7

From Description Logic Provers to Knowledge Representation Systems

Deborah L. McGuinness

Peter F. Patel-Schneider

Abstract

A description-logic based knowledge representation system is more than an inference engine for a particular description logic. A knowledge representation system must provide a number of services to human users, including presentation of the information stored in the system in a manner palatable to users and justification of the inferences performed by the system. If human users cannot understand what the system is doing, then the development of knowledge bases is made much more difficult or even impossible. A knowledge representation system must also provide a number of services to application programs, including access to the basic information stored in the system but also including access to the machinations of the system. If programs cannot easily access and manipulate the information stored in the system, then the development of applications is made much more difficult or even impossible.

7.1 Introduction

A description logic-based knowledge representation system does not live in a vacuum. It has to be prepared to interact with several sorts of other entities. One class of entities consists of human users who develop knowledge bases using the system. If the system cannot effectively interact with these users then it will be difficult to create knowledge bases in the system, and the system will not be used. Another class of entities consists of programs that use the services of the system to provide information to support applications. If the system cannot effectively interact with these programs then it will be difficult to create applications using the system, and the system will not be used.

However, before one can talk about effective interaction, there has to be basic interaction between the knowledge representation system and applications or users. This basic interaction has to do with the mechanics of telling information to the system and retrieving information from it. At this level the system just maintains what is was told and responds to the queries by running an inference procedure for the logic it implements.

The basic interface is not sufficient for effective access to the system. On the application side there is need for a treatment of exceptional conditions, wider interface to applications, remote interfaces, and concurrent access, among others. There is also need for responsive reaction by the system. On the human side there is need for better presentation of the results of queries, particularly the suppression of irrelevant detail; explanation of the inferences performed by the system; better support for the creation of large description logic knowledge bases, particularly by several people working in collaboration.

Even if all the above are present in a system, it will still not be complete. There is also a need to have effective information about the system widely available. This information has to be in various forms, including the obvious user manuals, but also including interactive tutorials and demonstration system.

A system that does not include all of the above services is not a complete knowledge representation system.

Our discussion of the services that need to be provided will mostly be described in terms of an arbitrary description logic knowledge representation system. However, some of our examples will be given in the context of the CLASSIC family of knowledge representation systems developed at AT&T [Borgida *et al.*, 1989; Brachman *et al.*, 1991; Patel-Schneider *et al.*, 1991], as CLASSIC has had the longest lived and most extensive industrial application history of any description logic knowledge representation system. The CLASSIC application that we will refer to the most is the configuration of transmissions equipment—an application developed within AT&T [Wright *et al.*, 1993; McGuinness *et al.*, 1995; McGuinness and Wright, 1998b; McGuinness *et al.*, 1998].

In a typical configuration problem, a user is interested in entering a small number of constraints and obtaining a complete, correct, and consistent parts list. Given a configuration application's domain knowledge and the base description logic inference system, the application can determine if the user's constraints are consistent. It can then calculate the deductive closure of the user-stated knowledge and the background domain knowledge to generate a more complete description of the final parts list. For example, in a home theater demonstration configuration system [McGuinness *et al.*, 1995], user input is solicited on the quality a user is willing to pay for and the typical use (audio only, home theater only, or combination), and then the application deduces all applicable consequences. This typically generates descriptions for 6–20 subcomponents which restrict properties such as price range, television diagonal, power rating, etc. A user might then inspect any of the individual components possibly adding further requirements to it which may, in turn, cause further constraints to appear on other components of the system. Also, a user may ask the system to "complete" the configuration task, completely specifying each component so that a parts list is generated and an order may be completed.

This home theater configurator example is fairly simple but it is motivated by real world application uses in configuring very large pieces of transmission equipment where objects may have thousands of parts and subparts and one decision can easily have hundreds of ramifications. It was complicated applications such as these that drove our work on access to information. More information can be found on description logics for configuration in in this book in Chapter 12. Another example application that drove our work on information access and presentation needs was a simple description logic backend system supporting knowledge-enhanced search for the web called FINDUR [McGuinness, 1998; McGuinness *et al.*, 1997] which is also described in Chapter 14.

7.2 Basic access

Basic access to a description logic knowledge base consists of simple mechanisms to create description logic knowledge bases and to query them. The foundational aspects of this basic interaction have been well-studied. For example, Levesque [1984] proposed that the basic interface to any knowledge representation system consist of two kinds of interactions—one to *tell* information to the system and one to *ask* whether information follows from what was previously told to the system.

Many frame-oriented knowledge representation systems embody such distinctions, such as the Generic Frame Protocol [Chaudhri *et al.*, 1997], and OKBC (Open Knowledge Base Connectivity) [Chaudhri *et al.*, 1998a]. In the description logic community, this basic interaction was standardized into an interface specification that defined a number of tell and ask operations that a description logic knowledge representation system should implement [Patel-Schneider and Swartout, 1993].[1] This specification is commonly known as the KRSS specification. The description of a minimal description logic knowledge representation system interface given here will generally follow this KRSS specification. The KRSS specification incorporates the DFKI standardized syntax and semantics [Baader *et al.*, 1991]. Examples given here follow the syntax of Chapter 2, for the abstract syntax, and the syntax of KRSS for a LISP-like syntax that can actually be used from within a computer.

One problem with defining a tell-and-ask interface for a description logic knowledge representation system is that even a minimal interface depends on the expres-

[1] The KRSS specification also incorporates a number of operations that fall under the advanced interface that will be discussed later.

Table 7.1. *Syntax and semantics of making definitions.*

Program Syntax	**Abstract Syntax**	**Semantics**
`(define-concept CN C)`	$CN \equiv C$	$CN^{\mathcal{I}} = C^{\mathcal{I}}$
`(define-primitive-concept CN C)`	$CN \sqsubseteq C$	$CN^{\mathcal{I}} \subseteq C^{\mathcal{I}}$
`(define-role RN R)`	$RN \equiv R$	$RN^{\mathcal{I}} = R^{\mathcal{I}}$
`(define-primitive-role RN R)`	$RN \sqsubseteq R$	$RN^{\mathcal{I}} \subseteq R^{\mathcal{I}}$
`(define-attribute AN A)`	$AN \equiv A$	$AN^{\mathcal{I}} = A^{\mathcal{I}}$
`(define-primitive-attribute AN R)`	$AN \sqsubseteq R$	$AN^{\mathcal{I}} \subseteq R^{\mathcal{I}}$

Table 7.2. *Inclusion syntax and semantics.*

Program Syntax	**Abstract Syntax**	**Semantics**
`(included C D)`	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$

sive power of the logic. As an example, if the description logic implemented by the system does not include individuals then of course there is no need to include any facilities for making statements about individuals. To overcome this difficulty this chapter will describe the interfaces required for a system that implements a typical description logic with both concepts and individuals.

Such a system has to have a method for creating a terminology of concepts. A syntax for creating such a terminology, taken directly from the KRSS specification, is given in Table 7.1. A terminological knowledge base, or TBox, is then a set of such definitions perhaps with the condition that every concept, role, and attribute name has at most one definition. There may also be the side condition that there are no recursive definitions.

Some representation systems may have other definitions allowable or other restrictions. For example, some systems allow the definition of transitive roles, via a define-transitive-role definition. Other systems prohibit non-primitive roles.

If the underlying description logic allows for recursive definitions, then it may be easier to provide an even more basic interface to define concepts. Table 7.2 shows a minimal interface for a system that employs arbitrary concept inclusions as its means of defining concepts.

If the system incorporates individual reasoning, then it has to have a mechanism for adding information about these individuals. One such method is via the assertions in Table 7.3. An assertional knowledge base, or ABox, is then a set of such assertions.

Once information has been told to the system, there has to be a mechanism for

Table 7.3. *Assertion syntax and semantics.*

Program Syntax	Abstract Syntax	Semantics
`(instance IN C)`	$IN \in C$	$IN^{\mathcal{I}} \in C^{\mathcal{I}}$
`(related IN I R)`	$\langle IN, I \rangle \in R$	$\langle IN^{\mathcal{I}}, I^{\mathcal{I}} \rangle \in R^{\mathcal{I}}$

Table 7.4. *Query syntax and semantics.*

Query	Meaning
`(concept-subsumes? C1 C2)`	$C_1 \sqsubseteq C_2$
`(role-subsumes? R1 R2)`	$R_1 \sqsubseteq R_2$
`(individual-instance? IN C)`	$IN \in C$
`(individual-related? IN I R)`	$\langle IN, I \rangle \in R$

determining what follows from this information. A minimal mechanism for this is via a set of queries, such as those given in Table 7.4. A query is answered by the system by determining if the meaning of the query is implied by the information that has been told to the system.

The interface described above is sufficient for determining the contents of a knowledge base but only in the theoretical sense. For reasonable access to the information in a knowledge base a richer interface is required. One part of this richer access even really belongs in the basic interface, namely retrievals of taxonomy information. The interface in Table 7.5 provides a simple interface to the taxonomy information implicit in a description logic knowledge base. The meaning of the calls should be obvious from their description, except perhaps the "`-direct-`" versions, which

Table 7.5. *Taxonomy retrieval syntax.*

```
(concept-descendants C)
(concept-offspring C)
(concept-ancestors C)
(concept-parents C)
(concept-instances C)
(concept-direct-instances C)
(role-descendants R)
(role-offspring R)
(role-ancestors R)
(role-parents R)
(individual-types IN)
(individual-direct-types IN)
(individual-fillers IN R)
```

Table 7.6. *UnTell syntax.*

```
(undefine-concept CN)
(undefine-role RN)
(undefine-attribute AN)
(un-tell-instance IN C)
(un-tell-related IN I R)
```

return the concepts, individuals, or roles that are directly related to the query, i.e., that have no intervening concept or role.

Another basic service that is missing from above interface is the ability to remove information from the knowledge base. This is not the ability to perform arbitrary changes to the implicit information represented by the knowledge base. Instead it is just the ability to "un-tell" information that had been previously told to the system. A basic interface for this purpose is given in Table 7.6. There may be restrictions on what can be un-told, such as requiring that concepts that are currently mentioned in the definition of other concepts cannot be removed from the knowledge base.

7.3 Advanced application access

The basic interface described above provides only minimal access to a description logic knowledge base. Effective access requires a number of augmentations to the basic interface.

One of the most important augmentations has to do with defining a complete application programming interface. The basic interface assumes that the system is implemented in a language like LISP, where there is a simple way of creating descriptions and other values for the various operations and there is a mechanism for returning values of any type. This was acceptable when systems and applications were all implemented in LISP, but this is no longer the case.

A complete application programming interface must then provide a syntax for creating all the types of values that need to be passed to the representation system. Further, it needs to provide or define mechanisms for returning values, particularly compound values such as the sets of concepts that are returned by the taxonomic retrieval operations.

7.3.1 Efficiency

Because the operations of the representation system may represent the largest resource consumption of an application, it is often necessary to know how expensive various operations of the system may be. For example, it is often necessary to know the usual resource consumption of the most-frequently called operations of the knowledge representation system or those operations that are called at critical time in the operation of the whole system.

The CLASSIC family has been particularly aggressive in ensuring that queries to the system are fast, working under the assumption that the most-common operations are queries. Most queries in CLASSIC are simply retrievals of data stored by the system, as CLASSIC responds to the addition of knowledge by computing most of its consequences. Further, the performance of the addition of knowledge to the system is optimized over the retraction or change of knowledge.

CLASSIC achieves these characteristics of fastest queries, fast additions, and slower retractions and changes by retaining data structures that record the current set of consequences and also record, on a fairly granular level, which knowledge affects other knowledge. This is not full truth-maintenance data, which would be prohibitively expensive to compute (and store), but is just enough to make additions cheap. It also serves to make retractions and changes somewhat cheaper than they otherwise would be, but this effect is much less than the change in the speed up additions of knowledge.

7.3.2 Wide application programming interface

In the vast majority of applications, the knowledge representation system has to serve as a tightly integrated component of a much larger overall system. For this to be workable, the knowledge representation system must provide a full-featured interface for the use of the rest of the system.

The NEOCLASSIC system, which is programmed in C++, and is designed to be part of a larger C++program, provides a very wide application programming interface. In addition to the above interface, there is a large interface that lets the rest of the system receive and process the actual data structures used inside NEOCLASSIC to represent knowledge, but without allowing these structures to be modified outside of NEOCLASSIC.[1] This interface allows for much faster access to the knowledge stored by NEOCLASSIC, as many accesses just retrieve fields from a data structure. Further, direct access to data structures allows the rest of the system to keep track of knowledge from NEOCLASSIC without having to keep track of a "name" for the knowledge querying using this name. (In fact, it is in this way possible to dispense with any notion of querying by name.)

There are also ways to obtain the data structures that are used by NEOCLASSIC for other purposes, including explanation. We have used this facility to write graphical user interfaces to present explanations and other information.

An additional interface that is provided by both LISP CLASSIC and NEOCLASSIC

[1] Of course, as C++does not have an inviolable type system, there are mechanisms to modify these structures. It is just that any well-typed access cannot.

is a notification mechanism, or hooks. This mechanism allows programmers to write functions that are called when particular changes are made in the knowledge stored in the system or when the system infers new knowledge from other knowledge. Hooks for the retraction of knowledge from the system are also provided. These hooks allow, among other things, the creation of a graphical user interface that mirrors (some portion or view of) the knowledge stored in the representation system.

Others in the knowledge representation community have recognized the need for common APIs, (e.g., the Generic Frame Protocol [Chaudhri *et al.*, 1997] and the Open Knowledge Base Connectivity [Chaudhri *et al.*, 1998a]). Some systems embrace the notion of loading many different forms of knowledge bases and accept wrapper specifications for other source formats and APIs. For example, ONTOLINGUA has implemented capability for loading a number of formats including CLASSIC, OKBC, ANSI KIF, KIF 3.0, CML, CLIPS, ONTOLINGUA, PROTÉGÉ, SNARK, and DAML+OIL. It also provides the ability to dump frames in multiple formats such as OKBC, CLASSIC, CLOS, CML, ONTOLINGUA, and DAML+OIL and it has also been made interoperable with at least two reasoners including one in lisp and one in java.

7.3.3 Remote and concurrent access

The standard computing environment is becoming more and more distributed. If a description logic knowledge representation system is to be part of this environment it must allow effective remote access. There are several mechanisms for allowing remote access, including applications that run on the same machine as the description logic knowledge representation system but themselves provide a remote access mechanism. Examples of such applications are the wines [Brachman *et al.*, 1991] and stereo configuration demonstration systems [McGuinness *et al.*, 1995] mentioned later in this chapter.

The description logic knowledge representation system itself can also directly provide a remote access mechanism. This can be as simple as providing the system with a pipe-like interface where clients can send a sequence of commands to the system from remote machines, and receive responses via the same pipe. NEOCLASSIC provides this sort of simple remote access mechanism.

A more complicated remote access mechanism would be to provide a CORBA interface to the system. This kind of access was proposed by Bechhofer *et al.* [1999], Their interface gives a CORBA layering around a tell-and-ask interface. Providing a wider CORBA access to description logic knowledge representation systems, such as providing CORBA access to the actual data structures of the system, is more difficult, as the CORBA mechanism for dealing with recursive objects is annoying. Nevertheless, an effective remote access mechanism should provide the same functionality as is desired for local access.

If remote access to a description logic knowledge representation system is provided, then the issue of concurrent access becomes vital. (This is not to say that concurrent access is not of interest if the system does not allow remote access.) The interesting issues with respect to concurrent access involve simultaneous access to the same repository of knowledge. Most of the issues with respect to concurrent access are the same as concurrent access to databases, including locking and providing transactions. In fact, there have been informal proposals to use a database system to store the information in a description logic knowledge representation system like CLASSIC just so as to piggyback on the facilities for concurrent access provided by the database system.

The remote interface proposal mentioned above provides a limited form of transactions, basically allowing clients to batch up a collection of updates to a knowledge base and apply them all at once as an atomic transaction. This interface, however, does not provide any mechanism to abort transactions or to provide a local view of the knowledge base during the execution of a transaction.

At least one other knowledge representation system has dealt with the notion of concurrent access by leveraging the notion of sessions. ONTOLINGUA allows users to log in to a particular session that may already be opened by a previous user. All users logged into the same session see the same version of the knowledge base. A more sophisticated approach to concurrent access and knowledge base editing is embodied in ONTOBUILDER [Das *et al.*, 2001]. In this system, users can not only do something similar to sharing a session, but the implementation also facilitates collaboration through dialogue with other users currently signed on to the same ontology and allows locking of concepts for updates.

7.3.4 Platforms

Another important access aspect concerns the platforms on which the knowledge representation system runs. This encompasses not only the machines and operating systems, but also the language in which the system is written (if it is visible), the version of the libraries that the system uses, and the mechanism for linking to the system. Many applications have needs for a particular operating system or language, and cannot utilize tools not available in this context.

Some description logics like CLASSIC have been made available on a reasonable number of platforms. The underlying language of a member of the CLASSIC family is visible, not just because of the application programming interface which is, of necessity, language-specific, but also because programmers can write functions to extended the expressive power of the system, and these functions have to be written in the underlying language of the system.

CLASSIC is currently available in two different languages: LISP and C++. The C++member is the more recent, and the reimplementation used C++precisely to make CLASSIC available for a larger number of applications. This was done even though C++is not the ideal language in which to write a representation system.

The members of the CLASSIC family have also been written in a platform-independent manner. This has required not using some of the nicer capabilities of the underlying language or of particular operating systems. For example, NEO-CLASSIC does not use C++exceptions, partly because few C++compilers supported this extension to the language. LISP-CLASSIC runs on various LISP implementations and on various operating systems, including most versions of Unix, MacOS, and Windows. NEOCLASSIC runs under four C++compilers and on both Unix and Windows NT.

With the influence of the web and more distributed development environments, it may be expected that more description logics may be made available on multiple platforms and may be integrated into more hybrid environments. One example of another knowledge representation system that found a need to do this is the CHIMAERA Ontology Evolution Environment [McGuinness *et al.*, 2000b]. This system has been connected to ONTOLINGUA for ontology editing and simple inference, a lisp-based reasoner for some diagnostics, and a hybrid java-based reasoning environment that supports both first order logic reasoning as well as special purpose reasoning for the DAML+OIL description logic.

7.4 Advanced human access

7.4.1 Explanation

Many research areas which focus on deductive systems (such as expert systems and theorem proving) have determined that explanation modules are required for even simple deductive systems to be usable by people other than their designers. Description Logics have at least as great a need for explanation as other deductive systems since they typically provide similar inferences to those found in other fields and also support added inferences particular to description logics. They provide a wide array of inferences [Borgida, 1992b] which can be strung together to provide complicated chains of inferences. Thus conclusions may be puzzling even to experts in description logics when application domains are unfamiliar or when chains of inference are long. Additionally, naive users may require explanations for deductions which may appear simple to knowledgeable users. Both sets of needs became evident in work on a family of configuration applications and necessitated an automatic explanation facility.

The main inference in description logics is subsumption—determining when membership in one class necessitates membership in another class. For example, Person is subsumed by Mammal since anything that is a member of the class Person must be a member of the class Mammal. Almost every inference in description logics can be rewritten using subsumption relationships and thus subsumption explanation forms the foundation of an explanation module [McGuinness and Borgida, 1995].

Although subsumption in most implemented description logics is calculated procedurally, it is preferable to provide a declarative presentation of the deductions because a procedural trace typically is very long and is littered with details of the implementation. A declarative explanation mechanism which relies on a proof theoretic representation of deductions may be used as a framework. Such a mechanism has been specified [McGuinness, 1996] and implemented for CLASSIC and later specified for $\mathcal{ALN}$ [Baader *et al.*, 1999a].

All the inferences in a description logic system can be represented declaratively by a proof rules which state some (optional) antecedent conditions and deduce some consequent relationship. The subsumption rules may be written so that they have a single subsumption relationship in the denominator. For example, if Person is subsumed by Mammal, then it follows that something that has all of its children restricted to be Persons must be subsumed by something that has all of its children restricted to be Mammals. This can be written more generally (with C representing Person, D representing Mammal, and R representing `child`) as the $\forall$ restriction rule below:

$$\text{All restriction} \qquad \frac{\vdash C \sqsubseteq D}{\vdash \forall R.C \sqsubseteq \forall R.D}$$

Using a set of proof rules that represent description logic inferences, it is possible to give a declarative explanation of subsumption conclusions in terms of proof rule applications and appropriate antecedent conditions. This basic foundation can be applied to all of the inferences in description logics, including all of the inferences for handling constraint propagation and other individual inferences. There is a wealth of techniques that one can employ to make this basic approach more manageable and meaningful for users [McGuinness and Borgida, 1995; McGuinness, 1996].

Expressive description logic-based systems may require a large number of proof rules. If one is interested in limiting both explanation implementation work and also limiting the size of explanations, it is be beneficial to prune the number of inferences to be explained. In one configuration family of applications [McGuinness and Wright, 1998b] the help desk logs were logged and analyzed to determine the most questions that related to explanation. These inferences included inheritance (if A is an instance of B and B is a subclass of C, then A "inherits" all the properties of C), propagation (if A fills a role R on B, and B is an instance of something which is known to restrict all of its fillers for the R role to be instances of D, then A is an instance of D), rule firing (if a is an instance of E and E has a rule associated with it that says that anything that is an E must also be an F, then a is an instance of F), and contradiction detection (e.g., I can not be an instance of something that has at least 3 children and at most 2 children). In the initial development version, explanation was only provided for these inferences in an effort to minimize development costs, resulting in a quite useful explanation mechanism with much less effort than a full explanation system. (The two current implementations of explanation in CLASSIC contain complete explanation.) One demonstration system [McGuinness *et al.*, 1995] incorporates special handling for the most heavily used inferences providing natural language templates for presentations of explanations aimed at lay people.

7.4.2 Error handling

Since one common usage of deductive systems is for contradiction detection, handling error reporting and explanation is critical to usability. This usage is common in applications where object descriptions can easily become over-constrained. For example, in the home theater system application, one could generate a non-contradictory request for a high quality stereo system that costs under a certain amount. The description could later become inconsistent as more information is added. For example, a required high-quality, expensive speaker set could violate a low total price constraint. Understanding evolving contradictions such as this challenges many users and leads them to request special error explanation support. Informal studies with internal users and external academic users indicate that adequate error support is crucial to the usability of the system.

Error handling could be viewed simply as a special case of inference where the conclusion is that some object is found to be described by the a special concept typically called bottom or nothing. For example, a concept is incoherent if it has conflicting bounds on some role:

$$\text{Bounds Conflict} \qquad \frac{\vdash C \sqsubseteq (\geqslant m\, r) \quad \vdash C \sqsubseteq (\leqslant n\, r) \quad n < m}{\vdash C \sqsubseteq \bot}$$

If an explanation system is already implemented to explain proof theoretic inference rules, then explaining error conditions is *almost* a special case of explaining any inference. There are two issues that are worth noting, however. The first is that information added to one object in the knowledge base may cause another object to become inconsistent. In fact, information about one object may impact another series of objects before a contradiction is discovered at some distant point along an inference chain. Typical description logic systems require consistent knowledge bases, thus whenever they discover a contradiction, they use some form of truth maintenance to revert to a consistent state of knowledge, removing conclusions that depend on the information removed from the knowledge base. Thus, it is possible, if not typical, for an error condition to depend upon some conclusion that was later removed. A simple minded explanation based solely on information that is currently in the knowledge base would not be able to refer to these removed conclusions. Thus, any explanation system capable of explaining errors will need access to the current state of the knowledge base as well as to its inconsistent state.

Because of the added complexity resulting from the distinction between the current (consistent) state and the inconsistent state of the knowledge base and because of the importance of error explanation, we believe system designers will want to support special handling of error conditions. For example, in a number of situations surveyed, users typically asked for explanations of a particular object property or relationships between objects. Under error conditions, users had more trouble identifying an appropriate query to ask. This suggests that special error support should be introduced. In CLASSIC, for example, an automatic error explanation option is generated upon contradiction detection. This way the user requires no knowledge (other than the explanation error command name) in order to ask for help.

Another issue of importance to error handling is the completeness or incompleteness of the system. If a system is incomplete then it may miss deductions. Thus, it is possible for an object to be inconsistent if all of the logically implied deductions were to be made but, because the system was incomplete, it missed some of these deductions and thus the object remains consistent in the knowledge base. In order for users to be able to use a system that is incomplete, they may need to be able to explain not only error deductions but deductions that were missed because of incomplete reasoning. An approach that completes the reasoning with respect to a particular aspect of an object is described in [McGuinness, 1996, Chapter 5]. Given the completed information, the system can then explain missed deductions.

7.4.3 Pruning

If a knowledge representation system makes it easy to generate and reason with complicated objects, users may find naive object presentations to be much too complex to handle. In order to make a system more usable, there needs to be some way of limiting the amount of information presented about complicated objects. For example, in the stereo demonstration application, a typical stereo system description may generate four pages of printout. The information contained in the description may be clearly meaningful information such as price ranges and model numbers for components but it may also contain descriptions of where the component might be displayed in the rack and which superconcepts are related to the object. In certain contexts it is desirable to print just model numbers and prices, and in other contexts it is desirable to print price ranges of components. We believe it is critical to provide support for encoding domain independent and domain dependent information which can be used along with contextual information to determine what information to print or explain. As one example, we consider some of the knowledge bases written for the DARPA High Performance Knowledge Base project. This project includes a very general upper level ontology with many slots defined on many of the classes. Most objects in the system inherit a large number of slots from upper ontology classes and it is not uncommon for normalized objects to have hundreds of slots associated with them even though they only have a couple of properties defined on them in the local knowledge bases.

Knowledge representation systems faced with information overload need to take some approach to filtering. One of the simplest approaches allows a specification on roles concerning whether they should be displayed on objects or not. This may work for homogeneous knowledge bases where role information is uniformly interesting or uninteresting. Our experience is however, that context needs to be taken into account in more heterogeneous knowledge base applications. One example implementation that allows context and domain dependent information to be considered along with domain independent information is implemented in CLASSIC. A meta language is defined for describing what is interesting to either print or explain on a class by class basis. Any subclass or instance of the class will then inherit the meta description and thus will inherit "interestingness" properties from its parent classes. The meta language essentially captures the expressive power of the base description logic with some carefully chosen epistemic operators to allow contextual information (such as known fillers or closed roles) to impact decisions on what to print.

The meta language has been used to reduce object presentation and explanation by an order of magnitude in at least one application [McGuinness *et al.*, 1995]. This reduction was required for the application to be able to include object presentation. The algorithms of the basic approach are included in [McGuinness, 1996], the theory of a generalized approach are presented in [Borgida and McGuinness, 1996] and further analyzed in [Baader *et al.*, 1999a].

7.4.4 Knowledge acquisition

If an application is expected to have a long life-cycle, then acquisition and maintenance of knowledge become major issues for usability. There are two kinds of knowledge acquisition which are worth considering: (i) acquisition of additional knowledge once a knowledge base is in place, and (ii) acquisition of original domain knowledge. A complete environment will address both concerns, however the original acquisition of knowledge is a much more general and difficult problem and conveniently enough, is not the activity that many users will find themselves doing repeatedly while maintaining a project.

We observe that with knowledge of the domain and appropriate analysis of evolution, it is possible to build a knowledge evolution environment suitable for non-experts to use for extending knowledge bases. One such project considered the evolution support environment for configurators. The specific domain and usage patterns were analyzed, and it was found that only certain classes had new subclasses added to them as product knowledge evolved. It was also found that instances were typically populated in particular patterns. A special purpose interface was developed for a family of configurators that exploited these findings and supported new configurator application development by non-experts [McGuinness and Wright, 1998b]. Also, in related work, Gil and Melz [1996] have analyzed planning-based uses of another description logic-based system that systematically supports knowledge base evolution with respect to the known plan usage.

A more general problem that does not rely on domain or reasoning knowledge has been addressed in the editor work [Paley *et al.*, 1997] for the general frame protocol and also in editor work for collaborative generation and maintenance of ontologies by non experts in the Collaborative Topic Builder component of FINDUR [McGuinness, 1998] and recently in CHIMAERA work [McGuinness *et al.*, 2000b] for merging, analyzing, and maintaining ontologies. The general work, of course, is broader yet shallower with respect to reasoning implications. In the FINDUR collaborative topic builder environment, simple hierarchies of node names (with role filler and value restriction information) is used to support query expansion to provide more intelligent web searching. In order to deploy this broadly, a web-based distributed ontoloty editor was required to allow non-experts to input, modify, and maintain background ontologies. The basic functionality for this interface follows the same requirements specified in Section 7.2 although this particular implementation limited some of the interface specifications according to expected usage patterns. For example, in the medical deployments [McGuinness, 1999] of FINDUR, it was expected that all of the roles that were to be used had been defined and thus pull down lists of these roles were hardcoded into the interface and new role specification was not one of the exposed functionalities in the GUI. It also allows importing of seed ontologies and supports contradiction detection from ontology input. CHIMAERA's environment takes the analysis task to a much more detailed level and it provides a number of different ways of not only detecting explicit contradictions but also possible contradictions and possible term merges.

7.5 Other technical concerns

The computer science concerns that affect the suitability of a knowledge representation system have to do with the behavior of the system as a computer program or routine, ignoring its status as a representer of knowledge. The most-studied aspect of this collection of concerns has to do with the computational analysis of the basic algorithms embodied in the system, in particular their worst-case complexity. Because this worst-case complexity has been so well studied, we will not say anything about it further, except to state that it *is* important in determining the suitability of a knowledge representation system for particular task, notably tasks that need a performance *guarantee*.

7.6 Public relations concerns

Researchers sometimes underestimate the varied public relations aspects involved with making a system usable. Barriers to usability come in many forms: potential users who are unaware of a system's existence will not use it; potential users who do not understand how a system can meet the users needs are unlikely to use it; potential users who do not have enough understanding to visualize an abstract solution to their problem using a new system are unlikely to depend on the new system over tools they understand and can predict; and finally potential users who have a limited set of approved tools which does not include the new system are unlikely go to the effort of getting the new system approved for their internal use. In order to address these issues, description logic system designers need to devise ways to make their systems known to likely users, educate those users about the possible uses, provide support for teaching users how to use them for some standard and leverageable uses, and either obtain approval for their systems or provide ammunition for users to gain approval.

In experiences with CLASSIC, the following tools have been employed to overcome the above stated barriers to usability.

Beyond the standard research papers, users demand usage guidelines aimed at non-PhD researchers. A paper that provides a running (executable) example on how to use the system is most desirable, such as [Brachman *et al.*, 1991]. This paper also tries to provide guidance on when a description logic-based system might be useful, what its limitations are, and how one might go about using one in a simple application. A take off of that paper was done as the basis of a tutorial on building ontologies in other knowledge representation systems including PROTÉGÉ and ONTOLINGUA [Noy and McGuinness, 2000].

A demonstration system is also of great utility as it helps users understand a simple reasoning paradigm and provides a prototyping domain for showing off novel functionality which exploits the strengths of the underlying system. In the CLASSIC project a number of demonstration systems were developed, including a simple application that captures "typical" reasoning patterns in an accessible domain. This one system has been used in dozens of universities as a pedagogical tool and test system. While this application was appropriate for many students, an application more closely resembling some actual applications was needed to (i) give more meaningful demonstrations internally and to (ii) provide concrete suggestions of new functionality that developers might consider using in their applications. This led to a more complex application with a fairly serious graphical interface [McGuinness *et al.*, 1995]. Both of these applications have been adapted for the web.[1] It was only when a demonstration system that was clearly isomorphic to the developer's applications was available that there could be effective providing of clear descriptions and implemented examples of the functionality that we believed should be incorporated into development applications.

Interactive courses are also of benefit in training potential users in how to use a description-logic based knowledge representation system. Several courses [McGuinness *et al.*, 1994; Abrahams *et al.*, 1996] on how to use CLASSIC have been developed, including one from a university for course use, which includes a set of five running assignments to help students gain experience using the system. Other general description logic courses can be found on the Description Logic web site at `http://www.dl.kr.org/`.

For a system to be used in the business community, it has to satisfy their demand for common standard implementation languages, reasonable support, and standard platform toolkits. Some description logic implementations, such as CLASSIC, attempted to meet this need by providing an implementation in C while still maintaining the lisp research version. This later proved problematic to maintain and the decision was made to provide an implementation in C++that was to meet both developers and implementers needs. Interestingly enough, years later though the lisp version is the one that appears to be most heavily used. More details of the evolution of that of usability of that system can be found in [Brachman *et al.*, 1999].

7.7 Summary

Although a knowledge representation system must have sufficient expressive power and appropriate computational complexity to be considered for use in applications, there are many other issues that also determine whether it will be used. These issues involve access to the knowledge stored in the system, such as explanation and presentation of the knowledge, other technical issues, such as efficiency and

[1] The web version of the wines demonstration system was provided by Chris Welty and is available at `http://untangle.cs.vassar.edu/wine-demo/index.html`.

programming interfaces, and non-technical issues, such as publicity and demos. If these issues are not addressed appropriately, a knowledge representation system will not be used in real applications.