed to integrate the idea of generating closed individual instances of descriptions into the subsumption algorithms.) Given a complete counter-example component, this would result in a complete subsumption detection. The penalty would be an exponential algorithm (due to exploring all possible **one-of** combinations) where previously CLASSIC had an incomplete[10] but polynomial time subsumption algorithm. This could be very useful in applications where the use of **one-of** is limited and more complete reasoning is desirable.

[10]CLASSIC is incomplete with respect to the standard semantics and complete with respect to an alternate semantics.

Chapter 6

Filtered Views

In large applications, objects and their descriptions may become too large and complex to view in traditional ways. For example, when asked to explain individuals or concepts, the number of roles alone might overwhelm the user, thereby negating the benefits of even the most abbreviated proof presentation technique. In fact, even printing or browsing — the simplest precursors to explanation as a debugging technique — can become ineffective if too much information is thrown at the user. In order to avoid inundating users with too much detail, it is important to be able to delineate "interesting aspects" of objects.

We address the problem of specifying such aspects of objects for knowledge bases that support Description Logics. We illustrate the general technique by providing an example language for specifying interesting properties of classes and individuals in CLASSIC. In keeping with the philosophy of CLASSIC, the basic language is relatively limited, but we support procedurally-defined patterns as a way of escaping these limitations.

This language provides a way for the user to provide a pattern specifying interesting aspects, and then match objects (classes or individuals) against the pattern. The result of the match is a substituition containing the information of interest. This result can then be used to produce views that can be shown to developers (for debugging), or even to end-users. The language of patterns makes use of meta-individuals and meta-relationships that describe the logical or textual structure of the knowledge base. An earlier design for filtering is implemented and is in use in CLASSIC applications. The current design is not implemented.

6.1 Pruning in Description Logics

Real-world knowledge bases (KBs) are often quite large because of the variety of details that they capture. Description Logic-based KBs are no exception — in fact, they tend to be particularly complex because they facilitate the generation of highly structured and/or interconnected objects. In many situations, while building, debugging, or browsing the KB, it is desirable to be able to limit the information that is displayed about a class or individual object. For example, while developing a stereo-system configurator, we may be in a situation where we are debugging the inferences concerning the choice of appropriate components, given some specific goal of acoustic fidelity. In this case, we want to see those roles or attributes of the stereo that describe the actual components (e.g., amplifier, speakers, etc.) and their technical properties relating to acoustics (e.g., power rating, price, etc.). Conversely, if we are evaluating the expected reliability of the stereo, then the past repair history of the models and their manufacturers become relevant. In neither of these cases is one interested in the attributes of the stereo components which offer bit-maps for pictures to be displayed for them.

In relational database systems, such selective filters are obtained through the definition of so-called "views", which are queries that select and organize the relevant information. In object-oriented programming languages, this is achieved by writing one or more so-called "print-functions" for each class of objects. Such functions usually select a subset of the "data members" of the class, and create a string with some desired format from them.

What makes the task different in our case is the desire to show the nested structure of objects, and to do so in a way which may be context-dependent. For example, when viewing a stereo system as a whole, we may wish to see its components, including its speakers, but we may be interested only in a speaker's maker, and model number; however, when a speaker is viewed independently, more information, possibly concerning price, power rating and performance, may be required.

In addition, we want to be able to specify various criteria of "interestingness" so that uninteresting information is not presented. For example, all makers of stereos are known to be companies, so we may only be interested in that information about the `maker` which is **not** known by virtue of it being an instance of the concept `COMPANY`, i.e., the restriction is strictly more specific than `COMPANY` .

Finally, for the purposes of explanation and debugging, we may want to ask about information deduced in some special way, e.g., by the firing of some particular rules.

Our goal is to provide a declarative language for specifying such views in DL-based knowledgebase applications. In order to facilitate learning such a specification language, we propose to build on the existing syntactic structure of DLs. We will separate the task of selecting the information to be shown from the task of formatting it for display on screen or paper, concentrating here exclusively on the former.

The relevant information will be specified by means of concept patterns, which may be *matched* against individuals and concepts in the KB. These patterns may be stored with classes and then used every time a subclass or instance of the class is printed or explained. Additionally, a pattern could be specified by the user when explaining or printing any particular object. Following a time-honored approach, we shall introduce our proposed language by means of a series of examples. (The grammar appears in Figures 6.1 and 6.2 and the matching algorithm is described in Section 6.3.)

6.2 Specifying Concept Patterns

In this section, we will give concrete motivation for and examples of many of the aspects of our language. We give examples of the base description logic language for patterns augmented with epistemic constructors, a pattern combination operator, and an extensible constructor similar to CLASSIC's **test** constructor.

Suppose we have an individual stereo system, called `Stereo2`, described as having exactly one `price` which is an integer between 300 and 2000. Also, the stereo system has exactly one `amplifier` and that `amplifier` is a particular individual, named `Adcom55`. The system also has at least 4 `speakers`, and two of them are `Speaker1` and `Speaker2` (the other two `speakers` are not known yet). We can state that in CLASSIC as follows:

```
Stereo2 → (and (at-least 1 price) (at-most 1 price)
               (all price INTEGER)
               (all price (and (min 300) (max 2000)))
               (at-least 1 amplifier) (at-most 1 amplifier)
               (fills amplifier Adcom55)
               (at-least 4 speaker)
               (fills speaker Speaker1 Speaker2))
```

The basic idea of our approach is to use variables (marked by an initial _) in descriptions as a way of specifying information that needs to be matched and returned (presumably for later display). Thus, if we were interested in the `amplifier` for the stereo, we would use the concept pattern

(**fills** `amplifier _A`)

to obtain the actual filler of the attribute as the substitution for _A. Matching this concept pattern against `Stereo2` would succeed, returning the substitution [_A $\mapsto$`Adcom55`].

On the other hand, if we were to match the concept pattern

(**fills** `price` _P)

against `Stereo2`, the match would fail, since no `price` is known. In this case, we may settle for more general information about the `price`, which might be obtained by looking for restrictions that would apply to any eventual filler for this attribute. In CLASSIC, such information would appear as an **all** restriction, and could be sought by matching the concept pattern

(**all** `price` _R)

whose match against `Stereo2` would succeed with the substitution [_R $\mapsto$(**and** `INTEGER` (**min** 300) (**max** 2000))]. If we were interested only in the upper and lower bounds of the range of values which the `price` might take, we could use the more specific concept pattern

(**all** `price` (**and** (**min** _X) (**max** _Y)))

which, when matched against `Stereo2`, would yield the substitution [_X $\mapsto$300, _Y

$\mapsto$2000].

Suppose however that we want to combine the two previous cases, preferring to obtain the more specific information whenever possible: if a value for `price` is known, that is desired; otherwise, we settle for the range of `prices` inferred so far.

This means that we want to match for **min** and **max** only if no actual `price` is known. For this, it is tempting to use the concept pattern

(**and** (**at-most** 0 `price`) (**all** `price` (**and** (**min** _X) (**max** _Y))))

but this would be a mistake since the stereo must have a `price` (though its value may not yet be known) so (**at-most** 0 `price`) would cause the match against `Stereo2` to fail immediately. To eliminate this problem, we augment our language with some constructors that provide access to the epistemic state[1] of the KB: (**known** p) is a role whose fillers are exactly the ones currently known for p. This allows us to express the correct concept pattern for `price` ranges as

(**and** (**at-most** 0 (**known** `price`)) (**all** `price` (**and** (**min** _X) (**max** _Y)))).

Another epistemic constructor we have found useful is **notKnownToFill**. This constructor is particularly necessary in expressively limited languages such as CLASSIC that do not contain a **not** constructor. A simple use of this would allow users to match stereo systems against a pattern that disallowed a particular component. For example, matching `Stereo2` against the pattern:

(**notKnownToFill** `amplifier Adcom55`)

would fail. It would not be desirable to allow unbound variables in **notKnownToFill** expressions because the match algorithm would then potentially need to return all the individuals in the knowledge base. As a result, we require either named individuals or previously bound variables in **notKnownToFill** expressions. We will see more examples of the usefulness of this constructor once we introduce the notion of meta-roles in Section 6.2.1. We should note that these epistemic constructors are only applicable in patterns matched against individuals, not against concepts.

---

[1]Our epistemic constructors were motivated by the needs of our applications and were heavily influenced by the history of literature of epistemic additions to description logics, such as [49, 50, 61].

One last epistemic constructor is called **closed**.[2] This lets users match against roles that have all the fillers they are going to have. This might be useful when a configuration system needs to check if the parts list has been completed and thus is ready to be sent to the factory. For example, if we match `Stereo2` against

(**closed** `amplifier`)

we would succeed since there is one known filler and that role can have at most one filler so the system would have closed the role. Conversely, if we matched `Stereo2` against

(**closed** `speaker`)

we would fail since there are only two known fillers and we know there needs to be four fillers, so that role can not be closed yet.

Finally, we need to have a way of combining two concept patterns above so that each could be matched against an individual, and the result of the two matches could be returned. The constructor **try** is intended to have this effect: (**try** M1 M2 ... Mn) matched against C always succeeds and returns the combined substitutions of all the successful matches. So the concept pattern we started out looking for is

(**try** (**fills** `price` _P)
    (**and** (**at-most** 0 (**known** `price`)) (**all** `price` (**and** (**min** _X) (**max** _Y)))))

or, better,

(**try** (**fills** `price` _P)
    (**and** (**at-most** 0 (**known** `price`)) (**all** `price` (**min** _X)))
    (**and** (**at-most** 0 (**known** `price`)) (**all** `price` (**max** _Y)))).

The second concept pattern does not require *both* _X and _Y to be known, in order for one of them to be returned. Matching either of the above concept patterns against `Stereo2` yields the substitution [_X $\mapsto$300 , _Y $\mapsto$2000].

Since we wish to display not just immediate components of objects, but also properties further down the "part-of" hierarchy, we also need to be able to match some object,

[2]In very recent CLASSIC literature, this notion is referred to as "full", i.e., no more fillers are allowed to be added to the role.

and then state additional constraints or matches on it. This is accomplished using the **as** construct:

(**fills** `maker` _M **as** `AMERICAN-COMPANY`)

matches the `maker` _M if it is an instance of `AMERICAN-COMPANY`, while

(**fills** `maker` _M **as**
(**and** `AMERICAN-COMPANY` (**fills** reliability _R)))

also looks for the reliability rating of the `maker`.

One final complication that needs to be considered concerns the behavior of **fills** matching for roles with multiple fillers. When matching

(**fills** `speaker` _S)

we might expect it to yield the substitution [_S $\mapsto$\{ `Speaker1`, `Speaker2` \}]. However, in this case we run into problems when trying to make sense of the concept pattern

(**fills** `speaker` _S **as** (**fills** `price` _P))

since we need to associate potentially different `prices` with each `speaker`. We therefore propose to return *sets of* matches in such cases[3]. Assuming `Speaker1` is known to have a `price` of 100 and `Speaker2` is known to have a `price` of 200, we would return the set: \{ [_S $\mapsto$`Speaker1`, _P $\mapsto$100], [_S $\mapsto$`Speaker2`, _P $\mapsto$200] \}.

We also introduce set-variables, which have the postfix -set, as in

(**fills** `speaker` _S-set),

which when matched against `Stereo2`, would yield the substitution [_S-set $\mapsto$\{`Speaker1`, `Speaker2` \}].

The collection of values will be useful when evaluating all values at once, for example, for comparing sets of values. Such variables are needed in any case in order to deal with constructors like **one-of**, which needs to be matched against patterns like

(**one-of** _I-set).

Finally, as with the design of CLASSIC and LOOM, we believe it will be very important to allow an "escape hatch" where one can express arbitrary conditions, which cannot

[3] This is like in Prolog, where successful goals may return one or more variable bindings.

be handled by the expressively limited DL. In this case the constructor will be **test-m**, which will take as its argument a user-provided function, which itself can take arguments, and will return a set of substitutions; if this set is empty, the match is taken to fail. In order to avoid using "global variables" to which **test-m** returns bindings, our **test-m** constructor also takes a variable as an argument which gets bound to the return value of the function.

For example, suppose we wanted to obtain the most expensive stereo component; if we had a function, called MostExpensiveComponent, which would take an individual and retrieve all the known fillers of component roles, and then find the one with the highest `price`, we could use the concept pattern as in the following pattern

(**test-m** _X MostExpensiveComponent Ind).

If there is any component with a `price`, this match will succeed and assuming Component1 has the highest `price`, the substitution will be [_X $\mapsto$Component1].

Functions can also be written to take sets as arguments. For example, we might have a function that takes a set of numbers as an argument and then adds all the numbers in the set, producing the sum. Given such a function, called SummedPrice, we might find the total `price` of the current `speakers` in a stereo system by matching against the following pattern:

(**and** (**fills** `speaker` _X-set)
    (**test-m** _Y SummedPrice X-set))

For thoroughness, we can also allow functions to be written that bind set variables and thus take the form (**test-m** _x-set <function> <args>).

6.2.1 Meta-Objects and Meta-Relations

In this subsection, we provide motivation for and examples of our support for using more than just the knowledge about the properties of an object. Sometimes, when deciding what information to present or to explain, it is useful to allow the user to access information *about* an object in the context of the knowledge base. For example, users may want to print the primitive concepts that are ancestors of a concept in the concept hierarchy. The description logic based system has calculated the primitives associated with every object, but we have no DL description that accesses just this information. In the base CLASSIC system, one could call an external system function to obtain just this information or one could "print" the object and obtain this information along with quite a bit of other information. Also, users may have presentation ideas for classes of objects that are "interesting", and would want to communicate this information without introducing "dummy" interesting classes into the domain model. Finally, as noted in the introduction, users may know some information about an object (e.g., it is a `SPEAKER`) and would only want to see *more specific* facts, if there are any.

In order to capture information about descriptions, we associate with each concept, and in fact every description, a corresponding *meta-individual* — one that has its own roles and attributes and can belong to meta-classes like `INTERESTING`. The meta-individual corresponding to description D will be referred to as **metaind**(D). We also introduce a set of built in meta-roles – roles that are filled with meta-individuals, whose fillers are calculated (in a lazy manner) by the system.

In a pattern, a meta-individual is accessed by placing the **meta** constructor in front of a description. For example, if users want to print the primitives on a concept, they should access the meta-individual associated with the concept, and use the built in meta-role `primitives`, as in the following pattern:

(**meta** (**fills** `primitives _X`)).

In other words, we can think of the system as storing the information about the primitives that are part of the definition of a description D by filling the meta-role `primitives` on the individual **metaind**(D) with the individuals corresponding to the primitive concepts.

Conceptually, meta-roles such as `primitives` or `subsumedBy` relate descriptions to descriptions, but we can interchangeably think of them as relating descriptions to the corresponding meta-individuals. (It is for this reason that the syntax of individual patterns indicates that a metaIndId is a concept pattern with only bound variables — these have corresponding meta-individuals.) This allows us to apply a uniform syntax at the meta-level, for example by using the pattern (**fills** `subsumedBy _X` **as** (**fills** ...)).

The user may also explicitly put meta-objects into classes, such as the INTERESTING class. One can then select only the primitives that are "interesting" using the following pattern:

(**meta** (**fills** primitives _X **as** INTERESTING)).

As a different example, the user may know that the maker role is always restricted to be a COMPANY, so he or she may be interested in seeing the restriction on maker *only if* that information is strictly more specific than COMPANY. The following pattern may be used:

(**all** maker _X **asmeta** (**fills** strictlySubsumedBy COMPANY)).

The **asmeta** construct is identical to the **as** construct except that the description following **asmeta** applies to the meta-individual associated with the variable instead of applying to the variable itself. This pattern uses another built in meta-role named strictlySubsumedBy. A meta-individual I is an instance of (**fills** strictlySubsumedBy Y) if and only if the description D_I of I is subsumed by Y and Y is not subsumed by D_I. Similar to strictlySubsumedBy, we introduce the meta-roles subsumedBy, subsumes, strictlySubsumes, instanceOf, and hasInstance. (The complete list of meta-roles appears in Figure 6.2.)

We provide built in roles that will help users to access aspects of objects that would be natural to print or explain. For example, CLASSIC knowledge base designers can associate comments with objects and, particularly on rules, we find it useful to print out the comments associated with the rule. For this, we use the pattern:

(**meta** (**fills** comment _X)).

Since the description logic based system calculates the parents of objects and children of concepts in the subsumption hierarchy, and an informative print function might print out both of these, we have included built in meta-roles parents and children. These can be used simply to match parent and child concepts of a concept by using the following pattern:

(**meta** (**try** (**fills** parents _X-set) (**fills** children _Y-set)))

We could also have included ancestors (i.e., all named concepts above an object in the generalization hierarchy) and `descendents` (i.e., all named concepts below a concept in the generalization hierarchy), but we can find these with the following patterns:

`ancestors`: (**meta** (**fills** `strictlySubsumedBy` _X **as** `NAMED-CONCEPT`))

`descendents`: (**meta** (**fills** `strictlySubsumes` _X **as** `NAMED-CONCEPT`))

The above patterns used the system defined meta-concepts called `NAMED-CONCEPT`, which along with `UNNAMED-CONCEPT`, reflects the system distinction between meta-individuals for descriptions that are named or unnamed by the user. For example, (**at-least** `4 r`) is subsumed by (**at-least** `3 r`), (**at-least** `2 r`), (**at-least** `1 r`), and (**at-least** `0 r`), as well as named concepts that subsume (**at-least** `4 r`). It is rare that a user would want to print out all the unnamed descriptions that subsume (**at-least** `4 r`). [4]

It is also useful to know what information has been explicitly "told" to the system versus what has been derived. To support this distinction, we provide a meta-role "told" which is filled with named concepts that are given as superconcepts of the description and one additional description containing other restrictions. Thus if a concept is defined as:

(**and** `A B` (**all** `r A`) (**all** `q B`)),

then the `told` role would be filled with (meta-inds for):

`A`, `B`, (**and** (**all** `r A`) (**all** `q B`)).

In some of our applications with very complicated objects, our users found it helpful to print only `told` information on objects. In order to find all the told information on an object, match the object against:

(**meta** (**fills** `told` _X))

Another typical use of the `told` role is to discover information that has been deduced,

[4]The notion of named concepts is important for some meta-roles, particularly those related to subsumption. Arguments to meta-roles may be a named object, a variable that is already bound, or an unbound variable. In the case where a subsumption meta-role takes an unbound variable, we only consider bindings that are named. This eliminates the problem that descriptions may subsume an infinite number of descriptions. For example, (**at-least** 2 r) subsumes any description of the form (**at-least** n r) where $n \geq 2$. This means that our previous definitions of ancestors and descendants can eliminate the use of `NAMED-CONCEPT`. For clarity though, we will continue to use `NAMED-CONCEPT` in our examples. This notion, and in general notions particular to meta-roles, is not covered in the Prolog specification in Section 6.3.

i.e., known but not told. For example, suppose the user wants to find the primitives that were deduced for an object. The following pattern could be used:

(**meta** (**and** (**fills** `primitives` _X)
(**notKnownToFill** `told` _X)))

Another slightly more complicated example involves finding the fillers of a role `p` that were deduced (and thus were not told). To do this, the user can use a pattern that matches the p-fillers, selects the restrictions in the told information, and then uses a test function called "notContaining" to make sure that the filler information was not contained in the told information. The function `notContaining` is assumed to take an expression and then check to see if another expression is a subpart of the larger expression. This might be as simple as stripping off any leading conjunctive constructors and then doing a straight syntactic match or it could break the larger expression into its atomic descriptions as defined in [90] and then see if the atomic descriptions in the smaller description are contained in the larger expression. The pattern below may be used for this matching.[5]

(**and** (**fills** `p` _X)
(**meta** (**fills** `told` _Y **as** (**and** `UNNAMED-CONCEPT`
(**test-m** _Z notContaining _Y '(**fills** `p` _X))))))

In CLASSIC, concepts may have rules associated with them, so we add a meta-role named `rule`, which is filled with meta-objects associated with rules. Then, if a user wants to print the rules on a concept, the appropriate pattern is:

(**meta** (**fills** `rule` _X)).

In addition, it is sometimes useful to access the left-hand side of the rule (which we define as the concept with which the rule is associated *and* any filter[6] that may be

[5] Note that substitutions must be applied to all variables in the function call before calling the function. This allows the function to take expressions like (**fills** `p` _X) where the value _X is determined previously in the rest of the pattern.

[6] CLASSIC has a notion of a filtered rule which allows a rule to have an additional CLASSIC expression that must be satisfied before the rule is fired.

defined on that rule). The right hand side of the rule is simply the consequent of the rule. Meta-roles `lhs` and `rhs` are used for these notions. Then, in order to determine what information was added because of inherited rules, we can use the following pattern:

(**meta** (**fills** `strictlySubsumedBy` _X **as**
    (**and** `NAMED-CONCEPT`
        (**fills** `rule` _Y
            as (**fills** `rhs` _Z)))).

The user may also determine whether a particular expression is the result of a rule firing. For example, to ask if on the individual B, (**at-most** 1 r) came from the rule named A, one needs to determine if matching the following pattern against B succeeds or fails:

(**meta** (**fills** rule A **as**
    (**fills** rhs _W **as** (**fills** `subsumedBy` (**at-most** 1 r))))).

We have now introduced the base DL language and the extensions necessary to individually and declaratively access all of the information that CLASSIC prints and more. Next, we will present the grammar for the entire language.

6.2.2 Summary: a Syntax for Concept Patterns

We summarize the discussion so far by presenting the syntax of concept patterns for CLASSIC in Figures 6.1 and 6.2. Patterns that can be matched against concepts or descriptions are derived from a non-terminal concept pattern (**cp**), while those to be matched against individuals derive from an individual pattern (`ip`).

6.3 Matching Concept Patterns

As we have seen above, when matching a concept pattern P against an individual or a concept D (written as `match(P,D)`), we expect to obtain a set of substitutions for the variables in P which make the match "succeed". The question is what is the appropriate notion of success.

```
<cp> ::=        ConceptId
                | Var ⟨ asmeta <cp> ⟩
                | (all <rp> <cp>)
                | (fills <rp> <ip>)
                | (one-of <iset>)
                | (min <np>)
                | (max <np>)
                | (at-least <np> <rp>)
                | (at-most <np> <rp>)
                | (and <cp> +)
                | (try <cp> +)
                | (test <tp>)
                | (same-as (<attributeId>+) (<attributeId>+))
                | (test-m Var <TestID> <arg> *)
                | (test-m SetVar <TestID> <arg> *)
                | (meta <ip>)
<ip> ::=        IndId
                | MetaIndId
                | Var ⟨ as <cp> ⟩
                | (notKnownToFill <roleId> <IndId>+)
                | (notKnownToFill <roleId> Var+) ;; where Var is already bound
                | <cp>
                | (closed <roleId>)
<iset> ::=      IndId+ | SetVar
<cset> ::=      ConceptId+ | SetVar
<np> ::=        Integer | Var ⟨ as ConceptDescr ⟩
<rp> ::=        roleId | known roleId | Var
<tp> ::=        TestId | Var
MetaIndId ::=   <cp> with no unbound variables
roleId ::=      <user-defined-role> | <meta-role> ;; meta-roles only valid on metainds
```

Figure 6.1: Concept and Individual Patterns for CLASSIC

```
meta-role ::=
          told |                          ;; filled with named concepts and one
                                          ;; conjunctive description of other
                                          ;; restrictions
          comment | primitives |
          rule |                          ;; available on concept meta-objects
          lhs | rhs |                     ;; available on rule meta-objects
            ;; the meta-roles below take meta individuals associated
            ;; with descriptions as arguments, rather than just named concepts
          subsumes | strictlySubsumes | subsumedBy | strictlySubsumedBy |
            ;; the meta-roles below take sets of descriptions as arguments
          subsumesMember | strictlySubsumesMember |
          subsumedByMember | strictlySubsumedByMember
          parents | children |
          instanceOf | hasInstance |
```

Figure 6.2: Meta-Roles for CLASSIC

A purely "syntactic" approach considers descriptions as variable-free terms, and attempts to treat concept pattern matching as subterm-matching using a version of unification. The main drawback of this approach is that the result of matching a pattern depends on the syntactic presentation of the description:

- **match**((**all** child _X), (**all** child `NOTHING`)) succeeds with substitution [_X $\mapsto$`NOTHING`].
- **match**((**all** child _X), (**at-most** 0 child)) fails

although the concepts (**all** child `NOTHING`) and (**at-most** 0 child) are semantically equivalent.

This problem can be addressed by first mapping semantically equivalent concepts to the same normal form (one which makes all relevant information explicit), and then matching against the normal form. In the above case the normal form would be

(**and** (**all** child `NOTHING`) (**at-most** 0 child))

so matching either pattern will succeed.

A final alternative would have been to define matching in terms of subsumption: **match**(P,D) succeeds with substitution σ if and only if σ(P) subsumes D. The problem with this is that **match**((**all** speaker _X) , (**all** speaker AMERICAN-MADE)) would succeed for any substitution which assigns to _X a value more general than AMERICAN--MADE (rather than what we would have expected, namely that _X = AMERICAN-MADE). And in fact,

match((**all** speaker _X) , (**fills** price 2000))

would also succeed for _X = THING.

Another issue in semantic matching arises from the problem of resolving multiple matches to the same variable. Consider the following match statement:

match((**and** (**all** tweeter _X) (**all** woofer _X)),
(**and** (**all** tweeter (**one-of** Ind1 Ind2)) (**all** woofer (**one-of** Ind2 Ind3))))

In order to come up with an "optimal" match for _X, the algorithm would need to find the least common subsumer of (**one-of** Ind1 Ind2) and (**one-of** Ind2 Ind3) resulting in the substitution [_X $\mapsto$(**one-of** Ind1 Ind2 Ind3)].

In order to avoid having to define and implement a very complicated semantic notion of matching, without recognizable benefits, we have chosen to base our approach on normal form matching. The normalization algorithm for CLASSIC is described in [20].

As far as individuals are concerned, we assume that every individual i starts out with an associated description descr(i) containing all the descriptions asserted or derived about that individual. (In CLASSIC, this is obtained using cl-ind-expr.) This is augmented as follows:

- For every filler b of role r, descr(i) contains both (**fills** r b) and (**fills** (**known** r) b).
- If r is closed on i and has an entire filler set i_1, i_2, ... i_n, the descriptor contains (**closed** r) and (**all** r (**one-of** i_1 i_2 ... i_n)).
- For every role r whose entire current filler set is: i_1, i_2, ... i_n, the descriptor contains (**at-least** n (**known** r)), (**at-most** n (**known** r)), (**all** (**known** r) (**one-of** i_1 i_2 ... i_n))

- For every role `r` with value restriction `C`, the descriptor contains (**all** (**known** `r`) `C`).

We present a Prolog-style specification of the matching algorithm, which works recursively on the structure of both the description and the concept pattern.[7] To describe matching, we will use a term-like syntax that puts the constructor first followed by its arguments in parenthesis. So, (**all** r C) will be written all(r,C). This description includes some simple functions including descriptor (described above) and metaind, instanceOf, getFillers, extend, closed?, and notmembersof. Metaind(c,ci) gets the metaindividual associated with c and binds it to ci. InstanceOf(i,c) succeeds if and only if `i` $\rightarrow$ `C`. GetFillers(i,r,iset) gets the current set of of `r`-fillers on i and binds them to iset. Extend(s0,s1,s2) takes two substitutions s0 and s1 (which may be empty) and returns substitution s2 which is the union of substitutions s0 and s1 with one exception: if s1 and s2 have different mappings for the same variable, then extend fails. Closed?(i,r) succeeds if role r is closed on individual i. (In CLASSIC, this is identical to cl-ind-closed-role.) NotMembersOf(jset,iset) fails if any member of jset is a member of iset and succeeds otherwise.

The match statement of the form:

match(c, _x, <_x,c>).

should be read, "the concept c matches the concept pattern of the form _x and returns the substitution <_x, c>)." This match always succeeds.

The match statement of the form:

match(c , asmeta(_x,mc) , s) :-
    metaind(c,ci), match(ci,mc,s0), extend(<_x,c> , s0, s).

[7]Recall that in Prolog, for any case for which there is no clause specified, the attempted match fails – e.g., match(all(p,c),fills(p,X),_) fails. This is used to deal with **and** in the normal form. Additionally, matches fail if any statement on the right of :- fails. For example, consider match(all(r,C), and(all(r,_X),atleast(2,p)) ,s) :- match(all(r,C), all(r, _x), s0), match(all(r,C), atleast(2,p), s1), extend(s0, s1, s). This fails because match(all(r,C), atleast(2,p), s1) fails.

should be read, "the concept c matches the concept pattern of the form asmeta(_x,mc) and returns the substitution s. This match succeeds when the following statements succeed:

- ci is the metaindividual associated with c
- ci matches the concept pattern mc and returns the substitution s0
- s is the extension of the substitution <_x,c> and the substitution s0."

NOTATION:

c, d are pure classic descriptions

mc, md are concept patterns (with or without variables)

mi, mj are individual-patterns

_x, _y are variables

_x-set is a set variable

i, j are classic individuals

iset, jset is a set of classic individuals

r is a role id or (known roleId)

n is an integer

t is a classic defined test id

s is a substitution

DESCRIPTION MATCHING:

```
; match(<normalized descr>, <concept pattern>,      <resulting substit>) :-
        <conditions>.
match( c ,                 d ,                     ()) :- subsumes(d,c).
match( c ,                 _x ,                    <_x,c>).
match( c ,                 asmeta(_x,mc) ,         s) :-
        metaind(c,ci), match(ci,mc,s0), extend(<_x,c> , s0, s).
match( c ,                 meta(mc) ,              s) :-
        metaind(c,ci), match(ci,mc,s).
match( c ,                 and(mc,md) ,            s) :-
```

```
        match(c,mc,s0), match(c,md,s1), extend(s0,s1,s).
;; in the following 3 match rules for and(c,d), only apply the second
;; and third rules, if the first rule failed.
match( and(c,d),          mc,                s) :-
        match(c,mc,s0), match(d,mc,s1), extend(s0,s1,s).
match( and(c,d),          mc,                s) :-
        match(c,mc,s).
match( and(c,d),          mc,                s) :-
        match(d,mc,s).
;In the rules for try(mc,md), once one try rule succeeds,
;do not try any additional try(mc,md) rules.
match( c,                 try(mc,md),        s) :-
        match(c,mc,s0), match(c,md,s1), extend(s0,s1,s).
match( c,                 try(mc,md),        s) :- match(c,mc,s) .
match( c,                 try(mc,md),        s) :- match(c,md,s) .
match( c,                 try(mc,md),        ()).
match( all(p,c) ,         all(p,mc),         s) :-
        match(c,mc,s).
match( all(p,c) ,         all(_x,d),         <_x,p>) :- subsumes(d,c).
match( all(p,c) ,         all(_x,mc),        s) :-
        match(c,mc,s0), extend(<_x,p>,s0,s).
match( c ,                all(known(r),mc),  s) :-
        on?(indprocessing),
        getFillers(c,known(r),iset),
        for all i in iset, match(i,mc,s0), extend(s,s0,s).
match( fills(p,iset),     fills(p,_x),       <_x,i>) :-
        choose i from iset.
match( fills(p,iset),     fills(p,as(_x,mj),  s) :-
        choose i in iset
        match(i,mj,s0), extend(s0,<_x,i>,s).
```

```
match( fills(p, iset),        fills(_x,i),                <_x,p>) :-
        memberof(i,iset).
match( fills(p, iset),        fills(_x,_y),               [<_x,p>,<_y,i>]) :-
        choose i in iset.
match( fills(p,iset),         fills(_x,as(_y,mj)),        s) :-
        choose i in iset
        match(i,mj,s0), extend([<_x,p>,<_y,i>],s0,s).
match( fills(p,iset),         fills(p,_xset),             <_xset,iset>).
match( fills(p,iset),         fills(p,as(_xset,testM(_xset,f,args))), <_xset,v>):-
        f(iset,args,v).
match( fills(p,iset),         fills(_y,_xset),            [<_xset,iset>,<_y,p>]).
match(fills(p,iset),          fills(_y,as(_xset,testM(_xset,f,args)), [<_xset,v>,<_y,p>]):-
        f(iset,args,v).
match( atleast(n,p),          atleast(_x,p),              <_x,n>).
match( atleast(n,p),          atleast(as(_x,d),p),        <_x,n>) :-
        instanceOf(n,d).
match( atleast(n,p),          atleast(m,_y),              <_y,p>):- m ≤ n.
match( atleast(n,p),          atleast(_x,_y),             [<_x,n>, <_y,p>]).
match( atleast(n,p),          atleast(as(_x,d),_y),       [<_x,n>, <_y,p>]) :-
        instanceOf(n,d).
match( atmost(n,p),           atmost(_x,p),               <_x,n>).
match( atmost(n,p),           atmost(as(_x,d),p),         <_x,n>) :-
        instanceOf(n,d).
match( atmost(n,p),           atmost(m,_y),               <_y,p>):- n ≤ m.
match( atmost(n,p),           atmost(_x,_y),              [<_x,n>, <_y,p>]).
match( atmost(n,p),           atmost(as(_x,d),_y),        [<_x,n>, <_y,p>]) :-
        instanceOf(n,d).
match( oneof(iset),           oneof(_x-set),              <_x-set,iset>).
match( oneof(nil),            mc, ()).
match( oneof(iset),           mc,                         s):-
```

```
          on?(indprocessing),
          for all i in iset, match(i,mc,s0), extend(s,s0,s).
match( min(n),            min(_x),              <_x,n>).
match( min(n),            min(as(_x,d)),        <_x,n>) :-
          instanceOf(n,d).
match( max(n),            max(_x),              <_x,n>).
match( max(n),            max(as(_x,d)),        <_x,n>) :-
          instanceOf(n,d).
match( test(t),           test(_x),             <_x,t>).
match( c,                 test-m(_x,f,args),    <_x,v>):-
          f(c,args,v).
```

MATCHING FOR INDIVIDUALS:

```
match( i,  i,                  ()).
match( i,  d,                  ()) :- instanceOf(i,d).
match( i,  _x,                 <_x,i>).
match( i,  as(_x,mj),          s):-
          assert-on(indprocessing)
          match(i,mj,s0), extend(<_x,i>,s0,s),
          assert-off(indprocessing).
match( i,  mc,                 s):-
          assert-on(indprocessing),
          descriptor(i,d), match(d,mc,s),
          assert-off(indprocessing).
match( i,  closed(r),          ()):-
          closed?(i,r).
match( i,  notknowntofill(r,jset),()):-
          getFillers(i,r,iset), notMembersOf(jset,iset).
```

extend(A,B,C) : C = A union B except if <_x,k> is in A and <_x,l> is in B yet k is not equal to l, in which case, extend fails.

Note that this specification reflects the semantics of CLASSIC, which does not take properties of individuals into account during concept classification. Thus, there are some rules in the concept matching section that are "guarded" by first checking to see if individual processing is on, and thus only fire when matching has been called from within individual processing.

6.4 Utilizing Concept Patterns

The obvious use for concept patterns is for filtering object displays. The user could write a pattern any time he/she would like to see an object and the system can match the object description against the pattern in order to determine what should be presented. Since it is common for users to use the same concept pattern for all instances of a concept, we have chosen to allow users to associate a concept pattern with each concept in the hierarchy. (This concept pattern can be considered to be the filler of a `print-filter` meta-role.) Then, any time a concept is about to be printed, its concept pattern is applied to filter out irrelevant or uninteresting information. If the match fails then nothing is printed out for the object. In CLASSIC, we in fact allowed two distinct patterns to be associated with classes – one pattern for what is interesting to print and another for what is interesting to explain.

In the case of individuals, it makes sense to apply the filters of *all* the concepts of which this individual is an instance. This forces us to consider the issue of combining concept patterns. A natural solution is to combine two concept patterns P1 and P2 that came from different parents by simply using the concept pattern (**try** P1 P2). Thus, all information that has been specified to be of interest in parent classes will be of interest on this object. Another issue for pattern combination is the treatment of variables occurring in both P1 and P2. When different inherited patterns make multiple use of variables with the same name, those variables can be thought of as being rewritten meaning that variables in different patterns will be different.

In this design, we have chosen to use **try** as the constructor to combine inherited patterns. In a more sophisticated design, we might consider letting users specify the semantics of combinations, for example allowing the user to decide when to use **and** as the combining constructor instead of **try**.

Concept patterns are inherited by all instances of concepts and by all subclasses of concepts. One way of exploiting this fact is to associate a standard concept pattern with a very general concept and then have all objects below this concept use the pattern. For example, we could make a general rule that we always want to see the comment field on every concept or individual, by associating the pattern

(**meta** (**fills** comment _X))

with the top concept in the knowledge base. This would then be inherited by all other concepts in the KB. We might also associate with `CONSUMER-THING` that the `price` is interesting, thus we would associate the pattern

(**fills** `price` _Y).

A specification that may be specific to the home theater domain is a specification of what is interesting on home theater stereo systems. Users may want to see the `price` ranges if no `price` is known, and also, it may be that certain components – `receivers`, `speakers`, and `tvs` – may be of interest. We may want specific `price` and filler information, and in the case of `tvs`, we may want to know the `diagonal` and its required minimum. The following pattern can be associated with `HOME-THEATER-STEREO-SYSTEM`:

(**try** (**and** (**at-most** 0 (**known** `price`)) (**all** `price` (**min** $\_X_1$)))
    (**and** (**at-most** 0 (**known** `price`)) (**all** `price` (**max** $\_X_2$)))
    (**fills** `receiver` $\_X_3$ **as** (**try** (**fills** `price` $\_X_4$) (**fills** `maker` X_5)))
    (**fills** `speaker` $\_X_6$ **as** (**try** (**fills** `price` $\_X_7$) (**fills** `maker` X_8)))
    (**fills** `tv` $\_X_9$ **as** (**try** (**fills** `price` $\_X_{10}$) (**fills** `maker` X_{11})
        (**all** `diagonal` (**min** $\_X_{12}$))
        (**fills** `diagonal` $\_X_{13}$)))

Additionally, one problem we posed in our introduction was to show the price range for speakers in a stereo system if the price range is more specific than the price range of "normal" `HIGH-QUAL-SYSTEMS`, i.e., if the price range is more specific than >300.

We could use the following pattern

```
(all speaker (try (fills price _P)
                  (and (at-most 0 (known price))
                       (all price (min _X as (min 301))))
                  (and (at-most 0 (known price))
                       (all price (max _Y as (max MAXINT-1))))))
```

6.5 Evolution of the Proposal

For some historical perspective, this proposal came out of a generalization of our object and explanation pruning implementation in CLASSIC. We first implemented pruning variables with default settings chosen from our knowledge of patterns of CLASSIC use. These settings proved quite useful for domain independent information within applications but remained limited in their ability to handle domain specific information. We subsequently implemented a simpler design than what we just presented for meta descriptions of interestingness. That implementation is sufficient for encoding the required meta-information for use in our home theater system configurator [91]. In fact, using that implementation, we shrunk output by approximately an order of magnitude. That encoding of meta information uses a number of procedural filters for determining interestingness. We wanted to minimize the use of procedural encodings of meta information thus bringing us to our current design.

Our implemented version, for example, allowed us to state that a value restriction on a role path was interesting (such as the value restriction on the `manufacturer` role on `CONSUMER-PRODUCTs`) and it allowed a procedural filter to be applied to that restriction (such as a function checking for descriptions that are more specific than `COMPANY`). Now, using meta roles, we can say this declaratively:

(**all** `manufacturer` _X **asmeta** (**fills** `strictlySubsumedBy COMPANY`)).

Also, the implemented language allowed us to state that fillers of any role (or rolepath) were interesting, but because we did not have the **as** construct, we could not declaratively access properties of those fillers. In our stereo application, we wrote a parts list function that gathered all of the fillers of any of the component roles and then found their prices. Using our current design, we can do this simply by writing the pattern:

(**fills** `component` _X **as** (**fills** `price` _Y)).

Additional work has been done on the area of filtering outside the scope of this thesis. Some of this work concerning algorithms and matching against a structural normal form is presented in [19].

There are many extensions that could be made to this work. As with CLASSIC, we can analyze application's use of test matches, and build up a library of parameterized **test-m** functions, or possibly even include them in the language.

For example, our current proposal does not include several role constructors, such as **inverse**, which would have been useful in more complicated examples. For example, a limited form of inverse – one that only returns named concepts – would allow us to find out if a description D is *not* told but inferred by a propagation:

```
(meta (and (fills subsumedBy D )
           (fills told _X as (and UNNAMED-CONCEPT)
                                  (notKnownToFill subsumedBy D))
           (fills (inverse _r) _Y
                  as (all _r D))))
```

It might also be useful to see if **test-m**s were being used where they were not needed. For example, one function we found useful in our home theater application was "most-specific". One may be tempted to write a function that takes a set of concepts and returns a set including only the leaf concepts of the input set. Then, for example, if one is printing primitives, one could use the pattern:

(**meta** (**fills** prim _Xset **as** (**test-m** _Yset mostspecific)))

It is possible to encode the same information without using a test function:

```
(meta (and (fills prim _Xset)
           (fills prim _Y asmeta
                  (notKnownToFill strictlySubsumesMember _Xset))))
```

One of our goals with this effort was to minimize the number of such functions that were used as filters in our meta implementation for CLASSIC, and instead be able to express the same semantics declaratively, using only the core meta-language, which would not include any procedural notions such as **test-m**.

6.6 Summary

Because of the need for pruning presentations and explanations of complex objects, we investigated pruning methods for Description Logics. We presented a pruning solution by introducing a meta language which extends the base description logic with variables, epistemic operators, and a user extensible constructor. This more powerful language may be used to write patterns that are used to match against descriptions to determine "interesting" aspects of objects at print or explanation time. We presented the syntax of the language, and the matching algorithm. We have shown how concept patterns may be utilized to individually display objects, to filter all subclasses and instances of a concept by associating patterns with concepts, and to organize meta-individuals.

Chapter 7

Related Work

In this chapter, we will review some of the major areas where work has been centered around explaining deductive systems. Our work is the first major effort in the description logic community with the goal of providing an explanation capability for *all* deductions. Thus, in order to find related work we expand our literature search beyond description logics to other areas of computing where complicated deductions are performed and where explanation has been studied. We will describe some explanation efforts in the areas of expert systems, theorem proving, logic programming, other frame-oriented knowledge representation and reasoning systems, and natural language. Finally, of course, we will mention other efforts in the description logic community.

7.1 Explanation in Expert Systems

Rule-based expert systems have been very successful in applications of AI, and from the beginning, their designers and users have noted the need for explanations of their recommendations. Originally, explanations took the form of a trace of rule firings. Today, many commercial expert system tools include this kind of functionality. One of the earliest systems with a serious explanation component was MYCIN [111]. Simply put, MYCIN explanations were a a natural language paraphrase of an execution trace of rules. MYCIN also allowed questions of a general nature about the corpus of rules such as:

Why do you ask the age of the patient?

Is there a rule that discusses streptococcal morphology?

Do you ever conclude that gram positive cocci are contaminants?

Expert systems implemented in Prolog also can provide a relatively simple approach to explanation through the use of meta-programming. For example, Sterling and Shapiro [113] (pps. 314-317) provide the simple example below. Given

type(dish1, bread)

size(dish1, small)

explain

how(place_in_oven(dish1,top))?

The answer is:

place_in_oven(dish1,top) is proved using the rule

IF pastry(dish1) and size(dish1,small)

THEN place_in_oven(dish1,top)

pastry(dish1) is proved using the rule

IF type(dish1,bread)

THEN pastry(dish1)

type(dish1,bread) is a fact in the database.

size(dish1,small) is a fact in the database.

Sometimes, this kind of explanation may be adequate for simple applications, but in situations where complicated reasoning is performed or where there is a large amount of information, explanations can grow to overwhelming size quickly.[1]

As has been frequently pointed out([37, 35, 117], for example), paraphrasing an execution trace often leads to insufficient explanations. For example, a paraphrase of rules will not include general strategies or justifications for conjuncts of a rule. Swartout

[1]This work and others, for example [121] Section 4.3, discuss ways to use a meta-interpreter program to help filter explanations. This will help with the size of the explanation but will not change the fact that the explanation is just presenting a filtered trace of rule firings.

and Moore distinguish three categories of problems with such simple approaches: inadequacy of low level rules (which solve small, local problems but do not give insight into more general strategies), failure to distinguish roles of knowledge (which means control clauses have the same importance as critical domain level decisions), and insufficient knowledge (thus justifications of compiled rules may never be available).

Swartout and Moore identify two themes in "second generation" expert systems that support better explanations. The first is enhanced knowledge representation and the second is improved techniques for generation of explanations including natural language presentation. In this thesis, we have concentrated on the first theme, so we will point out some relevant examples of the first theme. Clancey's work in NEOMYCIN addressed abstraction for control knowledge. The system includes metarules such as "explore-and-refine" which supports explanations at a higher level and provides representations of more abstract problem-solving strategies. Thus, if NEOMYCIN is attempting to refine a disorder, it can use the following knowledge:

<Domain Knowledge>

Infection

causal-subtypes (meningitis bacteremia ...)

Meningitis

causal-subtypes (bacterial viral ...)

$<$Abstract Control Knowledge$>$

Task: EXPLORE-AND-REFINE

Argument: CURRENT-HYPOTHESIS

Metarule001

If the hypothesis being focused upon

has a child

that has not been pursued,

THEN pursue that child.

(IF (AND (CURRENT-ARGUMENT $CURFOCUS)

```
        (CHILDOF $CURFOCUS $CHILD)
        (THNOT (PURSUED $CHILD)))
    (NEXTACTION (PURSUE-HYPOTHESIS $CHILD)))
```

With this kind of abstract control knowledge, NEOMYCIN can separate domain knowledge from problem solving knowledge and can answer questions about general strategies. So, if a scenario existed where a patient is presumed to have meningitis, using this rule, NEOMYCIN will attempt to refine the hypothesis to either bacterial or viral meningitis. If asked why, it can use Metarule001 to state that it is trying to refine the current hypothesis when it is pursuing the child bacterial meningitis. Thus, in Swartout and Moore's terms, it may handle the problem of inadequacy of low level rules with abstraction, but it still may not distinguish different roles of knowledge and it still may have insufficient knowledge in terms of justifications or background terminology.

The work beginning with XPLAIN [115] and continued in the Explainable Expert System (EES) [96] addresses the last two issues by containing a terminology and capturing design information. There is definitional knowledge about the domain and some factual information about a particular event (both included in XPLAIN as factual knowledge) as well as problem solving knowledge such as "In adjusting the drug dosage recommendation, check for factors which can cause dangerous conditions that also can be caused by administering the drug". In EES, there is additional knowledge in the form of "tradeoffs" - dimensions for measuring problem solving methods, "preferences" - context-sensitive rules for guiding choices, and "terminology" - a (DL) description of the domain. The system records a derivation of the low-level procedures from the (declaratively stated) domain principles. In the framework of EES (presented in [96]), a program writer sits between the knowledge base and a generated design history and executable expert system. When an explanation is requested, the design history can be used to help explain the rationale behind the part of the expert system that was executed. All of this knowledge allows much deeper understanding of what the system is doing and what the user is asking about. This approach addresses the issues of insufficient knowledge with the terminology knowledge base and the declarative statements of problem solving strategies which can then support justifications of the compiled expert system.

EES has plans such as:

```
(define-plan diagnose-component
  :capability (DIAGNOSE (obj (c is (inst-of COMPONENT))))
  :method
    (let
     ((actual-symptoms
      (loop for each symptom in (POTENTIAL-SYMPTOM c)
         when
            (DETERMINE-WHETHER-DESCRIBED (obj c) (by symptom))
         collect symptom)))
      (FIND-CAUSES (obj actual-symptoms) (of c))))
```

It applies this plan to a goal that the program is trying to achieve by translating the goal into a DL description and then checking if the capability of the plan subsumes the goal. If no subsumption relationship holds, EES will try to reformulate the goal and then check to see if any appropriate subsumption relationships hold that would allow one of the plans to be used. Reformulating and applying transformations is much of the reasoning contribution of this architecture and thus, this is one of the most important things to explain.

This kind of architecture is invaluable for explaining how to use the problem solving strategies. What it does not address, however, is explanations of the core built-in reasoning system.[2] Our work's focus is on explaining the inferences that are the result of calculating the deductive closure of the facts given to the knowledge base. In the setting of EES, that would translate to explaining why a capability of a plan subsumes a goal or why particular objects are instances of descriptions, e.g., why something is an instance of COMPONENT (or a subclass of COMPONENT).

[2]In fairness to the EES architecture, the examples in their literature do not show that they exploit the underlying DL to do extensive complicated subsumption or instance checking although they do make a great deal of use of the DL definitional component. Thus, it is conceivable that explanations of subsumption and normalization may not be as critical in this work.

One additional approach to explanation in expert systems is worth discussion. Reconstructive explanation, as described by Wick and Thompson [126], takes the stance that an optimal way to solve a problem may differ greatly from an understandable way to explain the problem. Their reconstruction of a plausible explanation of some conclusion has some compelling advantages. The core reasoning system may be optimized for computation, the explanation reconstruction may be optimized for understandable explanations, and the reconstruction may be a simpler reasoner since it already knows the conclusion that it is reconstructing. While we believe this is an interesting alternative approach that could be applied to DLs, our experience indicates that even our "stable" systems continually evolve. Our preference is to limit our evolving reasoning systems to one (for derivation and annotated for explanation) rather than two modules (one for derivation and another for explanation). Our goal is to put an explanation structure in place that is easily updated by the same person who would be updating the efficient reasoning component.

7.2 Explanation in Theorem Proving Systems

Since we proposed to view explanations in DLs as proofs of subsumption "theorems", it is appropriate to review the work on theorem proving environments, such as Theo [44] and Ω-MKRP [69]) and on presenting explanations of deductions in such systems. In these systems, proofs, and possibly failed attempts at proofs, may be considered as explanations. As many point out, for example, Felty and Miller in [53], proofs may be voluminous and difficult to read by everyone other than expert designers of the systems. Therefore, complete proofs are typically discarded. Instead, the system attempts to pinpoint particularly useful portions of the complete proof. For example, Nqthm, the Boyer-Moore theorem prover has an inspection facility [72] for reading proof output. The working hypothesis is that understanding failed proof attempts is crucial to successful use of the theorem prover. Also critical to the use of the theorem prover is the notion of a "checkpoint", which can be thought of as a point in the proof *"at which some 'daring' transition happens, for example one can turn a provable goal into one that is not provable (such as generalization)"* [72]. When a proof fails, it is useful to examine such checkpoints. The inspection facility provides a way of accessing, manipulating, and pruning different kinds of checkpoints. An analog to this line of thought could be incorporated into our approach if, for example, we believed propagations were the key to understanding particular kinds of deductions. Although we do not currently implement it, we have discussed allowing users to "check off" the inferences they are interested in having explained. Our work in Chapter 6 can also be used to implement such an approach. We did not hard code it into the interface since it was not as clear in our deductive system as it may be in the Boyer-Moore system, that one kind of important inference was usually/always critically more important than other important inferences.

Another noteworthy approach is the χ -proof system [53]. This work views proofs as flexible tree-structured deductions. The system claims that trees are good presentations of the proofs and the system provides a mechanism for "lexicalizing" the trees into readable natural language. Proof trees are constructed by placing the theorem to be proved at the root and choosing an inference rule that can be used to prove the theorem and making that a child of the root. The process is repeated on the inference rule's premises. Once a successful tree is produced, the system provides a means for presenting the tree in an English-like form. For example, consider the following formula:

$$\forall x \forall y \forall z\ (R(x,y) \wedge R(y,z) \supset R(x,z)) \wedge \forall x \forall y (R(x,y) \supset R(y,x))$$
$$\supset \forall x (\exists y R(x,y) \supset R(x,x)).$$

Their (reasonably unreadable) tree for this proof is:

```
implies_r(and_l(forall_r(implies_r(exists_l(forall_l(forall_l(forall_l
      (forall_l(forall_l(positive(and_r(positive(axiom(R(a,b)),
                                    thin(axiom(R(b,a)))),
                          thin(axiom(R(a,b)))),
                    thin(axiom(R(a,a)))))))))))))
```

This can be presented as follows:

Assume $\forall x \forall y \forall z (R(x,y) \wedge R(y,z) \supset R(x,z)) \wedge \forall x \forall y (R(x,y) \supset R(y,x))$. Assume $\exists y R(a,y)$. Choose b such that $R(a,b)$. By modus ponens, we have $R(b,a)$.

> hence, $R(a,b) \wedge R(b,a)$. By modus ponens, we have $R(a,a)$. Since a was arbitrary, we have $\forall x(\exists y R(x,y) \supset R(x,x))$.

In addition, the work provides some ways of rewriting proofs that use techniques that are less obvious into techniques that are more explicit. For example, one may choose to prove A by an "indirect method", such as proof by contradiction: assuming ∼A, deduce a contradiction, and thus, prove A. If A could be proven without first assuming it's negation, the proof may be more understandable. The cited paper provides some techniques for revisions such as this.

The work is interesting since it builds proof trees as explanations and provides a way to present those trees. We can think of our complete explanation as such a proof tree. We believe that in our environment, complete proofs are most useful when the root "theorem" is very simple - such as our atomic descriptions. We also find that in our environment, it is most useful to see well chosen stand-alone pieces of a proof. One idea from the χ -proof system work that would be interesting to explore further is the idea of (logically well-founded) transformations of proof trees. This is most beneficial in a pre-calculated multi-step proof, so it is less applicable in a system such as ours where the user is typically choosing the next step incrementally. It is possible though that some domains or applications may find typical explanations to be using complete explanations.

This idea has been explored in related areas such as the work of Huang in presenting natural deduction proofs in PROVERB [67, 68]. There, he moves beyond work such as Edgar and Pelletier's [51] natural language presentation of such proofs by using a reconstructive approach to generate proofs logically equivalent to the original natural deduction proof, but at a higher level of abstraction (which he calls the assertion level). The recent work on this has progressed to a more sophisticated approach using planning to determine content, sentence structure, and presentation order. We have not tackled this issue.

Since the computation of a counter-example is a form of model construction, another area of theorem proving of interest to our work is the area of symbolic model checking as exemplified by the foundational work by Burch, Clarke, McMillan, Dill, and Hwang [33].

This work and subsequent work (for example, model checking work in LICS [133]) is interesting since the work typically represents relations and formulas symbolically and then provides efficient decision procedures for model checking. An interesting area of investigation would be determining if this work, particularly the work utilizing binary decision diagrams [28], can be used to make DL model generation and checking algorithms more efficient.

7.3 Explanation in Logic Programming

Although expert-system rules just resemble implications, pure Prolog and its datalog-cousins are in fact realizations of real logics, and hence can be seen as having a proof-theoretic characterization, which is usually based on unit-resolution. Horn-logic proofs can easily be presented as trees, where the right-hand side of a rule provides the children for the node corresponding to the left-hand side, with the additional need to show the variable substitutions that made unification possible.

For example, given `child(X) :- person(X), age(X,Y), Y<12.` , we get

```
               child(anna)
             /      |       \
person(anna)  age(anna,Y)   Y<12
                    |          |
              age(anna,5)   5<12
```

7.3.1 Prolog tracing

One way of explaining how a Prolog program computed a result is to take the procedural view of logic programs, adopt an “execution model” of the language, and then show “traces” of programs.

One view of Prolog execution is the box model proposed by Byrd [34], and subsequently described in many sources including [38]. In this model, a rule or program is thought of as being enclosed inside a box. There are four ports to the box: *call, exit, redo*, and *fail.* Calling is an initial invocation of the Prolog procedure. Exit is used when there is a successful return from the procedure. (Thus, subclauses have been satisfied and typically variables have been bound.) Redo is used when the system is backtracking because a subsequent goal has failed. Fail is used when the entire program fails.

```
       ---------------------------------------------
 call  |                                           | exit
-----> |    descendant(X,Y)                        |------>
       |       :- offspring(X,Y)                   |
       |                                           |
       |    descendant(X,Z)                        |
       |       :- offspring(X,Y)                   |
       |       :- descendant(Y,Z)                  |
 fail  |                                           | redo
<----  |                                           |<-----
       ---------------------------------------------
```

For example, consider the simple program:

p :- q,r.

q :- s.

q :- t.

r :- a, b.

s.

a.

?- p.

Using this program and the box model, the trace of execution can be displayed as follows (Nesting of boxes is shown in parentheses below.):

```
(1)  Call :  p
(2)  Call :  q
(3)  Call :  s
(3)  Exit :  s
(2)  Exit :  q
(4)  Call :  r
(5)  Call :  a
(5)  Exit :  a
(6)  Call :  b

(6)  Fail :  b
(5)  Redo :  a
(5)  Fail :  a
(4)  Fail :  r
(2)  Redo :  q
(3)  Redo :  s
(3)  Redo :  s

(7)  Call :  t

(7)  Fail :  t
(2)  Fail :  q
(1)  Fail :  p
```

From this, we can see everything that has been tried in the Prolog execution. It is worth noting that this shows how goals have failed in addition to how they have been proven. The failure justification does not appear in the proof trees.

This kind of tracing can however overwhelm the user with information. Also, its procedural nature makes it well suited to some programmers but ill-suited to many knowledge engineers.

The transparent Prolog machine (TPM) of Eisenstadt and Brayshaw [52] augments the standard tree model to include more status information of all the nodes and also provides a "long-distance view" of the program execution, including the Byrd model information in it. They have a very sophisticated graphical interface which seems to present the material well. The detail that they need to present is different than what we need to do since we are aiming our explanation to knowledge engineers and end-users. The Prolog environment work has a primary focus on debugging so more detail is presented, including information flow through the "parameters/arguments" of predicates. An analog in our system would be to include information about all the ways the DL attempted to classify an object lower in a generalization hierarchy but failed to do so.

7.3.2 Explanation in Deductive Databases

In many ways deductive databases have a problem that is quite similar to ours. These systems typically want to present a declarative view of the inferences that have been performed yet they may not be able to rely on an execution trace of the system since, for efficiency, the rules may have been optimized in a form that would not be understandable by users. Also, some evaluation strategies, e.g., bottom-up evaluation used in CORAL, do not enforce a fixed evaluation strategy thus making operational descriptions less informative. Two approaches to explanation in these systems are illustrated by the CORAL [6] and LDL [110] systems.

CORAL explanations can be presented as derivation trees made up of tree fragments. Given a rule r that used facts f1 ... fn to infer a fact f, a tree fragment has a root f and children f1 ... fn. So, for example, we may know (from [6]) that

```
anc_bf(X,Y) :- m_anc_bf(X), edge_bf(X,Y).
```

If a fact like `anc_bf(4,2)` was deduced, it would be the root of the tree fragment and the children would be `m_anc_bf(4)` and `edge_bf(4,2)`.

In comparison, our DL proof rules have a single conclusion in the denominator which would be the root f in CORAL tree fragments, a rule name (which is not present in the CORAL work), and a numerator with a number of components which would be the children in the CORAL tree fragment. Tree fragments with no children are base facts in CORAL, and would be told information in our system. A tree fragment can have a child node expanded with its underlying tree fragment. This can be thought of as asking one of the follow up questions in our explanation process. CORAL handles recursive derivation trees by letting the user conceptually point to a portion of a tree fragment that should be expanded. In this way, the user chooses how much of a derivation tree to display. Since we allow "complete explanation" questions, i.e., complete derivation tree generation, we needed additional control for recursion which is mentioned in Chapter 4. The CORAL explanation implementation logs rule instantiations during execution and uses that log to produce the tree fragments and derivation trees. (The logging is done when the program has full access to the completely instantiated rule, and additional relations are created to hold the derivation information of the particular predicate that is being justified.) An analog to this in our work would be to produce a log of all atomic justifications at run time. However, we do not create such a log since our annotations of CLASSIC structures on demand provide more efficient storage and a more convenient structure from which to generate explanations in CLASSIC. The CORAL work does not address the issues of explanation of missing information nor does it support reporting only "interesting" facts.

CORAL explanation also must address the issue of rewriting routines. For example, CORAL uses magic sets example which add "magic filters" to existing knowledge base rules as well as adding new magic rules (which are used to deduce the magic filters). Thus, an explanation that explained all the children or clauses in a rule would suddenly find a new child (corresponding to the magic filter) and the knowledge engineer would see additional rules used to deduce that clause. The CORAL approach prunes magic filters and does not explain them nor any rule that was used to deduce them.

The LDL work [110] assumes access to an "intended interpretation" which is what the user has in mind and which embodies the user's intuition. Given this interpretation and the interpretation that a program has generated, there may be information in the intended interpretation that is not in the program's interpretation or vice versa. This explanation work aims to explain "wrong facts", those facts in the program's interpretation but not in the user's interpretation, and "missing facts", those facts in the user's interpretation but not in the program's interpretation. A "wrong fact" submitted by the user is explained with a proof tree (which, for simplicity could be thought of as the derivation tree in the CORAL work.)[3] The user can then inspect the tree and see if any of the facts used in it are incorrect. Missing atoms occur when there is no rule in the program that can conclude the atom and the rule body can be satisfied in the program's interpretation. The user can ask why information is missing and the system displays the rules that could have been used to produce the missing information. The user then chooses the rule she expected to fire and the system can display the partial instantiation of that rule based on what is deducible in the current state of the knowledgebase. (If the rule could be fully instantiated consistently, then the rule would have fired.) The user can then identify what kept the rule from firing.

Again the proof tree as a justification is similar to our notion. The LDL approach to explaining missing information would be problematic if used in DLs for a few reasons. First, the assumption of an existing intended interpretation where the user is knowledgeable about what rules should have been used in a deduction may be appropriate for knowledgeable program debugging but will not hold when the intended audience for the explanation module contains naive users. Additionally, in description logics, there may be a very large number of different rules that could have deduced something and there are a number of different kinds of rules that could have been used, and thus a presentation of the possible ways something could have been deduced would be inappropriate.

The important similarity between these two pieces of work and ours is that the explanations focus on what was derived instead of looking at a procedural trace of

[3]Actually, their proof trees are more highly structured since they exploit stratification to handle negation and missing facts, but the extra structure is not needed for the purposes of this analysis.

exactly how something was derived.

7.4 Explanation in Frame KR&R Systems

Some KR systems built using Prolog incorporate some kind of trace of the Prolog rules into an explanation facility. The THEO [93] frame system, for example, has an "explain successful clauses" (prs!) functional that will print out the Prolog rules associated with each step of the explanation of a slot's value. For example, given that Tom is Meghan's mother and Joan is Tom's spouse, THEO can ask for an explanation of the filler of the mother slot on Meghan. THEO will produce the following, showing the Prolog rule and the supporting facts.

```
USER(): (prs! mother meghan)
(MOTHER MEGHAN JOAN)
  <-(PROLOG)-
    |(MOTHER ?PERSON ?MOTHER)
    |:-
    |(FATHER ?PERSON ?FATHER)
    |(SPOUSE ?FATHER ?MOTHER)
  (FATHER MEGHAN TOM)
  (SPOUSE TOM JOAN)
```

Cyc, as described in [43], provides a kind of theorem prover and relies on it for a form of explanation. It has a simple theorem prover that attempts to prove a theorem by finding a set of assertion sets from which the theorem would follow. If those assertion sets are in the knowledge base, the theorem is "proved". Then if "justify" is called after the theorem prover has found an assertion set, it will determine if any of the members of the assertion set is derived (rather than an assertion or theorem). If any member of the assertion set is derived, Cyc uses the "theorem-prover" to deduce it. This process continues until no remaining members of the assertion sets are derived. Finally, the tree of assertion sets is returned.

More generally, any frame system with a truth maintenance system (TMS) can use its dependency links as the basis of an explanation. The current version of Cyc[4] keeps a set of current arguments for believing why every assertion is true or false. Since assertions can have multiple arguments supporting their truth or non-truth, Cyc also has heuristics for determining which alternative argument to prefer. Thus, using arguments and heuristics to determine preferences, Cyc can choose which argument to present as an explanation for an assertion.

The CLASSIC implementation also has links that can be thought of as a TMS for supporting retractions. For efficiency reasons, it is however relatively coarse-grained ("object B needs to be revisited when object A is modified"). Therefore, it does not provide a sufficient basis for explanations. Also, changing the CLASSIC TMS so that it maintained a finer granularity would cost too much in terms of space, and thus this approach is not viable in CLASSIC.

7.5 Natural Language Explanation

There is a wealth of literature on natural language explanations. We will not attempt to survey the entire field, instead we will mention some work that could have fit into the previous sections, yet we have chosen to highlight it here because we perceive a major thrust of this work to be natural language presentation.

Early work by Dan Chester [36] provides a nice simple foundation for presenting natural deduction style proofs. His EXPOUND system takes a formal proof, and presents an English description of it. Essentially, he takes a formal proof, creates a graph of dependencies, uses the graph to determine clusters of related deductions and their appropriate order, and then uses the clusters and the ordering to construct an explanation that is reasonably easy to follow. He also provides some simple rules for eliminating proof lines. One interesting aspect of this work is its simple structured method for presenting proofs by contradiction. This work provides an alternative approach to explanation that is most appropriate when longer complete explanations are of use. One

[4]Information from an email exchange with author, Keith Goolsbey.

of his contributions is methods for ordering explanation fragments in order to provide a more coherent longer explanation. Our hypothesis is that people are looking for meaningful explanation fragments, so we are less concerned with a longer coherent explanation, and instead we focus on appropriate explanation fragments. We do however include a "complete explanation" capability and it would be interesting to see if his techniques could be used to make presentations of those complete explanations more useful.

Another classic piece of work that is worth noting is the work by Weiner [124]. This work, as Chester's did, focuses on the structure of explanations in an attempt to make them more understandable and manageable. This work is interesting because it takes a reasoning system (AMORD [42]), and augments it so that it is not only able to prove assertions, but so that it can also take into account certain features needed in generating acceptable explanations. It maintains a user's and a system's view so that it can take into account the user's current beliefs. It uses the system's entire knowledge to deduce things but modifies its output to eliminate information that is in the user's view. It provides some other nice structure for presenting explanations and making them more accessible. Some of his main areas of concern are composition, strategies for presenting explanations, counterfactual explanations, and user expectations.

It would be valuable to consider the ideas in this work when presenting user-tailored description logic explanations. This work however makes one assumption that we believe to be false in description logics. This work assumes that the user "knows" the deductive closure of the user knowledge (and thus information in that deductive closure should not be included in explanations). Our work with users provided evidence that DL users do not necessarily believe all statements in the deductive closure generated from their input knowledge. Thus, one of the underlying assumptions of this work would need to be modified before this work could be adapted to description logics. Also, this work is similar to Chester's in that it aims to produce more coherent longer explanations, while we focus on explanation fragments.

Some work in the area of deductive databases that is oriented towards providing intuitive answers might appear to be appropriate to consider because it also deals with incomplete understanding of a system's response. Gal [57] considered the problem of responding "cooperatively" to queries from users. One contribution of her thesis, was her work utilizing deductive database integrity constraints to attempt to give the user more useful answers. She wanted to return more informative answers, for example when asked "How many students failed CS400 last semester?" in a situation where CS400 was not offered last term. She wanted to point out that the course was not offered instead of answering that no students failed. She provided a parser for natural language, a response generator that translated the query into an equivalent form including appropriate integrity constraints and then formed an answer after investigating potential incorrect presuppositions (like CS400 was offered last term), and a synthesizer which provided a natural language response. Part of the response generator included a set of heuristics used to select pertinent information for inclusion in the cooperative answer.

Gaasterland, et.al. [54, 55, 56] expand on Gal's work in a number of ways still with the goal of providing more cooperative explanations. They, as most in the cooperative answering area, take inspiration from Grice [60] and aim to follow his maxims of cooperative conversation. One area they tackle is correcting invalid user presuppositions in a way that should not surprise the user and also should not overwhelm the user with information. Another facility they provide is additional user control over followup questions appropriate for particular answers. They also point out that users may sometimes need more or different information than exactly what they asked for. A solution they propose to this problem is "relaxation" which expands the scope of the original query by relaxing implicit constraints in the query.

This work has a focus on domain dependent explanation and does not focus on explanations of the reasoning process. Many of the user misconceptions could be viewed as a query failure based on the semantics of the domain (for example, a course was not offered or it can not have more than a certain number of students). While this work is also of interest in a complete explanation system, we have begun with a focus on explaining the underlying core domain independent reasoning system. It is an interesting area of future work to investigate how this approach to handling things like user misconceptions and incorrect presuppositions could be used to augment an explanation system for a DL reasoner.

7.6 Explanation in other DL systems

With a couple of exceptions, most implemented systems have not tackled the explanation problem.

NIKL [70, 104] was one of the first systems to have a limited "why not" capability. The explanation essentially points out a conjunct of the alleged subsumer concept that is not implied by the alleged subsumee. While this is a useful notion, it is only one small piece of the explanation functionality that we will present. Explanation did not progress much beyond this capability in NIKL.

LOOM [82] has more recently added some limited explanation capability. From discussions with the designer, we understand their plans include explanation capabilities similar to ours that can explain most inferences, however the implementation is still limited to explaining a few of the more common inferences in a manner similar to ours. They do however include a natural language presentation of explanations.

Another area of DL research that has been recently pursued is the work of Rousset, et. al. on KB verification, as presented in [105, 66]. Their major goal is knowledge-base validation/verification, thus the focus is only on explaining why descriptions are incoherent. Their work is grounded in proof rules and thus shares a similar cognitive approach to ours[5]. The DL language in their work and the one examined in this thesis overlap but are not contained in each other. Of course, explaining incoherencies is only one of the many deductions that we explain.

[5]The first author was a visiting researcher with our group at AT&T a few years ago, and we displayed our explanation implementation and discussed the issue of validating and verifying DL knowledge bases.

Chapter 8

Conclusion

Explanation capabilities are a critical component for making deductive systems usable by the general computer science audience. In this thesis, we have investigated four major issues related to explaining reasoning systems. We have described in detail how these problems present themselves in description logics and we have provided a solution for a class of description logics. We began by considering the issue of explaining a highly optimized, procedurally implemented system in a declarative manner. We provided a proof-theoretic foundation for viewing the inferences in description logics, yielding a declarative framework to our approach. Next, we considered the problem of long and complicated deduction chains. We began by noting that subsumption (denoted in this thesis by $\Longrightarrow$) and recognition (denoted as $\rightarrow$) are the core inferences in DLs, since all other inferences performed can be rewritten in these terms. Thus, it is these inferences that need to be investigated in detail. We provided a mechanism based on proof rules, and our notions of atomic descriptions and atomic justifications, which allows us to offer one step and/or complete explanations of subsumption and recognition judgements. Since incoherent concepts are subsumed by a special bottom concept `NOTHING`, incoherencies (and thus errors) can be explained as a subsumption relationship. Our work in these areas has been implemented and used in CLASSIC and in some of its applications.

Next, we considered the problem of explaining negative deductions. In description logics, the major negative deduction is non-subsumption. We presented a way to explain non-subsumption by exploiting the complete and simple process of determining structural recognition. Specifically, we explain non-subsumption by providing a counter-example individual that is an instance of the alleged subsumee but not an instance of the alleged subsumer concept. The main contribution was an algorithm that generates such a counter-example individual for CLASSIC, using the implemented CLASSIC system as much as possible. We believe our algorithm to be complete[1], even though CLASSIC is incomplete, and thus we can detect if a subsumption relationship was missed because of the incomplete implementation. (This algorithm has yet to be implemented for CLASSIC.)

Finally, we noted that objects in large applications in object-oriented representations can be quite complicated so even printing them can be cumbersome. Once one takes on the task of explaining object descriptions too, *pruning* becomes essential. We addressed this problem for description logics. We presented a pruning mechanism which relies on a declarative language and on meta individuals for capturing descriptions of what is interesting to print or explain. An early version of our design is implemented and has been used in applications.

We have shown by description, example, and implementation that this approach handles the needs for the CLASSIC Knowledge Representation System. We also claim that this approach is more generally applicable to Description Logics implemented in the normalize/compare paradigm. The purpose of the next section is to buttress this claim, and in the process also review in greater detail the technical concepts introduced by the thesis.

8.1 Generality

One way to support our claims of generality is to show that our framework can be extended in a reasonably straightforward fashion to explain a DL that is an extended version of CLASSIC. In particular, we will discuss what would need to be done if a new constructor were added to the language.

We will proceed through the four technical chapters, and give an overview of what would need to be added in order to explain the new inferences associated with that

[1]We provide relatively informal arguments of correctness for key procedures. Obviously our system is more complete than CLASSIC since it uses CLASSIC's reasoner and extends it by taking more properties of individuals into account.

constructor. In this discussion, we will distinguish the tasks done by the *explanation designer/implementor* from those required of the DL designer/implementor (marked with $^{\mathcal{D}}$), since the latter is groundwork that must be done in order to add the constructor and is not overhead caused by explanation implementation.

Since **some** is one of the most useful constructors not included in CLASSIC which has been discussed in much of the DL literature, we will use it as our example. The expression (**some** `component SPEAKER`) says that there is at least one filler of the `component` role that is an instance of the `SPEAKER` concept. The following are example subsumption inferences involving **some**, which one might reasonably expect to be implemented:

(**some** `component SPEAKER`) $\Longrightarrow$ (**at-least** 1 `component`)

and

(**and** (**some** `component SPEAKER`) (**some** `component HEADPHONES`)) $\Longrightarrow$
(**at-least** 2 `component`).

assuming that `SPEAKER`s and `HEADPHONES` are disjoint.

First, the syntax of the language of descriptions needs to be extended$^{\mathcal{D}}$ to include the production <descr> ::= (**some** <role-name> <descr>). Any extension to the base DL grammar will require reconsideration of atomic descriptions. In this case, the atomic description grammar, presented in Section 3.6.1, needs to be extended to include (**some** <role-name> <descr>). Note that since (**and** (**some** p C) (**some** p D)) is *not* logically equivalent to (**some** p (**and** C D)), the restriction is a <descr> rather than an <atomic-descr>, as in the case of **all**.

In order to implement subsumption, one must determine$^{\mathcal{D}}$ the subsumption rules to be incorporated into the system.[2] For **some**, one might choose a set of subsumption rules almost identical to those published in [12], shown in Figure 8.1. (For a better understanding of how to determine such rules see [14].)

An explanation designer will want to analyze the rules to determine the structural subsumption rules (i.e., those that deduce (**some** p C) $\Longrightarrow$ (**some** r D) for any p, r, C, and D) and the normalization rules.

[2]Note that the set of subsumption rules may be intentionally incomplete in order to maintain better computational properties.

SomeIsa $$\frac{\vdash C \Longrightarrow D}{\vdash (\textbf{some}\ p\ C) \Longrightarrow (\textbf{some}\ p\ D)}$$

SomeNil $$\vdash (\textbf{some}\ p\ \texttt{NOTHING}) \equiv \texttt{NOTHING}$$

SomeOne $$\vdash (\textbf{some}\ p\ \texttt{THING}) \equiv (\textbf{at-least}\ 1\ p)$$

OneAll $$\vdash (\textbf{and}\ (\textbf{at-least}\ 1\ p)\ (\textbf{all}\ p\ C)\ ...\ ) \Longrightarrow (\textbf{some}\ p\ C)$$

SomeAnd $$\vdash (\textbf{and}\ (\textbf{some}\ p\ C)\ (\textbf{all}\ p\ D)\ ...\ ) \Longrightarrow (\textbf{some}\ p\ (\textbf{and}\ C\ D))$$

AtmostSome $$\vdash (\textbf{and}\ (\textbf{some}\ p\ C)\ (\textbf{at-most}\ 1\ p)\ ...\ ) \Longrightarrow (\textbf{all}\ p\ C)$$

SomeAtleast2 $$\frac{\vdash (\textbf{and}\ C_i\ C_j) \Longrightarrow \texttt{NOTHING}}{\vdash (\textbf{and}\ (\textbf{some}\ p\ C_i)\ (\textbf{some}\ p\ C_j)\ ...) \Longrightarrow (\textbf{at-least}\ 2\ p)} \quad C_k \text{ are consistent}$$

Figure 8.1: Original Set of Subsumption Rules for the **some** constructor

SomeNil $$\frac{\vdash C \Longrightarrow (\textbf{some}\ p\ \texttt{NOTHING})}{\vdash C \Longrightarrow \texttt{NOTHING}}$$

SomeOne $$\frac{\vdash C \Longrightarrow (\textbf{some}\ p\ \texttt{THING})}{\vdash C \Longrightarrow (\textbf{at-least}\ 1\ p)}$$

SomeOne $$\frac{\vdash C \Longrightarrow (\textbf{at-least}\ 1\ p)}{\vdash C \Longrightarrow (\textbf{some}\ p\ \texttt{THING})}$$

OneAll $$\frac{\vdash C \Longrightarrow (\textbf{at-least}\ 1\ p) \quad \vdash C \Longrightarrow (\textbf{all}\ p\ C)}{\vdash C \Longrightarrow (\textbf{some}\ p\ C)}$$

Figure 8.2: Normalization Rules for the **some** constructor

In this case, the structural subsumption rule is:

SomeIsa $$\frac{\vdash C \Longrightarrow D}{\vdash (\textbf{some}\ p\ C) \Longrightarrow (\textbf{some}\ p\ D)}$$

The normalization rules, rewritten in our format, appear in Figure 8.2.

Once the DL implementation is extended$^{\mathcal{D}}$ to perform these inferences, it is straight forward to augment the explanation subsystem. First, we need to determine those new inferences that will be explained. For example, the original form of the SomeNil rule would be used to explain both why (**some** p `NOTHING`) $\Longrightarrow$ `NOTHING` *and* why `NOTHING` $\Longrightarrow$ (**some** p `NOTHING`). We would choose not to explain the second inference since we already have a rule that explains why `NOTHING` is subsumed by any arbitrary concept (since `NOTHING` is at the bottom of the hierarchy). Similarly, one needs to determine if any new inferences should be added that will facilitate explanations. The best examples of this are in error situations, so we will postpone this discussion until our error discussion.

Code needs to be written to augment the explanation data structure with the appropriate inference name and arguments whenever some new inference (deemed worthy of explanation) is used in a deduction.

In order to handle follow-up questions, we must look at the numerator of any new inference rule and verify that only subsumption or recognition judgements are used there. If this is the case, then there is no additional work to be done to make sure automatic follow-up generation is handled correctly. In the case of concept rules for **some** there is no additional work to do.

Having completed subsumption, we turn to individual recognition. Once again, the DL implementor needs to consider$^{\mathcal{D}}$ the kinds of deductions that result from the presence of **some**. The obvious structural recognition rule is

$$\textsf{SomeInstance}\quad \frac{\vdash I_1 \longrightarrow \ (\textbf{fills}\ r\ I_2)\ \ \vdash I_2 \longrightarrow \ D}{\vdash I_1 \longrightarrow \ (\textbf{some}\ p\ D)}$$

One also needs to consider$^{\mathcal{D}}$ the closed world reasoning aspects. For example, I is an instance of (**and** (**some** p C) (**fills** p Iset)), with role p closed, and all but one of the fillers of p are definitely not instances of C, then the last individual must be an instance of C. Thus, the DL needs a propagation rule

$$\textsf{SomeProp}\quad \frac{\vdash I \longrightarrow (\textbf{some}\ p\ C)\ \ \vdash I \longrightarrow (\textbf{fills}\ p\ Iset)\ \ \vdash I \longrightarrow (\textbf{at-most}\ n\ p)}{\vdash Ij \longrightarrow \ C}\quad \text{All i in Iset except } I_j \text{ are definitely not in C, n = |Iset|}$$

Once again, the explanation subsystem code needs to be augmented to update the explanation structures when these new rules are used.

Error situations deserve special attention. For example, consider the case of an individual `I` which is an instance of (**some** r C) and which has closed role `r` with fillers {Iset}. If all members of {Iset} are provably *not* instances of `C` then `I` is inconsistent. This problem would in fact already be caught by the current design, because the SomeProp rule adds `C` onto the last element of `Iset`, which leads to a contradiction on that individual. However this multiple-step deduction is convoluted and arbitrary on the choice of the "last element of `Iset`". Therefore the explanation designer might propose an additional (redundant) rule that would allow this to be explained in a single step:

SomeFillersConflict $\dfrac{\vdash I \longrightarrow (\textbf{some}\ p\ C) \quad \vdash\ I \longrightarrow (\textbf{fills}\ p\ Iset)}{\vdash I \longrightarrow\ \texttt{NOTHING}}$ All members of Iset are provably not instances of C

If new error rules are introduced as a result of adding the new constructor, they need to be used to update the explanation structures when they are used. In the worst case, the explanation designer needs to add the new simpler rules to the underlying DL and then appropriately augment explanation structures when they are used. structures when the rules are being followed. In addition, the special error follow-up function discussed in Section 4.4 needs to be augmented appropriately. In the case of the SomeFillersConflict rule, follow-up questions would concern how the individual `I` obtained its **some** restriction on p and how it obtained its p fillers.

In order to explain non-subsumption, the GetInd and Contradict algorithms discussed in Section 5.4 need to be extended. GetInd will need to produce an individual that is structurally recognized to be an instance of a description in the extended language and Contradict needs to produce a counter-example individual that is still an instance of the alleged subsumee but is not an instance of the alleged subsumer.

The above functions were specifically tied to CLASSIC, and hence would be expected to generalize least well.

For GetInd, note that

if I $\rightarrow$ (**some** p `C`) then

I $\rightarrow$ (**and** (**all** subpC `C`) (**at-least** 1 subpC)).

Here, subpC is a subrole of p. Given an alleged subsumee description, D, in the augmented language, we will rewrite any embedded **some** restriction in this way so that we completely eliminate **some** from the description. For example, if

`D` $\equiv$ (**and** (**some** p `C`) (**some** p `F`)),

the preprocessing phase would rewrite `D` as:

(**and** (**all** `subpC C`) (**at-least** `1 subpC`) (**all** subpF F) (**at-least** 1 subpF)).

The good news is that the rewriting process is deterministic and the new D only contains concept constructors in our previous algorithm. The bad news is that we are now handling subroles, which we had previously avoided. The complication introduced by subroles shows up in our Repair algorithm (from Figure 5.8). It is now possible to add a filler to a subrole and create a bounds conflict on a super-role. If the bounds conflict involves anonymous individuals, then an expanded repair strategy could succeed. For example, consider the description:

D$\equiv$ (**and** (**at-most** 2 p)
(**fills** p I1 I2)
(**all** subpC C) (**at-least** 1 subpC)).

If we try to add an anonymous individual as a filler of subpC, we will get a conflict with the **at-most** restriction on p. The solution is to avoid using anonymous individuals, and instead attempt to use either I1 or I2 as the filler of subpC (and thus propagate the information that I1 or I2 is an instance of C). The Repair algorithm needs to be extended to note this new kind of conflict and to try the new repair strategy. This requires trying every alternative in the filler set of the role where the conflict occurred.

For the second part, the Contradict algorithm needs to take the example individual I $\rightarrow$ D, and make it a counter example by possibly updating it so that I $\not\rightarrow$ E even when E contains (**some** p C). For this case, if I is currently an instance of the **some** restriction and I has an anonymous p-filler that is an instance of C, the algorithm will attempt to retract that filler from p (or p's subrole if the filler is derived). If the bounds restrictions are satisfied on p and its subrole, then the algorithm returns success. Finally, close for nonmembership needs to extended to close any roles for which there exists a **some** restriction.

Finally, in order to prune appropriately with the expanded language, the explanation designer needs to extend the grammar for normalized concept patterns and define the new match rules for the constructor. The normal form for **some**, like that for **fills**, combines the various values for the same role. i.e., the normal form for

(**and** (**some** `child` `DOCTOR`) (**some** `child` `LAWYER`)) is

(**some** `child` {`DOCTOR` `LAWYER`})).

And the normal form for

(**and** (**some** r `PERSON`) (**fills** r `MAMMAL`)) is

(**some** r `PERSON`)

because `PERSON` $\Longrightarrow$ `MAMMAL`.

The syntax of concept patterns can be extended with the production

(**some** <role-pattern> <concept-pattern>).

and the match relation must be extended by rules dealing with **some**:

```
match(some(p,C),      some(p, _X),    < _X,C>).
match(some(p,Cset),   some(p, _X),    < _X,C>):-    choose C in Cset.
```

8.2 Contributions

While considering issues of explaining reasoning systems, we have provided the following three main contributions:

1. A proof-theoretic foundation for explanations of subsumption and membership in description logics, together with techniques for presenting explanations in manageable chunks.

2. A method for explaining nonsubsumption by generating counter-examples using the existing implementation of the CLASSIC system and its explanation component.

3. An object oriented method of pruning object and explanation presentations.

For the first task, we presented a proof-theoretic method of explanation which provides the basis for building customized explanations. We cast the inferences performed by DLs as natural semantics style proof rules and then used them to generate proofs of deductions. Since proofs are usually overly long, we introduced the notion of atomic descriptions to simplify the questions that are asked to explain. We also introduced the notion of atomic justification and multi-step explanation in order to breakup long proofs into stand-alone proof fragments that can be meaningful in isolation. As part of this, we presented some syntactic forms for proof rules which allowed the system to automatically generate follow-up questions to single-step explanations. We also described a variety of inference rule categories (structural subsumption, normalization, skipped, combined) that helped provide better explanations.

The above framework extended naturally to deal with inferences concerning individuals and their class membership. The important case of explaining inconsistencies and errors was shown to fall out from the previous framework once we introduced the important notion of "intermediate objects", which recorded aspects of the inconsistent intermediate state of the knowledge base. After gathering user feedback on error handling, we also defined additional error-explanation support that hides intermediate objects from users.

The above ideas were realized in an implementation of the CLASSIC system, which saw extensive use by us and by others. In addition to validating the feasibility of the approach, the experience led to a number of heuristics, including a recipe for determining atomic descriptions techniques to provide explanation in an implemented normalize/compare DL with minimal impact upon the core DL implementation.

Concerning non-subsumption, our current explanation system provides a simple explanation based on unsatisfied atomic descriptions. In Chapter 6 we suggested explaining non-subsumption by providing a concrete counter-example. We showed how an algorithm for finding such a counter-example could be implemented relying as much as possible on the existing implementation of the DL, in this case CLASSIC, and the explanation component for membership. If we believe this algorithm to be complete, then we could use its inability to find a counter-example to identify when a subsumption relationship has not been discovered due to an *incompleteness* in DL implementation.

Finally, we provided a method of filtering object presentation and explanation generation and presentation using meta information. This was based on a language of concept patterns and a matching algorithm, which returned only portions of the information available. By using meta-objects and meta-roles, such patterns could select the novel or interesting parts of descriptions.

Our work has concentrated on calculating explanations and providing filtering mechanisms. Future work includes presentation strategies that use our work to provide more natural explanations and object presentations for various applications and user populations. We expect that such work would utilize domain knowledge, context, and user profiles. Some presentation techniques that we have just dabbled with include natural language, graphical presentation, and speech. Another area of future work includes knowledge acquisition for DLs in general. Interestingly, in discussions with users, it is rarely additional expressive power that they crave, it is usually a supporting development environment. Knowledge acquisition components that could impact this work significantly would be tools to support users in defining better meta descriptions and domain ontologies that are more natural and better suited to explanation.

Appendix A

Normalization Inferences

The following inferences are used in CLASSIC normalization. They are listed in alphabetical order by inference name followed by the arguments. [] means no arguments are reported.

at-least-and-one-of-implies-fillers [ ] When the **at-least** restriction on a role equals the length of the **one-of** restriction, e.g.,
(**and** (**at-least** 2 `grape`) (**all** `grape` (**one-of** `Zinfandel Petite-Syrah`))), then the fillers for the role must be exactly the individuals in the **one-of** restriction (in this case, the `grape` role must be filled by `Zinfandel` and `Petite-Syrah`).

attribute-implies-at-most-1 [ ] If a role is defined to be an attribute, then the **at-most** restriction is set to `1`, because attributes can have at most `1` filler.

classic-ind-implies-classic-thing [ ] A top-level CLASSIC individual will have the primitive `CLASSIC-THING` in its primitive list (because it must be in the CLASSIC realm). This inference is calculated during explanation only if `CLASSIC-THING` is not there for any other reason.

closed-implies-at-most [ ] If a role on an individual is closed, then the maximum number of fillers the role can have is the current number of fillers. The **at-most** restriction is set to this number.

fillers-implies-at-least [ ] If a role has known fillers (for example, the `grape` role might be filled with `Shiraz` and `Cabernet-Sauvignon`), then the **at-least** restriction on the role is set to the number of fillers (in this example, 2). (If there was already a more specific **at-least** restriction, this inference would not be applied.)

filter-one-of-by-interval [ ] Similar to **filter-one-of-by-tests**, this inference removes values in a **one-of** restriction that are not in the specified interval. If, for example, there is a minimum value of 3 required and a **one-of** restriction of {1 3 5}, then this inference will remove 1 from the **one-of**.

filter-one-of-by-tests [ ] Values in **one-of** restrictions are filtered by tests. If for example something is known to be an `INTEGER` (and thus has the **integerp** test) and has a **one-of** restriction of {1 1.2 5}, then this inference will remove the 1.2 value.

inconsistent-all-implies-at-most-zero [ ] If an inconsistent **all** restriction is encountered, such as (**all** `child` (**and** `FEMALE MALE`)) then the deduction is made that there can be no `children`, and the **at-most** restriction is set to `0`.

inheritance [(1) a told concept from which the information was inherited; (2) the role-path along which the concept was found] For object `A`, if `A` is a subclass or instance of `B`, and `B` has a property C, then `A` inherits property C from `B`.

interval-and-integer-implies-at-most [(1) the **min** restriction; (2) the **max** restriction] When a role is restricted to type `INTEGER`, and the complete interval is specified (both the **min** and the **max**), the **at-most** restriction is set to the number of integers in the interval. For example, if an object has the restriction

(**all** `age` (**and** `INTEGER` (**min** `15`) (**max** `18`))),

then the **at-most** restriction is set to `4`.

interval-implies-at-most [(1) the **min** restriction; (2) the **max** restriction] If a role has an interval, where both the `MIN` and `MAX` are specified and are equal, then there can be at most one filler for the role, and the **at-most** restriction is set to `1`.

interval-implies-host-thing [ ] If a concept or individual has a **min** or **max** restriction, then it will have the primitive `HOST-THING` added to its primitive list (because it must be in the `HOST` realm). This inference is calculated during explanation only if `HOST-THING` is not there for any other reason.

interval-length-equals-at-least-implies-fillers [ ] Similar to **at-least-and-one-of-implies-fillers**, when the interval length equals the **at-least** restriction, and the role is restricted to type `INTEGER`, then the fillers must be exactly the integers in the interval. For example, if an object has the restriction

(**and** (**at-least** 3 `favorite-number`)
(**all** `favorite-number` (**and** `INTEGER` (**min** 14) (**max** 16)))),

then the fillers for `favorite-number` get set to `14`, `15`, and `16`.

interval-test [ ] If a description is known to be an interval, then it must be subsumed by `NUMBER`. Thus, if test restrictions containing the functions **numberp** and **atom** are not already on this object, then this inference adds them.

inverse-propagation [(1) the role on the object whose **all** restriction led to this inference; (2) the role-path along which the **all** restriction was found; (3) the information which was propagated—can be either a named or an unnamed concept. Note: if the information which was propagated is identical to the information currently being explained, the explanation printing functions will not print this argument, but it will still be returned from **cl-exp-args**; (4) if the **all** restriction on the first role is a named concept, which contains a role restriction on the inverse role, which is what got propagated, then this argument is the named concept. Otherwise, it is `NIL`.] If role inverses `r1` and `r2` are defined, and individual `I1` has the following restriction (either at the top level, or on an **all** restriction):

(**and** (**at-least** 1 `r1`) (**all** `r1` (**all** `r2` **C**))),

for some description **C**, then **C** gets propagated onto `I1` (or onto the **all** restriction, if it was not at the top level). For example, if we know that `child` and `parent` are inverse roles, and we know that a particular individual, `Deb`, has at-least 1 `parent` and all of her `parents` have only `ATHLETE`s as children, then `Deb` must be an `ATHLETE`. The argument in this example is the `parent` role, indicating that all of `Deb`'s `parent`'s `children` are `ATHLETE`s.

inverse-role [(1) the individual whose role was known to be restricted (before this inference ran); (2) the role on that individual] If role inverses `r1` and `r2` are defined, then if individual `I1` has role `r1` filled with individual `I2`, then `I2`'s `r2` role is automatically filled with `I1`. If, for example, `parent` and `child` are inverse roles, and CLASSIC is told that `Louise`'s `child` is `Deb`, then `Deb`'s `parent` role gets filled with `Louise`. The arguments in this example are `Louise` and `child`, indicating that `Louise`'s `child` is `Deb`.

one-of-implies-at-most [ ] If an **all** restriction on a role contains a **one-of** restriction, then the largest number of fillers the role can have is the number of individuals in the **one-of** restriction, and the **at-most** restriction is set to this number. For the description (**all** `color` (**one-of** `White Red Rose`)), the **at-most** restriction on the `color` role is set to `3`.

one-of-implies-classic-thing [ ] If a concept or individual has a **one-of** restriction containing CLASSIC individuals, then it will have the primitive `CLASSIC-THING` added to its primitive list (because it must be in the CLASSIC realm). This inference is calculated during explanation only if `CLASSIC-THING` is not there for any other reason.

one-of-implies-host-thing [ ] If a concept or individual has a **one-of** restriction containing `HOST` individuals, then it will have the primitive `HOST-THING` added to its primitive list (because it must be in the `HOST` realm). This inference is calculated during explanation only if `HOST-THING` is not there for any other reason.

one-of-intersection [ ] Different **one-of** restrictions have been intersected to form the final **one-of** restriction. For example, if a concept has two **one-of** restrictions: `(`**one-of** `Louise Dick Helen)` and `(`**one-of** `Dick George Louise)`, either through told-info, inheritance, or some other inference, then the final **one-of** restriction will be `(`**one-of** `Louise Dick)`.

propagation [(1) the individual from which the information was propagated; (2) either a role or list of roles—the role or role-path through which the information was propagated; (3) the information which was propagated—can be either a named or an unnamed concept.] If information about one individual needs to be transferred to another individual because one individual fills a role of another individual, then this information is said to be propagated. If, for example, we know that `PARENT-OF-ATHLETES` describes individuals which have only `ATHLETE`s as children, and we know that `Dick` is a `PARENT-OF-ATHLETES`, and we know that `Dick`'s `child` is `Deb`, then we also know that `Deb` must be an `ATHLETE`. The information that `Deb` is an `ATHLETE` is propagated from the individual `Dick` through the `child` role.

It is also possible that information can be propagated from one individual to another through **all** restrictions and a **fills** restriction. For example, suppose that we don't know who `Jack`'s `friends` are, but we know that he has at least 1 `friend`, all his `friends` have `Susan` as a `teacher`, and all his `friends`' `teachers` are `VEGETARIANS`. Then we know that `Susan` must be a `VEGETARIAN`. This information is propagated from the individual `Jack` through the `friend teacher` role-path.

role-hierarchy-inherit-all [(1) the parent role from which the information was inherited; (2) the role-path along which the **all** restriction was found, not including the final parent role. (The second argument may be needed if the **all** restriction which is inherited is not a named concept.)] If there is an **all** restriction on a role, then all child roles inherit this **all** restriction. For example, if an object has a `wine` role with a value restriction that restricts all fillers to be instances of the concept `WINE`, then the subroles `white-wine` and `red-wine` would inherit this restriction from the `wine` role. The argument in this example is the `wine` role.

role-hierarchy-inherit-at-most [(1) the parent role from which the information was inherited] If a role hierarchy exists, and there is an **at-most** restriction on a role, then its child roles inherit the **at-most** restriction. If, for example, an object has a `wine` role with an **at-most** restriction of `3` and there are two subroles of `wine`, `white-wine` and `red-wine`, then both `white-wine` and `red-wine` would inherit the **at-most** restriction of `3` from their parent role `wine`.

role-hierarchy-inherit-at-least [(1) the child role from which the information was inherited] If a role hierarchy exists, and there is an **at-least** restriction on one of the child roles, then the **at-least** restriction must be "inherited"[1] up the hierarchy to the parent role and all ancestor roles. If in the above example, the `white-wine` role has an **at-least** restriction of 2, then the parent role `wine` also gets an **at-least** restriction of 2. The argument in this example is the `white-wine` role.

role-hierarchy-inherit-fills [(1) the child role from which the information was inherited] If a role hierarchy exists, and fillers are known for a subrole, then the fillers must be "inherited" up the hierarchy to all ancestor roles. If in the above example, the `white-wine` role had a filler of `Forman-Chardonnay`, then the parent role `wine` would also get the filler `Forman-Chardonnay`. The argument in this example is the `white-wine` role.

role-restr-implies-classic-thing [ ] If a concept or individual has a role restriction (**fills**, **all**, **at-least** or **at-most**), then it will have the primitive `CLASSIC-THING` added to its primitive list (because it must be in the CLASSIC realm). This inference is calculated during explanation only if `CLASSIC-THING` is not there for any other reason. For the functions **cl-exp** and **cl-print-exp**, this inference is not shown unless the all keyword is non-`NIL`.

rule [the list of rules which generated this information] If a rule firing caused something to be derived, then the rule will be the explanation for that fact. For example, if the rule `VEG-RULE` says that all `VEGETARIAN`s must be `HEALTHY-THING`s, and `Mary` is classified as a `VEGETARIAN`, then the fact that she is a `HEALTHY-THING` is explained by the rule `VEG-RULE`.[2]

same-as [(1) and (2) the two role paths from the **same-as** restriction; (3) the role-path along which the **same-as** restriction was found] If a **same-as** restriction exists and one role path has an **all** restriction which is a description (not a named

---

[1]Typically, one thinks of information as being inherited down a hierarchy, but we also use the word "inherit" when information moves up a hierarchy.

[2]We do not guarantee a complete explanation if one rule firing causes another rule to fire which in turn causes the atomic fact being explained to become more specific.

concept), then every piece of that description gets propagated to the other role path in the **same-as** restriction. For example, if `Mary`'s `best-friend` is the same as her `sister`, and her `best-friend` has at least 1 `pet`, then her `sister` has at least 1 `pet`, by the **same-as** inference.

same-as-implies-all [(1) and (2) the two role paths from the **same-as** restriction; (3) the role-path along which the **same-as** restriction was found] If a **same-as** restriction exists and one role path has an **all** restriction which is a named concept, then the other role path in the **same-as** restriction will have the same **all** restriction.

same-as-implies-at-least-1 [(1) and (2) the two role paths from the **same-as** restriction; (3) the role-path along which the **same-as** restriction was found] When a **same-as** restriction is asserted on an object, all the attributes in both paths of the **same-as** restriction get **at-least** restrictions of `1`.

same-as-implies-classic-thing [ ] If a concept or individual has a **same-as** restriction, then it will have the primitive `CLASSIC-THING` added to its primitive list (because it must be in the CLASSIC realm). This inference is calculated during explanation only if `CLASSIC-THING` is not there for any other reason. For the functions **cl-exp** and **cl-print-exp**, this inference is not shown unless the all keyword is non-`NIL`.

same-as-implies-filler [(1) and (2) the two role paths from the **same-as** restriction; (3) the role-path along which the **same-as** restriction was found] If a **same-as** restriction exists and one role path has a known filler, the other role path in the **same-as** restriction will have the same filler.

subrole-combination-implies-at-least [the subroles used to calculate the restriction] When a role hierarchy exists, subrole combinations can come into play. If two child roles have disjoint **all** restrictions (i.e., they must be filled with distinct individuals), then we can calculate a minimum for the **at-least** restriction on the parent role by just adding up the **at-least** restrictions on the child roles.

For example, if `relative`, `brother`, and `sister` are all roles and we know that `sister` must be `FEMALE`, `brother` must be `MALE`, and there is at least `1` `sister` and at least `1` `brother`, then we know that there is an **at-least** restriction of 2 on `relative`.

This inference also applies when two child roles have **all** restrictions that are not disjoint, but have a finite overlap. This would occur when at least one of the roles has a **one-of** restriction. In this case, the maximum number of instances they could have in common is calculated, and a new **at-least** restriction is calculated based on this.

subrole-combination-implies-at-most [(1) the parent role that has the **at-most** restriction; (2) a list of the subroles that have the **at-least** restrictions] Given a role hierarchy, an **at-most** restriction on a parent role can combine with **at-least** restrictions on child roles to produce an **at-most** restriction on a different child role. Given the roles `relative`, `brother`, and `sister` defined above, where the **all** restrictions on `brother` and `sister` are disjoint, then if we have an **at-most** `5 relative` restriction, and we know there are **at-least** `2 brothers`, then this inference is used to deduce that there are **at-most** `3 sisters`.

test-c-implies-classic-thing [ ] If a concept or individual has a **testc** restriction, then it will have the primitive `CLASSIC-THING` added to its primitive list (because it must be in the CLASSIC realm). This inference is calculated during explanation only if `CLASSIC-THING` is not there for any other reason.

test-h-implies-host-thing [ ] If a concept or individual has a **testh** restriction, then it will have the primitive `HOST-THING` added to its primitive list (because it must be in the `HOST` realm). This inference is calculated during explanation only if `HOST-THING` is not there for any other reason.

told-info [ ] CLASSIC was directly told this information.

Appendix B

Subsumption Inferences

The following inferences are used to determine structural subsumption in CLASSIC.

at-least-ordering [(1) subsumee **at-least**; (2) subsumer **at-least**] The **at-least** restriction on the subsumee is no smaller than the **at-least** on the subsumer. For example, (**at-least** `3 r`) is subsumed by (**at-least** `2 r`).

at-most-ordering [(1) subsumee **at-most**; (2) subsumer **at-most**] The **at-most** restriction on the subsumee is no larger than the **at-most** on the subsumer.

bad-at-least-ordering [(1) "subsumee" **at-least**; (2) "subsumer" **at-least**] The **at-least** restriction on the "subsumee" is smaller than the **at-least** on the "subsumer".

bad-at-most-ordering [(1) "subsumee" **at-most**; (2) "subsumer" **at-most**] The **at-most** restriction on the "subsumee" is larger than the **at-most** on the "subsumer".

bad-max-ordering [(1) "subsumee" **max**; (2) "subsumer" **max**] The **max** restriction on the "subsumee" is larger than the **max** on the "subsumer".

bad-min-ordering [(1) "subsumee" **min**; (2) "subsumer" **min**] The **min** restriction on the "subsumee" is smaller than the **min** on the "subsumer".

doesnt-satisfy-incoherent [(1) the incoherent **all** restriction on the "subsumer"; (2) the **all** restriction on the "subsumee"] An incoherent **all** restriction cannot be satisfied.

filler-subset [ ] The fillers of the subsumer are a subset of the fillers of the subsumee.

fillers-and-at-most-satisfies-all [(1) subsumee fillers; (2) subsumer **all** restriction] If all the fillers of a role `r1` on an individual `I1` are known (the number of fillers equals the **at-most** restriction), and each of the fillers satisfies a description `D1`, then `I1` satisfies the description (**all** `r1 D1`). For example, if `Mary` and `Bill` are both known to be `ATHLETES`, and `Sue` is defined as

(**and** (**fills** `child Mary Bill`) (**at-most** `2 child`)),

then `Sue` satisfies the description

(**all** `child ATHLETE`). This inference also applies to concept subsumption, when the subsuming **all** restriction is a `HOST` concept, and the subsumed concept's fillers are `HOST` individuals.

fillers-dont-satisfy-all [(1) "subsumee" fillers which don't satisfy the **all** restriction; (2) "subsumer" **all** restriction] Not all the fillers satisfy the **all** restriction.

fillers-missing [(1) the fillers not satisfied] The fillers of the "subsumer" are not a subset of the fillers of the "subsumee".

identical-all-restr [no arguments, if both **all** restrictions are the same classified concept; otherwise: (1) subsumer **all** restriction; (2) subsumee **all** restriction] The **all** restriction on the subsumer is equivalent to the **all** restriction on the subsumee.

ind-doesnt-pass-tests [(1) the tests the individual doesn't pass] The individual doesn't satisfy a test restriction. An individual can satisfy a test if running the test on the individual returns `T`, or if the individual has the identical test in its descriptor.

ind-doesnt-satisfy-same-as [(1) the **same-as** restrictions the individual doesn't satisfy] The individual doesn't satisfy a **same-as** restriction.

ind-member-of-one-of [ ] The individual satisfies a **one-of** restriction because it is one of the individuals in the **one-of**. There is no other way that a top-level individual can satisfy a **one-of** restriction.

ind-not-member-of-one-of [(1) the **one-of** restriction] The individual does not satisfy the **one-of** restriction because it is not one of the individuals in the **one-of**.

ind-passes-test [ ] The individual satisfies a test restriction because when the test runs on the individual, it returns `T`. Note: an individual can also satisfy a test if it has the identical test in its descriptor (**test-existence**). The test gets printed during explanation.

ind-satisfies-interval [(1) the piece of the interval currently being explained (the **min** or the **max**)] For a top-level `HOST` individual: for the **min** and/or **max**, whichever are specified, the individual is no smaller than the **min**, and no larger than the **max**.

ind-satisfies-same-as-with-filler-and-paths [(1) the intermediate individual, (2) and (3) the two paths on the individual which are equivalent, and (4) optionally, a list containing pairs of a **same-as** restriction which contributes to those paths being equivalent, along with the role-path along which the **same-as** was found. The final argument is not there if the two paths are the two paths of a **same-as**] The individual satisfies the **same-as** because there is an intermediate individual along both the paths of the **same-as**, and then the rest of both paths are equivalent somehow through other **same-as** restrictions. For example, suppose concept `C1` is defined as: `(SAMEAS (favorite-teacher best-friend) (mother husband))`. Now suppose that `Jane`'s `favorite-teacher` and `mother` are both `Sally`, and `Sally` has the restriction: `(`**same-as** `(best-friend) (husband))`. If we call `(`**cl-exp-subsumes-ind** `@C1 @Jane)`, we see that the **same-as** restriction is satisfied by **ind-satisfies-same-as-with-filler-and-paths**, with the intermediate individual `Sally`, and the equivalent tail paths: `(husband)` and `(best-friend)`. Note: this inference only applies to individuals.

interval-satisfies-one-of [(1) subsumee **min**; (2) subsumee **max**; (3) subsumer **one-of**] The subsumee is restricted to being an `INTEGER`. The **min** and **max** are both specified, and each integer from the **min** through the **max** is in the subsumer's **one-of** restriction. This inference will never appear on a top-level individual.

max-ordering [(1) subsumee **max**; (2) subsumer **max**] The **max** restriction on the subsumee is no larger than the **max** on the subsumer. This inference will never appear on a top-level individual.

min-ordering [(1) subsumee **min**; (2) subsumer **min**] The **min** restriction on the subsumee is no smaller than the **min** on the subsumer. This inference will never appear on a top-level individual.

non-subsuming-all-restr [(1) the "subsumer" **all** restr; (2) the "subsumee" **all** restr] The **all** restriction on the "subsumee" doesn't satisfy the **all** restriction on the "subsumer".

one-of-not-satisfied [(1) "subsumee" **one-of** restriction; (2) "subsumer" **one-of** restriction] The **one-of** in the "subsumee" contains individuals not in the **one-of** of the "subsumer".

one-of-satisfies-max [(1) subsumee **one-of** restriction; (2) subsumer **max** restriction] All the individuals in the **one-of** restriction are numbers, and each one is no larger than the **max**. This inference will never appear on a top-level individual.

one-of-satisfies-min [(1) subsumee **one-of** restriction; (2) subsumer **min** restriction] All the individuals in the **one-of** restriction are numbers, and each one is no smaller than the **min**. This inference will never appear on a top-level individual.

one-of-satisfies-tests [(1) the subsumee's **one-of**] Every individual in the subsumee's **one-of** restriction satisfies all the subsumer's tests. This inference will never appear on a top-level individual.

one-of-subset [ ] The individuals in the **one-of** restriction on the subsumee are a subset of the individuals in the **one-of** restriction on the subsumer. This inference will never appear on a top-level individual.

primitive-subset [ ] The primitives of the subsumer are a subset of the primitives of the subsumee.

prims-not-satisfied [(1) the primitives not satisfied] The primitives of the "subsumer" are not a subset of the primitives of the "subsumee".

role-not-closed [(1) the "subsumer" **all** restriction; (2) the role] The fillers cannot satisfy the **all** restriction, because the role is not closed on the individual.

same-as-existence [ ] The subsumee satisfies a **same-as** restriction because it has the identical **same-as** restriction.

same-as-not-satisfied [(1) the **same-as** restrictions not satisfied] Not all the **same-as** restrictions are satisfied.

same-as-subset [ ] The **same-as** restrictions of the subsumer are a subset of the **same-as** restrictions of the subsumee.

satisfies-same-as-with-filler [(1) the filler] The same filler is at the end of each of the two paths of the **same-as** restriction.

subsuming-all-restr [(1) subsumer **all** restriction; (2) subsumee **all** restriction] The **all** restriction on the subsumer subsumes the **all** restriction on the subsumee. If you want to find out why, you can call **cl-exp-subsumes-concept** on the two **all** restrictions.

test-existence [ ] The subsumee satisfies a test restriction because it has the identical test restriction. The test gets printed during explanation.

test-subset [ ] The test restrictions of the subsumer are a subset of the test restrictions of the subsumee.

tests-not-satisfied [(1) the tests not satisfied] The tests of the "subsumer" are not a subset of the tests of the "subsumee".

thing-doesnt-satisfy [(1) the **all** restriction not satisfied] `THING` cannot satisfy a non-vacuous **all** restriction.

transitive-closure-or-distribution [any number of (<**same-as** restr> <role-path>) pairs, where the role-path is the role-path from the top-level object, along which the contributing **same-as** restr was found] A concept or individual satisfies a **same-as** restriction by a combination of transitive closure and distribution over

all restrictions. For example, suppose concept `C1` is defined as:

(**same-as** `(lawyer) (best-friend)`),

and concept `C2` is defined as:

(**and** (**same-as** `(lawyer) (teacher)`)

(**same-as** `(teacher) (best-friend)`)).

If we call (**cl-exp-subsumes-concept** `@c1 @c2`), we see that the **same-as** restriction on `C1` is satisfied by **transitive-closure-or-distribution**, with the two **same-as** restrictions: `((teacher) (best-friend))` and `((lawyer) (teacher))`.

Appendix C

Conflict Inferences

This appendix describes the CLASSIC conflict error codes. CLASSIC has two kinds of error codes: syntactic errors that catch typing problems and semantic conflicts that may be used to derive incoherent information about an object. Explanation will report the semantic codes.

The arguments to each error code are listed after the error code name, enclosed within [ ]'s. [ ] means no arguments.

concept-inconsistent-rules-conflict [(1) antecedent concept name] A concept has inconsistent rule consequents.

disjoint-prims-conflict [(1) disjoint primitive told concept; (2) disjoint primitive told concept] Attempt to combine two disjoint primitive concepts.

empty-one-of-conflict [ ] A **one-of** restriction is the empty set—no individuals can satisfy this restriction. This may have been caused by no individuals being enumerated in a **one-of** restriction, or by the intersection of two **one-of** restrictions yielding the empty set, or because a **one-of** restriction was filtered by tests or an interval. This error is signaled only if an error is caused at the top level, i.e., (**cl-define-concept** `'CON '`(**all** `age` (**one-of**))) does not signal an error since it describes all individuals with no ages. This conflict would be reported as the reason that `CON` has a value restriction on r of `NOTHING`.

fillers-internal-realm-conflict [(1) the filler individuals] The fillers contain individuals in both the CLASSIC and `HOST` realms.

inconsistent-bounds-conflict [(1) role; (2) **at-least** restriction; (3) **at-most** restriction] A role has inconsistent number restrictions: the **at-least** restriction is greater than the **at-most** restriction.

inconsistent-interval-conflict [(1) **min** restriction; (2) **max** restriction] A `HOST` concept has an inconsistent interval: the **min** restriction is greater than the **max** restriction.

ind-interval-conflict [(1) individual; (2) min; (3) max] An individual is not within an interval.

ind-one-of-conflict [(1) individual; (2) list of individuals in **one-of** restriction] An attempt is being made to combine an individual with a **one-of** restriction, of which the individual is not a member.

ind-test-conflict [(1) individual name; (2) test function; (3) arguments to test function, if any] An individual is inconsistent with a test restriction.

one-of-internal-realm-conflict [(1) **one-of** restriction description] The **one-of** restriction contains individuals in both realms.

realm-conflict [ ] There is a realm conflict within the object currently being processed. (Note: this error is signaled if a primitive concept is being created in the `HOST` realm.)

Appendix D

Trace of Counterexample Generation

This is a CLASSIC trace of the scenario described in Section 5.4. The important thing to note is how IndB changes throughout the scenario so we will print it out periodically and note the changes.

```
(cl-startup)
(cl-define-primitive-concept 'F 'classic-thing)
(cl-define-roles '(q r t))
(cl-define-roles '(p) t)
(cl-define-concept 'D7  '(and
                          (all q (and (fills r IndB)
                                      (all r (fills p IndC))))
                          (all t (and F
                                  (atleast 1 p)))
                          (fills t IndB)))
(cl-define-concept 'E7 '(atmost 0 q))
(cl-create-ind 'I7 'D7)


USER(37): (cl-print-ind @i7)
I7 ->
 Derived Information:
  Parents: D7
  Ancestors: THING CLASSIC-THING
  Role Fillers and Restrictions:
   Q =>
```

```
    Role Restrictions:
     R[1 ; INF] -> IndB
       => Role Restrictions:
       P![1 ; 1] -> IndC
   T[1 ; INF] -> IndB
     => Primitive ancestors: F CLASSIC-THING
    Role Restrictions:
     P![1 ; 1]
@i{I7}
USER(38): (cl-print-ind @IndB)
IndB ->
 Derived Information:
  Primitive ancestors: F CLASSIC-THING
  Parents: F
  Ancestors: THING CLASSIC-THING
  Role Fillers and Restrictions:
   P![1 ; 1]
@i{IndB}
```

We could close I7 now. If we add (**and** (**all** q (**at-most**1 r)) (**at-most** 0 q) (**at-most** 1 t)) to @I7, then all I7s roles have all their fillers known. At this point, I7 would be printed as follows:

```
USER(42): (cl-print-ind @i7)
I7 ->
 Derived Information:
  Parents: D7
  Ancestors: THING CLASSIC-THING
  Role Fillers and Restrictions:
   T[1 ; 1] -> IndB
     => Primitive ancestors: F CLASSIC-THING
```

```
    Role Restrictions:
     P![1 ; 1]
   Q[0 ; 0] =>
    Role Restrictions:
     R[1 ; 1] -> IndB
       => Role Restrictions:
       P![1 ; 1] -> IndC
@i{I7}
```

Next, we will need to generate fillers for I7's fillers such as IndB. At this point, it is known that IndB has **at-least** 1 r-filler, but we don't have an identity for that filler. In order to close IndB with respect the the knowledge base, we generate an anonymous filler and use that to fill the r role.

```
USER(44): (cl-ind-add @IndB '(fills p Anon-p))
*WARNING*: Creating individual @i{ANON-P}.
@i{IndB}
USER(45): (cl-print-ind @IndB)
IndB ->
 Derived Information:
  Primitive ancestors: F CLASSIC-THING
  Parents: F
  Ancestors: THING CLASSIC-THING
  Role Fillers and Restrictions:
   P![1 ; 1] -> Anon-P
@i{IndB}
```

Now, IndB is consistent and all of its filler information is complete. If we close all of the roles on IndC and on our new Anon-p, then I7 is closed with respect to the knowledge base.

At this point, I7 is an instance of both D7 and E7 but it is still possible to modify it so that it becomes a non-instance of E7.

```
USER(58): (cl-instance? @i7 @e7)
T
```

In order for I7 to remain an instance of D7 but not be an instance of E7, we will need to add one (or more) q-fillers. We will attempt to add such an individual. This will cause a problem in CLASSIC so in the CLASSIC statement below, we bind the return values so that we can print the intermediate object that results:

```
USER(59): (multiple-value-setq (v1 v2 v3 v4)
     (cl-ind-add @i7 '(fills q New-ind)))
*WARNING*: Creating individual @i{NEW-IND}.
*WARNING*: Inconsistent number restriction for role @r{P} - at-least: 2;
  at-most: 1. Found while processing object @i{INDB}.
*WARNING*: Retracting addition of expression.
*CLASSIC ERROR* while processing
  (CL-IND-ADD @i{I7} (FILLS Q NEW-IND))
    occurred on object @i{INDB-*INCOHERENT*}
    along role-path (@r{P}):
  Inconsistent number restriction for role @r{P} - at-least: 2; at-most: 1.
CLASSIC-ERROR
USER(60): (cl-print-ind v4)
IndB-*Incoherent* -> : (INCONSISTENT-BOUNDS-CONFLICT @r{P} 2 1)
 Derived Information:
  Primitive ancestors: F CLASSIC-THING
  Parents: F
  Ancestors: THING CLASSIC-THING
  Role Fillers and Restrictions:
   P![2 ; 1] -> Anon-P IndC
     => THING
 @i{IndB-*INCOHERENT*}
```

What is shown here is that when we tried to add the new individual, `New-Ind`, `IndB` became incoherent. This does not mean that the process is a failure however, it is possible to remove the anonymous individual as an `r`-filler of `IndB` and eliminate the incoherency and produce a counterexample.

```
USER(61): (cl-ind-remove @IndB '(fills p Anon-p))

USER(62): (cl-ind-add @i7 '(fills q New-ind1)))
*WARNING*: Creating individual @i{NEW-IND1}.
@i{I7}
```

Now there is no incoherency on `IndB`. We can print the current form of `I7`.

```
USER(66): (cl-print-ind @i7)
I7 ->
 Derived Information:
  Parents: D7
  Ancestors: THING CLASSIC-THING
  Role Fillers and Restrictions:
   Q[1 ; INF] -> New-Ind1
     => Role Restrictions:
     R[1 ; 1] -> IndB
       => Role Restrictions:
       P![1 ; 1] -> IndC
   T[1 ; 1] -> IndB
     => Primitive ancestors: F CLASSIC-THING
    Role Restrictions:
     P![1 ; 1]
@i{I7}
```

Finally, if we close the remaining open roles, i.e., if we close `q` on `I7` and close `r` on `New-Ind1`, then we have succeeded at creating a counterexample which is closed with respect to the knowledge base.

```
USER(67): (cl-exp-not-subsumes-ind @e7 @i7)
I7 ->
  Role Restrictions:
  Q
   at-most: (1) :
     Subsumption:
       BAD-AT-MOST-ORDERING: At-most: 1 doesn't satisfy at-most: 0
 @i{I7}
```

So, the point of this lengthy example is to show that the contradict function in our algorithm must be able to know how to retract appropriate fillers when necessary. We handle this in our algorithm by looking for the particular kind of bounds conflict on a role that has an anonymous filler in it and then "repair" the problem by removing the anonymous filler and replace it with a named individual identified in the conflict.

References

[1] Z. Ahmed and L. Wanger and P. Kochevar. An Intelligent Visualization System for Earth Science Data Analysis. In *Journal of Visual Languages and Computing*, pp.307–320, (1994) 5.

[2] H. Ait-Kaci, *A lattice theoretic approach to computation based on a calculus of partially ordered type structures.*, PhD Thesis, University of Pennsylvania, 1984.

[3] H. Ait-Kaci and A. Podelski, "An overview of Life", *Next Generation Information System Technology: Proc. 1st Int. East/West Data Base Workshop*, Springer-Verlag LNCS 504, pp.42–58, 1990

[4] T.W. Anwar, H. Beck and S. Navathe, "Knowledge mining by imprecise querying: a classification-based approach", *Proc. 8th Conference on Data Engineering'*, Tempe, Arizona, February 1992, 622–630.

[5] Y. Arens, C.Y. Chee, C.N. Hsu, and C. Knoblock, "Retrieving and integrating data from multiple information systems", *Int. J. of Intelligent and Cooperative Information Systems 3*(1), 1994.

[6] T. Arora, R. Ramakrishnan, W.G. Roth, P. Seshadri, and D. Srivastava. Explaining Program Executions in Deductive Systems. Proceedings of the International Conference on Deductive and Object-Oriented Databases, Phoenix, AZ, 1993, pp 101–119.

[7] AT&T Bell Laboratories and University of Pittsburgh CLASSIC Knowledge Representation System Tutorial. Available by anonymous ftp on pogo.isp.pitt.edu, in ftp/pub/classic-tutorial.

[8] F. Baader and B. Hollunder. 'KRIS : Knowledge Representation and Inference System. in *SIGART Bulletin* 2(3), 1991, pp 8-14.

[9] F. Baader, B. Hollunder, B. Nebel, H. Profitlich. An Empirical Analysis of Optimization Techniques for Terminological Representation Systems or: Making KRIS get a move on. In *Applied Intelligence*, 4(2), 1994, pp. 109–132. Also appears in the *Proceedings of the Third International Conference on the Principles of Knowledge Representation and Reasoning*, Cambridge, Mass., October, 1992, pp 270–281.

[10] F. Baader, M. Lenzerini, W. Nutt, and P. F. Patel-Schneider, editors. *Working Notes of the 1994 Description Logic Workshop*, May 1994.

[11] J.M. Blanco, A. Illarramendi, A. Goni, "Building a Federated Relational Database System: An Approach using a Knowledge-Based System", *Int'l J. of Intelligent and Cooperative Information Systems*, vol. 3, no. 4, 1994, pp. 415-455

[12] 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.

[13] A. Borgida. Description Logics in Data Management. To appear in *IEEE Trans. on Knowledge and Data Management.* An earlier version is Available as Rutgers Tech. Report DCS-TR-295.

[14] A. Borgida. Towards the Systematic Development of Description Logic Reasoners: CLASP reconstructed. In *Principles of Knowledge Representation and Reasoning; Proceedings of the Third International Conference*, Cambridge, Mass., October 1992, pp.259–269.

[15] A. Borgida and R. J. Brachman. Customizable Classification Inference in the ProtoDL Description Management System. In *Proceedings of the International Conference on Information and Knowledge Management*, Baltimore, MD, November 1992, pp.482–490.

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

[17] A. Borgida, C.L. Isbell, and D. L. McGuinness. Reasoning with black boxes: Handling test concepts in CLASSIC. In *Proceedings of the International Description Logic Workshop*, Cambridge, Mass., 1996.

[18] A. Borgida, M. Lenzerini, D. Nardi, and B. Nebel, editors. *International Workshop on Description Logics*, June 1995.

[19] A. Borgida and D.L. McGuinness. Asking Queries about Frames. To appear in the *Proceedings of the Fifth International Conference on the Principles of Knowledge Representation and Reasoning*, Cambridge, Mass., October, 1996.

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

[21] R. J. Brachman, A. Borgida, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Resnick. The CLASSIC knowledge representation system, or, KL-ONE: The next generation. In *Proceedings of the 1992 International Conference on Fifth Generation Computer Systems*, June 1992.

[22] R. J. Brachman, A. Borgida, D. L. McGuinness, and L. A. Resnick. The CLASSIC knowledge representation system, or, KL-ONE: The next generation. In *Proceedings of the Workshop on Formal Aspects of Semantic Networks*, February 1989.

[23] R. J. Brachman and D. L. McGuinness. Knowledge representation, connectionism, and conceptual retrieval. In *Proceedings of the 1988 ACM SIGIR International Conference on Research and Development in Information Retrieval*, pages 161–174, June 1988.

[24] R. J. Brachman, D. L. McGuinness, P. F. Patel-Schneider, L. A. Resnick, and A. Borgida. Living with CLASSIC: When and How to Use a KL-ONE-Like Language. In *Principles of Semantic Networks: Explorations in the representation of knowledge*, J. Sowa, editor, Morgan-Kaufmann, 1991, pp. 401–456.

[25] 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 *Proceedings of the First International Conference on Information and Knowledge Management.* International Society for Mini and Microcomputers, November 1992.

[26] 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. Integrated Support for Data Archaeology. in *International Journal of Intelligent and Cooperative Information Systems*, 2(2), 1993, pp. 159–185.

[27] D. Brill. LOOM reference manual: Version 2.0. University of Southern California. 1993.

[28] R.E. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Trans. Comput.,* C-35(8), 1986.

[29] M. Buchheit, F. Donini, W. Nutt, and A. Schaerf. Refining the Structure of Terminological SystemsL Terminology = Schema + Views. In *Proceedings of AAAI*, 1994, Seattle, Wa., pp. 199–204.

[30] M. Buchheit, F. Donini, W. Nutt, and A. Schaerf. A Refined Architecture for Terminological Systems: Terminology = Schema + Views. DFKI Research Report DFKI-RR-95-09, 1995.

[31] M. Buchheit, F. Donini, and A. Schaerf. Decidable Reasoning in Terminological Knowledge Representation Systems. In *Journal of Artificial Intelligence Research, vol. 1*, 1993, pp. 109–138.

[32] M.Buchheit, M. Jeusfeld, W. Nutt, and M. Staudt, "Subsumption between queries in object-oriented databases", *Information Systems* 19(1), pp.33-54, 1994.

[33] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. in *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, Philadelphia, Pa., June 1990, 428–439.

[34] L. Byrd. Understanding the Control Flow of Prolog Programs. In *Proceedings of the Logic Programming Workshop*, July 1980, pp. 127–138.

[35] B. Chandrasekaran and S. Mittal. Deep versus compiled knowledge approaches to diagnostic problem-solving. In *Proceedings of the National Conference on Artificial Intelligence*, AAAI, 1982, pp.349–354.

[36] D. Chester. The translation of formal proofs into English. In *Artificial Intelligence*, 7:261–278, 1976.

[37] W. Clancey. The epistemology of a rule-based expert system: A framework for explanation. In *Artificial Intelligence*, 20(3):215–251, 1983.

[38] W.F. Clocksin and C.S. Mellish. Programming in Prolog. Springer-Verlag, 1981.

[39] W.W. Cohen and H. Hirsh. Learning the CLASSIC Description Logic: Theoretical and Experimental Results. In *Principles of Knowledge Representation and Reasoning; Proceedings of the Fourth International Conference*, Bonn, Germany, May 1994, pp.121–133.

[40] W.W. Cohen, A. Borgida, and H. Hirsh. Computing Least Common Subsumers in Description Logics. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, San Jose, California, 1992.

[41] R. Davis and D. Lenat. TEIRESIAS: Applications of meta-level knowledge. in *Knowledge-Based Systems in Artificial Intelligence* McGraw-Hill, 1982.

[42] J. De Kleer, J. Doyle, C. Rich, G.L. Steele, and G.J. Sussman. AMORD: a deductive procedure system, AI-435, MIT AI Laboratory, 1978.

[43] M. Derthick. An Epistemological evel Interface for CYC. MCC Technical Report. ACT-CYC-084-90, 1990.

[44] J. Despeyroux. Theo: An Interactive Proof Development System. in *Special Issue on Programming Logic: The Scandinavian Journal on Computer Science and Numerical Anaysis*, BIT (32), pp. 15-29, 1992.

[45] P. Devanbu, R.J. Brachman, P.J. Selfridge, and B. Ballard. LaSSIE—A Knowledge-Based Software Information System. In Communications of the ACM, May 1991. Special issue containing best papers from *Proceedings of the 12th International Conference on Software Engineering.* Also appears in the *IEEE Computer Society Tutorial on Domain Analysis*, Prieto-Diaz and Arango (Eds), in *Automating Software Design*, McCartney and Lowry (Eds) (*MIT Press*).

[46] P. Devanbu and D. Litman. Plan-based Terminological Reasoning. In *Proceedings of the Second International Conference on Principles of Knowledge Representation*, Boston, Mass., April, 1991.

[47] P. Devanbu and M. Jones. The Use of Description Logics in KBSE Systems. In *Proceedings of the 16th International Conference on Software Engineering*, May, 1994.

[48] F.M Donini, M. Lenzerini, D. Nardi, and W.Nutt. The Complexity of Concept Languages. In *Proceedings IJCAI*, Sydney, Australia, 1991.

[49] F.M Donini, M. Lenzerini, D. Nardi, A. Schaerf, and W.Nutt. Adding Epistemic Operators to Concept Languages. In *Proceedings KR92*, 1992, pp. 341–353.

[50] F.M Donini, D. Nardi, and R. Rosati. Epistemic ALC-Knowledge Bases. In *Proceedings of the International Workshop on Description Logics*, Rome, Italy, 1995, pp. 13-18.

[51] A. Edgar and J.P. Pelletier. Natural Language Explanation of Natural Deduction Proofs. In *Proceedings of the First Conference of the Pacific Association for Computational Linguistics* Simon Fraser University, Vancouver, Canada, 1993.

[52] M. Eisenstadt and M. Brayshaw. The Transparent Prolog Machine (TPM): An execution model and graphical debugger for logic programming. In *Journal of Logic Programming*, pp.277–342, 5(4), December, 1988.

[53] A. Felty and D. Miller Proof Explanation and Revision. Department of Compter and Information Science School of Engineering and Applied Science. University of Pennsylvania. Report MS-CIS-88-17, 1988

[54] T. Gaasterland. Cooperative Explanation in Deductive Databases. In working notes of the *AAAI Spring Symposium on Cooperative Explanation*, 1992.

[55] T. Gaasterland, P. Godfrey, and J. Minker. An overview of cooperative answering. In *Journal of Intelligent Information Systems*, Kluwer Academic Publishers, vol. 1, no. 2, pp. 123-157, 1992.

[56] T. Gaasterland, P. Godfrey, and J. Minker. Relaxation as a platform for cooperative answering. In *Journal of Intelligent Information Systems*, 1:293-321, 1992.

[57] A. Gal. Cooperative Responses in Deductive Databases. Ph.D. Thesis. Computer Science Department, University of Maryland, College Park. 1988.

[58] Y. Gil. Knowledge Refinement in a Reflective Architecture. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1994.

[59] M-H. Greboval and G. Kassel. The Production of Explanations, Seen as a Design Task: A Case Study. In *Proceedings of the Eleventh European Conference on Ariticial Intelligence*, Amsterdam, The Netherlands, August 1994, pp.351–355.

[60] H. Grice. Logic and conversation. In P. Cole and J. Morgan, editors, *Syntax and Semantics.* Academic Press, 1975.

[61] P. Hanschke. YAEF: Yet Another Epistemic Formalism. In *Proceedings of the International Workshop on Description Logics*, Bonn, Germany, 1994.

[62] H. Horacek. The Production of Explanations, Seen as a Design Task: A Case Study. In *Proceedings of the Tenth European Conference on Artificial Intelligence*, Vienna, Austria, August 1992, pp. 500-504.

[63] H. Horacek. Towards finding the reasons behind – generating the content of explanation. In *Proceedings of GWAI*, 1991.

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

[65] B. Hollunder, W. Nutt, and M. Schmidt-Schauss. Subsumption algorithms for concept description languages. In *Proceedings of the Ninth European Conference on Artifial Intelligence*, Stockholm, August 1990, pp.348–353.

[66] P. Hors and M.-C. Rousset. Consistency of Structured Knowledge: A Formal Framework Based on Description Logics. In *Expert Systems With Applications 8(3)*, 1995, pp.371–380.

[67] X. Huang. PROVERB – A System Explaining Machine-Found Proofs. In *Proc. of the sixteenth Annual Conference of the Cognitive Science Society* Atlanta, Georgia, 1994.

[68] X. Huang and A. Fiedler. Presenting Machine-Found Proofs. In *Proc. of the thirteenth International Conference on Automated Deduction*, 1996.

[69] X. Huang, M. Kerber, M. Kohlhase, E. Melis, D. Nesmith, J. Richts, and J. Siekmann. Omega-MKRP: A Proof Development Environment. In Alan Bundy, ed., *Automated Deduction — CADE-12* Proceedings of the Twelth International Conference on Automated Deduction, 1994 pp. 788-792.

[70] T. S. Kaczmarek, R. Bates, and G. Robbins. Recent Developments in NIKL. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, Philadelphia, PA, 1986, pp. 978–985.

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

[72] M. Kaufmann An Assistant for Reading Nqthm Proof Output. Computational Logic Inc. Technical Report 85, November 1992.

[73] T. Kirk, A Y. Levy, Y. Sagiv, and D. Srivastava. The Information Manifold. In working notes of the *AAAI Spring Symposium on Information Gathering from Heterogeneous Distributed Environments*, 1995.

[74] D.B Lenat and R.V. Guha. Building large knowledge-based systems : representation and inference in the Cyc project. Addison-Wesley Pub. Co. 1990.

[75] A Y. Levy, A. Rajaraman, and J.J. Ordille. Query answering algorithms for information agents In *Proceedings of the National Conference on Artificial Intelligence*, Portland, Oregon, 1996, pp.40–47.

[76] A Y. Levy, D. Srivastava, and T. Kirk. Data model and query evaluation in global information systems. In *Journal of Intelligent Information Systems, Special Issue on Networked Information Discovery and Retrieval*, 5(2), September, 1995.

[77] H. L. Levesque and R.J. Brachman. A fundamental Tradeoff in Knowledge Representation and Reasoning. In Brachman and Levesque, editors, *Readings in Knowledge Representation.* Morgan Kaufmann, 1985, pp. 42–70.

[78] S. Loiseau and M.-C. Rousset. Formal Verification of Knowledge Bases Focused on Consistency: two experiments based on ATMS techniques. In *International Journal of Expert Systems: Research and Applications*, September 1993.

[79] R. M. MacGregor. The Evolving Technology of Classification-based Knowledge Representation Systems. In John Sowa, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge.* Morgan Kaufmann, 1991.

[80] R. M. MacGregor. A Deductive Pattern Matcher. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, St. Paul, Minn, August, 1988, pp 403–408.

[81] R. M. MacGregor., Inside the LOOM Description Classifier, *ACM SIGART Bulletin* 2(3), 1991.

[82] R. M. MacGregor and R. Bates. The LOOM Knowledge Representation Language. Technical Report ISI/RS-87-188, USC/Information Sciences Institute, Marina del Ray, CA, 1987.

[83] R. M. MacGregor and D. Brill. Recognition Algorithms for the Loom Classifier In *Proceedings of the Tenth National Conference on Artificial Intelligence*, San Jose, CA, July, 1992, pp 774–779.

[84] W. Mark, S. Tyler, J. McGuire, J. Schlossberg. Commitment-Based Software Development. *IEEE Transactions on Software Engineering 18:10*, October, 1992.

[85] E. Mays, R. MacGregor, D. L. McGuinness, and T. Russ, editors. *Working Notes, AAAI Fall Symposium Series Symposium: Issues in Description Logics: Users Meet Developers.* American Association for Artificial Intelligence, October 1992.

[86] E. Mays, R. Weida, R. Dionne, M. Laker, B. White, C. Liang, and F.J. Oles Scalable and Expressive Medical Terminologies In *Proceedings of the 1996 AMIA Fall Symposium.*

[87] D. L. McGuinness. The CLASSIC knowledge representation system: Implementation, applications, and beyond. In Nebel et al., *International Workshop on Terminological Logics.* German Research Center for Artificial Intelligence (DFKI), May 1991, pages 80–86. Document D-91-13.

[88] D. L. McGuinness. Making description logic based knowledge represenation systems more usable. In Mays et al. *Working Notes, AAAI Fall Symposium Series Symposium: Issues in Description Logics: Users Meet Developers.* American Association for Artificial Intelligence, October 1992. pages 56–58.

[89] D. L. McGuinness. Where are all the users. In Baader et al. *Working Notes of the 1994 Description Logic Workshop*, May 1994.

[90] D. L. McGuinness and A. Borgida. Explaining Subsumption in Description Logics. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, Montreal, Canada, August, 1995, pp. 816-821.

[91] D. L. McGuinness, L. Alperin Resnick, and C. Isbell. Description Logic in Practice: A CLASSIC Application. In *Proceedings of the 14th IJCAI*, Montreal, Canada, August 1995. An extended version of this paper is available from the first author.

[92] D. L. McGuinness and L. Alperin Resnick. Description Logic-based Configuration for Consumers. In *Proceedings of the International Description Logic Workshop*, Rome, Italy, 1995.

[93] T. Mitchell, J. Cheng, D. Freitag, J. Jourdan, J. Schlimmer, S. Thrun, N. Watanabe, T. Yamanouchi, and D. Zabowski THEO: A Framework for Problem Solving and Learning. available from http://www.cs.cmu.edu/afs/cs.cmu.edu/user/stork/mosaic/theo-dist.html, 1995.

[94] B. Nebel. Computational Complexity of Terminologic Reasoning in BACK. In *Aritficial Intelligence 34(3)*, April 1988.

[95] B. Nebel, C. Peltason, and K. von Luck, editors. *International Workshop on Terminological Logics.* German Research Center for Artificial Intelligence (DFKI), May 1991. Document D-91-13.

[96] R. Neches, W. R. Swartout and J. Moore. Explainable (and Maintainable) Expert Systems. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA., 1985, pp. 382–389.

[97] E. Paek. A justification-based theory of explanation. PhD Thesis, Stanford University, 1991.

[98] P. F. Patel-Schneider, D. L. McGuinness, R. J. Brachman, L. Alperin Resnick, and A. Borgida. The CLASSIC knowledge representation system: Guiding principles and implementation rationale. *SIGART Bulletin*, 2(3):108–113, June 1991. A preliminary version available in the *Working Notes of the AAAI Symposium on Implemented Knowledge Representation and Reasoning Systems*, American Association for Artificial Intelligence, March 1991, pages 202–214.

[99] P. F. Patel-Schneider, D. L. McGuinness, M. K. Herman, L. Alperin Resnick, and E. Weixelbaum. Description Logic-based Knowledge Representation. In *Proceedings of the International Description Logic Workshop* 1995.

[100] P. F. Patel-Schneider, B. Owsnicki-Klewe, A. Kobsa, N. Guarino, R. MacGregor, W. S. Mark, D. L. McGuinness, B. Nebel, A. Schmiedel, and J. Yen. Term subsumption languages in knowledge representation. *AI Magazine*, 11(2):16–22, Summer 1990.

[101] P. F. Patel-Schneider and W. Swartout Description-Logic Knowledge Representation System Specification. AI Principles Research Department, AT&T Bell Laboratories, 1993.

[102] S. Reeves and M. Clarke. Logic for Computer Science. Addison Wesley, 1990.

[103] L. Alperin Resnick, A. Borgida, R.J. Brachman, D.L. McGuinness, P.F. Patel-Schneider, C. Isbell, and K.Zalondek. CLASSIC description and reference manual for the Common Lisp implementation: Version 2.3. AI Principles Research Department, AT&T Bell Laboratories. 1995.

[104] G. Robbins. The NIKL Manual. The Knowledge Representation Project, Information Sciences Institute, University of Southern California, 1986.

[105] M.-C. Rousset. Knowledge Formal Specifications for Formal Verification : A Proposal based on the Integration of Different Logical Formalisms. In *Proceedings of ECAI-94*, 1994.

[106] V. Royer and J. Quantz. Deriving inference rules for terminological logics. In *Logics in AI, Proceedings of JELIA'92*, D. Pearce, G.Wegner (eds), Springer Verlag, 1992, pp.84–105.

[107] M. Schmidt-Schauss and G. Smolka. Attributive concept descriptions with complements. In *Artificial Intelligence*, 48:1–26, 1991.

[108] P.G. Selfridge. Knowledge Representation Support for a Software Information System. In *Proceedings of the 7th IEEE Conference on Artificial Intelligence Applications*, pp. 134–140, February, 1991.

[109] A. Sheth, S. Gala, and S. Navathe, "On automatic reasoning for schema integration", *Int. J. of Intelligent and Cooperative Information Systems*, 2(1), pp. 23–50, 1993.

[110] O. Shmueli and S. Tsur. Logical Diagnosis of LDL Programs. In *New Generation Computing*, OHMSHA, LTD and Springer-Verlag, Volume 9, pp. 277–303, June 1991. Also in *Proceedings of the Seventh International Conference on Logic Programming*, 1990.

[111] E.H. Shortliffe. Computer-Based Medical Consultations: MYCIN. Elsevier, 1976.

[112] John Sowa, editor. *Principles of Semantic Networks: Explorations in the representation of knowledge.* Morgan-Kaufmann, San Mateo, California, 1991.

[113] L. Sterling and E. Shapiro. The Art of Prolog: Advanced Programming Techniques. The MIT Press, 1986.

[114] H.J. Suermondt. Explanation in bayesian belief networks. PhD Thesis, Stanford University, 1992.

[115] W. R. Swartout. XPLAIN: A system for creating and explaining expert consulting systems. In *Artificial Intelligence*, 21(3):285–325, September, 1983.

[116] W. R. Swartout and Y. Gil. EXPECT: Explicit Representations for Flexible Acquisition. In *Proceedings fo teh Ninth Knowledge Acquisition for Knowledge-Based Systems Workshop* February 26- March 3, 1995, Banff, Alberta, Canada.

[117] 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.

[118] G. Teege. Making the Difference: A Subtraction Operation for Description Logics. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourth International Conference*, Bonn, Germany. May, 1994. pp. 540–550.

[119] P.E. van der Vet, P.H. Speel, and N.J.I. Mars. Ontologies for very large knowledge bases in materials science: a case study. In *Towards Large Knowledge Bases: Proceedings of the Second International Conference on building and Sharing Very Large-Scale Knowledge Bases KB & KS'95*, Nicolaas J.I. Mars, ed., IOS Press, Amsterdam, the Netherlands, 1995.

[120] von Luck, K., Nebel, B., Peltason, C., and Schmiedel, A., The anatomy of the BACK System, *Techniche Universität Berlin, KIT-Report* No. 41., January 1987.

[121] A. Walker, M. McCord, J.F. Sowa, and W.G. Wilson. Knowledge Systems and Prolog: developing expert, database, and natural language systems, Second Edition. Adddison-Wesley, 1990.

[122] D. Waterman, J. Paul, B. Florman, and J.R. Kipps. An Explanation Facility for the ROSIE Knowledge Engineering Language. The Rand Corporation, Prepared for the Defense Advanced Research Projects Agency, 1986.

[123] R. Weida. Closed Terminologies in Description Logic In *Proceedings of the National Conference on Artificial Intelligence*, Portland, Oregon, 1996.

[124] J.L. Weiner BLAH, A system which explains its reasoning. In *Artificial Intelligence*, 15:19–48, 1980.

[125] E.Weixelbaum. C-CLASSIC reference manual: release 1.3. Software Technology Center, AT&T Bell Laboratories. 1993.

[126] M.R. Wick and W.B. Thompson. Reconstructive expert system explanation. In *Artificial Intelligence*, 54:33–70, 1992.

[127] W. A. Woods and J. G. Schmolze. The KL-ONE family. *Computers and Mathematics With Applications.* 23(2-5), March 1992.

[128] J. R. Wright, D. L. McGuinness, and C. Foster. Conceptual modeling for configurators. In *Proceedings of AT&T Human Factors Days*, March 1995.

[129] J. R. Wright, D. L. McGuinness, C. Foster, and G. T. Vesonder. Conceptual Modeling using Knowledge Representation: Configurator Applications. In *Proceedings of the Workshop on Artificial Intelligence in Distributed Information Networks, IJCAI-95*, Montreal, 1995.

[130] 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. A version of this paper also appears in *AI Magazine*, 1993, pp. 69-80.

[131] The *AAAI Spring Symposium on Cooperative Explanation*, 1992.

[132] The AAAI Workshop on Explanation. Eighth National Conference on Artificial Intelligence, Boston, MA., 1990.

[133] The Eleventh Annual IEEE Smposium on Logic in Computer Science. New Brunswick, NJ, July 27–30, 1996.

Vita

Deborah L. McGuinness

Degrees and Awards

1976 Valedictorian of Springfield High School, Springfield, Pennsylvania.

1980 B.S. in Computer Science from Duke University. Also completed the requirements for a major in Mathematics. Summa Cum Laude.

1981 M.S. in Computer Science, University of California at Berkeley.

1996 Ph.D. in Computer Science from Rutgers University.

Work History

1980-present AT&T Bell Laboratories (now AT&T Labs-Research). 1980-84: Home Information Systems Lab; 1984-85: AI and Computing Environment Research. 1985-present: AI Principles Research. 1996-present: Supervisor Emerging Technologies and Opportunities for AT&T Personal Online Services. (Dual with research position.)

Selected Publications

1988 Brachman and McGuinness. Knowledge representation, connectionism, and conceptual retrieval. *SIGIR88.*

1989 Borgida, Brachman, McGuinness, and Resnick. CLASSIC: A structural data model for objects. *SIGMOD89.*

1991 Brachman, McGuinness, Patel-Schneider, Resnick, and Borgida. Living with CLASSIC: When and how to use a KL-ONE-like language. In Sowa, *Principles of Semantic Networks* Morgan-Kaufman.

1993 Brachman, Selfridge, Terveen, Altman, Borgida, Halper, Kirk, Lazar, McGuinness, and Resnick. Integrated support for data archaeology. *International Journal of Intelligent and Cooperative Information Systems*, 2(2).

1995a McGuinness and Borgida. Explaining subsumption in description logics. In *IJCAI95.*

1995b McGuinness, Resnick, and Isbell. Description logic in practice: A CLASSIC: application. *IJCAI95.*

1996 Borgida and McGuinness. Asking queries about frames. *KR96.*