Usability Issues in Knowledge Representation Systems

Deborah L. McGuinness
AT&T Labs—Research
180 Park Avenue
Florham Park, NJ 07932
dlm@research.att.com

Peter F. Patel-Schneider
Bell Labs Research
600 Mountain Avenue
Murray Hill, NJ 07974
pfps@research.bell-labs.com

Abstract

The amount of use a knowledge representation system receives depends on more than just the theoretical suitability of the system. Some critical determiners of usage have to do with issues related to the representation formalism of the system, some have to do with non-representational issues of the system itself, and some might be most appropriately labeled public relations. We rely on over eight years of industrial application experiences using a particular family of knowledge representation systems based on description logics to identify and describe usability issues that were mandatory for our application successes.

Introduction

Determining whether a knowledge representation system is suitable for a particular use and, in fact, is used is driven by a number of important issues including the standard issues of expressive adequacy and computational complexity and more general issues concerning usability. We believe that while expressivity and computational aspects are critical to application success, usability issues play an equally important role in determining whether a knowledge representation system is suitable for a particular use, or, indeed, suitable for any use at all. If a knowledge representation system does not "sweat these details", it will not be used in most domains.

The usability issues that we will consider in this paper fall into three general categories. The first group has to do with the access to the knowledge in the system. One element of this access is the standard interface to the system, where new information is told to the system and queries are asked of it. Some foundational aspects of this have been well studied, for example in work by Levesque (Levesque 1982). In our work, we focus on the issues concerning the acquisition of knowledge for the system, the presentation of the results of queries, the explanation of these results, and recovery from inconsistent states of knowledge. Although there has been some work on these issues, they are much less studied than the standard interface and, arguably, no implemented system has adequate support in all of these areas.

The second group encompasses the general technical, but non-representational aspects, of the system. One element of this group is the theoretical running time of the algorithms used to process updates and queries. Again, this element has been well-studied, particularly in description logics where there are many papers describing algorithms for description-logic inference and their theoretical running times (Buchheit, Donini, & Schaerf 1993; Donini *et al.* 1991). Other non-representational aspects include the actual response time to updates and queries experienced, the programming interface that the system presents, and the prosaic, but important, issue of which platforms the system runs on. Again, these issues have received much less study than the theoretical running times of the algorithms. (One study of actual response time, and the improvement thereof, was performed by Baader *et al*. (1992).)

This paper would not be complete without mentioning the most vital issues having to do with how much the system is used. These issues are not typically considered part of the field of knowledge representation, and are thus somewhat outside the scope of a technical paper. The issues referred to here are, of course, advertising and approvals. If a system is unknown or not understood then it will not be used, and if a system is not on the list of approved systems for a particular organization then it will not be used in that organization.

We will describe and analyze these three groups of issues with respect to the usability of knowledge representation systems. This will be done 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). This family is based on expressively-limited description logics (also called terminological logics) (Baader *et al.* 1991); expressively-limited to ensure good computational properties, description-logic-based because of the desirable representational properties of description logics. There are two currently-supported members of the family, LISP CLASSIC (Resnick *et al.* 1995) and NEOCLASSIC (Patel-Schneider *et al.* 1997), and one older member C-CLASSIC (Weixelbaum 1991). We will be referring to characteristics of both of the current members in this paper, as currently neither dominates the other in the characteristics we are concerned with here.

Our experiences with CLASSIC may be of interest to the community for two main reasons. First, CLASSIC is the implemented system most similar to the description logic community-generated specification for description logics(Patil *et al.* 1992) and thus provides a representative basis for description logics. Second, arguably, CLASSIC has had the longest lived and most extensive industrial application history of any description logic-based system.

The CLASSIC family, and Description Logics in general, have been used in a number of classes of applications. We will mention one here as a motivational example for the some of the concerns we discuss. This class of applications has to do with configuration (Wright *et al.* 1993; Rychtyckyj 1996; McGuinness, Resnick, & Isbell 1995; McGuinness & Wright 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, Resnick, & Isbell 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.

Some of the above mentioned usability issues have been discussed in separate papers, and they have played at prominent role in at least one major presentation (Brachman 1992), but there are new issues and new insights in this paper, springing from five years of additional experience and evolution of the CLASSIC family of applications. If some of these issues had not been addressed, the family of applications would have been discontinued commercially. This paper brings together these issues in one place and directly comments on their role in the usability of knowledge representation systems.

Knowledge Access Concerns

The first group of concerns addresses access to the knowledge in the system. This is not the basic "tell-and-ask" access, but instead is access to other knowledge, including how the system produces knowledge, or control of the access to knowledge. We also consider issues having to do with information overload, acquisition of domain knowledge, and error handling.

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 1992) 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 & 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. We proposed and implemented a declarative explanation mechanism which relies on a proof-theoretic representation of the deductions. 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 p representing child) as the all restriction rule below:

$$\text{All restriction} \qquad \frac{\vdash C \Rightarrow D}{\vdash (\text{all } p\ C) \Rightarrow (\text{all } p\ D)}$$

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 1996; McGuinness & Borgida 1995).

In analyzing user needs and help desk query logs, although we found explanation of all deductions to be important, we found a small set of inferences which were the most critical to be explained. Without some automatic support for explaining this set, the system was not usable. These inferences include 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 I 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 I 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). We believe at a minimum, these inferences need to be explained for application uses which exploit deductive closure such as configuration. Application developers may also find it useful to do special purpose handling of such inferences. In the initial development version, explanation was only provided for these inferences in an effort to minimize development costs. The two current implementations contain complete explanation. One demonstration system incorporates special handling for the most heavily used inferences providing natural language templates for presentations of explanations aimed at lay people.

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, one could generate a non-contradictory request for a high quality home theater system that costs under a certain amount. The description could later become inconsistent as more information is added. For example, a required large screen television 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:

$$\text{Bounds Conflict} \qquad \frac{\vdash C \Rightarrow (\textsf{atleast}\ m\ r) \quad \vdash C \Rightarrow (\textsf{atmost}\ n\ r) \quad n < m}{\vdash C \Rightarrow \text{NOTHING}}$$

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 most of the implementations, users typically ask for explanations of a particular object property or relationships between objects. Under error conditions, users had more trouble identifying an appropriate query to ask, thus we included a simple explanation command that finds the last error encountered and generates an explanation of the contradiction. 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). Given the completed information, the system can then explain missed deductions.

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.

In CLASSIC there is a meta language 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. 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 & McGuinness 1996).

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 users will find themselves doing repeatedly while maintaining a project.

We believe, with knowledge of the domain and appropriate analysis of evolution, it is possible to build a knowledge evolution environment suitable for extending knowledge bases. In a fairly domain specific manner, we considered the evolution support environment for configurators. We looked at the information that was typically added and found generally only certain classes had new subclasses added to them as product knowledge evolved. We also found that instances were typically populated in particular patterns. While CLASSIC provides no general support for such additions, one domain specific environment was produced in an application family that supported specific subclass and instance addition. Also, in related work, Gil(Gil & Melz 1996) has analyzed planning-based uses of another description logic-based system and systematically supports knowledge base evolution with respect to the known plan usage. The more general problem that does not rely on domain or reasoning knowledge has been addressed in the editor work (Paley, Lawrence, & Karp 1997) for the general frame protocol. The general work, of course, is broader yet shallower with respect to reasoning implications.

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.

Efficiency

Although the worst-case complexity of knowledge representation systems has been well-studied, there are other resource-consumption issues that are important for determining the suitability of a knowledge representation system. These concerns are generally more prosaic, but perhaps even more important, than the concerns about worst-case complexity. For example, it is 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. (There are other reasons to compute all consequences that have been seen earlier.) 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.

Application Programming Interface

One other aspect of a knowledge representation system that is vitally important for its suitability in any real application is its application programming interface, or how it can be accessed by other computer programs. 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 expected to be part of a larger C++ program, provides a very wide application programming interface. (LISP CLASSIC has a similar wide application programming interface.) There are, of course, the usual calls to add and retract knowledge and to query for the presence of particular knowledge. In addition to this 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.

A less-traditional interface that is provided by both LISP CLASSIC and NEOCLASSIC 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 general frame protocol(Chaudhri *et al.* 1997) and the open knowledge base connectivity(Chaudhri *et al.* 1998)) and translators exist between the general frame protocol API specification and CLASSIC.

Platforms

A third important 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.

CLASSIC has 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, NEOCLASSIC 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.

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 leveragable 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.

Documentation: Beyond the standard research papers, users demanded usage guidelines aimed at non-PhD researchers. In an effort to educate people on when a description logic-based system might be useful, what its limitations were, and how one might go about using one in a simple application, a long paper was written with a running (executable) example on how to use the system (Brachman *et al.* 1991).

Demonstration Applications: Motivated by the need to help users understand a simple reasoning paradigm and by the need to have a quick prototyping domain for showing off novel functionality which exploits the strengths of the underlying system, a few demonstration systems were developed. The first developed was a simple application that captures

[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.

"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, Resnick, & Isbell 1995). Both of these applications have been adapted for the web.[2]

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.

Course Materials: Motivated by the need to grow a larger community of people trained in knowledge representation in general and description logics in particular, we collaborated with a training center to generate a course. Independently, at least one university developed a similar course and a set of five running assignments to help students gain experience using the system. We collaborated on the tutorial to support the educators and to gather feedback from the students.

Talks: The value of personal introduction to topics can not be underestimated. We have given numerous general talks about knowledge representation, the use of description logics, and some of their more successful application areas. Many other colleagues have acted similarly and we now see description logics being a topic of discussion in some related technical communities such as databases and configuration.

Standard Tool Use: We believe it is imprudent to ignore the business community's demands of common standard implementation languages, reasonable support, and standard platform toolkits. The business world was accommodated by providing a development version of the system written in C. (Although the research LISP implementation is still the academic system of choice, all of the commercial applications could only use the C version, essentially because it was written in a language that developers and their management felt comfortable with.) More recently, it has been found difficult to develop extensions in one system and rely on another organization to import those extensions into a development system, so we decided to support one version of CLASSIC for *both* research and development use. This led to the development of NEOCLASSIC which is written in C++. This addresses the issue of maintaining an implementation in a widely accepted language.

[2]The web version of our wines demonstration system was provided by Chris Welty and is available at http://untangle.cs.vassar.edu/wines. We collaborated with Charles Isbell, Matt Parker, and Chris Welty to produce the web version of our stereo configurator, which is is available at http://taylor.vassar.edu/stereo-demo/.

The issue of getting CLASSIC included in the standard approved platform for application development remains outstanding. One reason for this is that it entails providing evidence of the equivalent of commercially competitive support for the product.

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 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.

The majority of the efforts over the last several years of development of the CLASSIC family have been spent on these issues. Explanation and presentation components have been built, efficiency has been improved, large application programming interfaces have been constructed, and courses and demos have been designed. The entire system has even been reimplemented in C++. Together, these efforts have made the CLASSIC family much more acceptable for use in applications.

In fact, these issues were vitally important for the use of CLASSIC in the PROSE configuration system in AT&T (Wright *et al.* 1993). Before PROSE could be fielded, there had to be a version of CLASSIC written in C, with a large application programming interface, that supported recovery from inconsistent states of knowledge and the examination of these states. Before PROSE could be widely used, there had to be an explanation component, and considerable promotion had to be done. Without the development of a special purpose knowledge acquisition tool, the project would not have been continued. Arguably, it is precisely because of the work presented in this paper that we have maintained some of, if not *the* longest lived[3] commercial applications of description logics.

Acknowledgments

We are indebted to the rest of the CLASSIC group for their contributions in the design, implementation, and applications of CLASSIC. Major contributors in all aspects of CLASSIC include Alex Borgida and Lori Alperin Resnick. Others who have impacted portions of this work include Merryll Abrahams, Ron Brachman, Charles Foster, Charles Isbell, Elia Weixelbaum, and Jon Wright.

[3]CLASSIC-based configurators are still in use in Lucent and some have projected life spans into the year 2000, NCR has a commercially deployed CLASSIC-based knowledge discovery tool, and AT&T has deployed CLASSIC-based knowledge enhanced search applications(McGuinness 1998).

References

Baader, F.; Bürckert, H.-J.; Heinsohn, J.; Hollunder, B.; Müller, J.; Nebel, B.; Nutt, W.; and Profitlich, H.-J. 1991. Terminological knowledge representation: A proposal for a terminological logic. German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany.

Baader, F.; Hollunder, B.; Nebel, B.; Profitlich, H.-J.; and Franconi, E. 1992. An empirical analysis of optimization techniques for terminological representation systems, or, making KRIS get a move on. In *Proceedings KR-92*, 270–281. Morgan Kaufmann.

Borgida, A., and McGuinness, D. L. 1996. Inquiring about frames. In *Proceedings KR-96*, 340–349. Morgan Kaufmann.

Borgida, A.; Brachman, R. J.; McGuinness, D. L.; and Resnick, L. A. 1989. CLASSIC: A structural data model for objects. In *Proceedings SIGMOD-89*, 59–67. Association for Computing Machinery.

Borgida, A. 1992. From type systems to knowledge representation: Natural semantics specifications for Description Logics. *International Journal of Intelligent and Cooperative Information Systems* 93–126.

Brachman, R. J.; McGuinness, D. L.; Patel-Schneider, P. F.; Resnick, L. A.; and Borgida, A. 1991. Living with CLASSIC: When and how to use a KL-ONE-like language. In Sowa, J., ed., *Principles of Semantic Networks: Explorations in the representation of knowledge*. San Mateo, California: Morgan-Kaufmann. 401–456.

Brachman, R. J. 1992. "Reducing" CLASSIC to practice: Knowledge representation theory meets reality. In *Proceedings KR-92*, 247–258. Morgan Kaufmann.

Buchheit, M.; Donini, F. M.; and Schaerf, A. 1993. Decidable reasoning in terminological knowledge representation systems. *Journal of Artificial Intelligence Research* 1:109–138.

Chaudhri, V. K.; Farquhar, A.; Fikes, R.; Karp, P. D.; and Rice, J. 1997. The generic frame protocol 2.0. Technical report, Artificial Intelligence Center, SRI International, Menlo Park, CA.

Chaudhri, V. K.; Farquhar, A.; Fikes, R.; and Karp, P. D. 1998. Open knowledge base connectivity 2.0. Technical report, Technical Report KSL-09-06, Stanford University KSL.

Donini, F. M.; Lenzerini, M.; Nardi, D.; and Nutt, W. 1991. The complexity of concept languages. In *Proceedings KR-91*, 151–162. Morgan Kaufmann.

Gil, Y., and Melz, E. 1996. Explicit representations of problem-solving strategies to support knowledge acquisition. In *Proceedings AAAI-96*, 469–476.

Levesque, H. J. 1982. The logic of incomplete knowledge bases. In Brodie, M. L.; Mylopoulos, J.; and Schmidt, J. W., eds., *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. New York: Springer-Verlag. 165–186.

McGuinness, D. L., and Borgida, A. 1995. Explaining subsumption in Description Logics. In *Proceedings IJCAI-95*, 816–821.

McGuinness, D. L., and Wright, J. R. 1998. Conceptual modeling for configuration: A description logic-based configurator platform. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing Journal - Special Issue on Configuration*.

McGuinness, D. L.; Resnick, L. A.; and Isbell, C. 1995. Description Logic in practice: A CLASSIC application. In *Proceedings IJCAI-95*, 2045–2046.

McGuinness, D. L. 1996. *Explaining Reasoning in Description Logics*. Ph.D. Dissertation, Department of Computer Science, Rutgers University. Also available as Rutgers Technical Report Number LCSR-TR-277.

McGuinness, D. L. 1998. Ontological Issues for Knowledge-Enhanced Search. In *Proceedings of Formal Ontology in Information Systems*. Also to appear in Frontiers in Artificial Intelligence and Applications, IOS-Press, Washington, DC, 1998.

Paley, S. M.; Lawrence, J. D.; and Karp, P. D. 1997. A generic knowledge-base browser and editor. In *Proceedings AAAI-97*, 1045–1051.

Patel-Schneider, P. F.; McGuinness, D. L.; Brachman, R. J.; Resnick, L. A.; and Borgida, A. 1991. The CLASSIC knowledge representation system: Guiding principles and implementation rationale. *SIGART Bulletin* 2(3):108–113.

Patel-Schneider, P. F.; Resnick, L. A.; McGuinness, D. L.; Weixelbaum, E.; Abrahams, M.; and Borgida, A. 1997. NeoClassic user's guide: Version 1.0. AI Principles Research Department, AT&T Labs—Research.

Patil, R. S.; Fikes, R. E.; Patel-Schneider, P. F.; Mckay, D.; Finin, T.; Gruber, T.; and Neches, R. 1992. The DARPA knowledge sharing effort: Progress report. In *Proceedings KR-92*, 777–788. Morgan Kaufmann.

Resnick, L. A.; Borgida, A.; Brachman, R. J.; McGuinness, D. L.; and Patel-Schneider, P. F. 1995. CLASSIC description and reference manual for the COMMON LISP implementation: Version 2.3. AI Principles Research Department, AT&T Bell Laboratories.

Rychtyckyj, N. 1996. DLMS: An evaluation of KL-ONE in the automobile industry. In *Proceedings KR-96*, 588–596. Morgan Kaufmann.

Weixelbaum, E. S. 1991. C-Classic reference manual release 1.0. AT&T Bell Laboratories.

Wright, J. R.; Weixelbaum, E. S.; Brown, K.; Vesonder, G. T.; Palmer, S. R.; Berman, J. I.; and Moore, H. H. 1993. A knowledge-based configurator that supports sales, engineering, and manufacturing at AT&T network systems. In *Proceedings IAAI-93*, 183–193. American Association for Artificial Intelligence.