NASA-CR-202419

[illegible]
[illegible]
OCIT
[illegible]

Final Report covering the period of
6/1/1991 - 5/31/1993

# Management of Knowledge Representation Standards Activities
NASA-Ames Contract NCC 2-719

Principal Investigator

Ramesh S. Patil
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
Phone: (310) 822-1511
ramesh@isi.edu

# Final Report covering the period of 6/1/1991 - 5/31/1993

# Management of Knowledge Representation Standards Activities: The DARPA Knowledge Sharing Effort

Ever since the mid-seventies, researchers have recognized that capturing knowledge is the key to building large and powerful AI systems. In the years since, we have also found that representing knowledge is difficult and time consuming. In spite of the tools developed to help with knowledge acquisition, knowledge base construction remains one of the major costs in building an AI system: For almost every system we build, a new knowledge base must be constructed from scratch. As a result, most systems remain small to medium in size. Even if we build several systems within a general area, such as medicine or electronics diagnosis, significant portions of the domain must be represented for every system we create.

The cost of this duplication of effort has been high and will become prohibitive as we attempt to build larger and larger systems. To overcome this barrier we must find ways of preserving existing knowledge bases and of sharing, reusing, and building on them.

This report describes the efforts undertaken over the last two years under the NASA grant NCC 2-719 to identify the issues underlying the current difficulties in sharing and reuse, and a community wide initiative to overcome them. First, we discuss four bottlenecks to sharing and reuse, present a vision of a future in which these bottlenecks have been ameliorated, and describe the efforts of the initiative's four working groups to address these bottlenecks. We then address the supporting technology and infrastructure that is critical to enabling the vision of the future. Finally, we consider topics of longer-range interest by reviewing some of the research issues raised by our vision.

## Sharing and Reuse of Knowledge

There are many senses in which the work that went into creating a knowledge-based system can be shared and reused. Rather than mandating one particular sense, the approach we have taken in this project seeks to support several of them. One mode is to reuse is through the exchange of techniques and detailed analysis of a domain. That is, the content of some knowledge-base or an implemented algorithm is not directly used, but the approach behind it is communicated in a manner that facilitates its reimplementation. Another mode of reuse is through the inclusion of an existing knowledge-base into a new system. That is, the content of some module is copied into another at design time and merged (possibly after extension or revision) into the new system. A third mode is through the run-time communication of knowledge, data or services. That is, one module invokes another either as a procedure from a function library or as a collaborated agent (specialist) in the problem-solving activity.

These modes of reuse do not work particularly smoothly today. Explaining how to reproduce a technique often requires communicating subtle issues that are more easily expressed formally; whether stated formally or in natural language, the explanations require shared understanding of the intended interpretations of terms. The reuse of a knowledge-base is only feasible to the extent that their model of the world is compatible with the intended new use. The reuse of external agents' knowledge or service is feasible only to the extent that we understand what requests the agent are prepared to accept. These difficulties in sharing and reuse and possible approaches to resolving them were studied at a 3 day workshop involving 40 top scientists in artificial intelligence, organized by ISI (and supported by DARPA, NSF, and industry).

Technical analyses of knowledge representation technology indicated four key impediments and four complementary areas in which development of common, agreed-upon conventions would enhance leverage between individual research efforts. The four areas of impediments are: (1) heterogeneous representations, (2) multiple dialects within language families, (3) lack of common conventions for communication among intelligent agents, and (4) model "mismatch" at the knowledge level. The proposed approaches to addressing each of these problem are: (1) mechanisms for translation between knowledge bases represented in different languages; (2) common versions of languages and reasoning modules within families of representational paradigm; (3) protocols for communication between separate knowledge-based modules, as well as between knowledge-based systems and databases; and, (4) libraries of ``ontologies," i.e., pre-fabricated foundations for application-specific knowledge bases in a particular topic area.

To further develop and refine the solutions proposed, develop conventions, and to test the solutions in implemented systems, Working groups (comprised of researchers from the DARPA AI community and other academic and industry volunteers) have been established for each of these four areas, and are developing draft specifications which are circulated for review both within the working groups and among the project participants at ISI. The specifications are fed back to various collaborating DARPA projects which are building relevant technology, so that their work can move toward providing preliminary implementations of the specifications. Under the current grant, ISI is responsible for supporting and coordinating the activities of these groups, and disseminating the result of these activities to the broader research community and potential users.

In the following sections we describe the impediments and the results of the working groups.

## The impediments to knowledge sharing

**Impediment 1**. Heterogeneous Representations: There are a wide variety of approaches to knowledge representation, and knowledge that is expressed in one formalism cannot directly be incorporated into another formalism. However, this diversity is inevitable—the choice of one form of knowledge representation over another can have a big impact on a system's performance. There is no single knowledge representation that is best for all problems, nor is there likely to be one. Thus, in many cases, sharing and reusing knowledge will involve translating from one representation to another. Currently, the only way to do this translating is by manually recoding knowledge from one representation to another. We need tools that can help automate the translation process.

**Impediment 2**. Dialects within Language Families: Even within a single family of knowledge representation formalisms (for example, the KL-One family), it can be difficult to share knowledge across systems if the knowledge has been encoded in different dialects. Some of the differences between dialects are substantive, but many involve arbitrary and inconsequential differences in syntax and semantics. All such differences, substantive or trivial, impede sharing. It is important to eliminate unnecessary differences at this level.

**Impediment 3**. Lack of Communication Conventions: Knowledge sharing does not necessarily require a merger of knowledge bases. If separate systems can communicate with one another, they can benefit from each other's knowledge without sharing a common knowledge base. Unfortunately, this approach is not generally feasible for today's systems because we lack an agreed-on protocol specifying how systems are to query each other and in what form answers are to be delivered. Similarly, we lack standard protocols that would provide interoperability between knowledge representation systems and other, conventional software, such as database management systems.

**Impediment 4**. Model Mismatches at the Knowledge Level: Finally, even if the language-level problems previously described are resolved, it can still be difficult to combine two knowledge bases or establish effective communications between them. These remaining barriers arise when different primitive terms are used to organize them; that is, if they lack shared vocabulary and domain terminology. For example, the type hierarchy of one knowledge base might split the concept Object into Physical-Object and Abstract-Object, but another might decompose Object into Decomposable-Object, Nondecomposable-Object, Conscious-Being, and Non-Conscious-Thing. The absence of knowledge about the relationship between the two sets of terms makes it difficult to reconcile them. Sometimes these differences reflect differences in the intended purposes of the knowledge bases. At other times, these differences are just arbitrary (for example, different knowledge bases use Isa, Isa-kind-of, Subsumes, AKO, or Parent relations, although their real intent is the same). If we could develop shared sets of explicitly defined terminology, sometimes called ontologies, we could begin to remove some of the arbitrary differences at the knowledge level. Furthermore, shared ontologies could provide a basis for packaging knowledge modules—describing the contents or services that are offered and their ontological commitments in a composable, reusable form.

## The Knowledge-Sharing Effort

The desire to collaborate through knowledge sharing and reuse has arisen within a segment of the broad knowledge representation community that is interested in scaling up to larger systems and that views the sharing and reuse of knowledge bases as a means to this end. Closely related to this effort is a concern for building embedded systems in which knowledge representation systems support certain functions rather than act as ends in themselves.

In particular, our goal is to support researchers in areas requiring systems bigger than a single person can build. These areas include engineering and design domains, logistics and planning domains, and various integrated modality areas (for example, multimedia interfaces). Researchers working on such topics need large knowledge bases that model complex objects; because these models drive complex systems, they cannot be skeletons. Putting together much larger systems, of which various stand-alone systems being built today are just components, is an interesting challenge.

The creation of such knowledge resources requires community wide effort. This effort engenders a need for agreed-on conventions to enable us to build pieces that fit together. Eventually, in pursuing the goal of large, shared knowledge bases as part of a nationwide information infrastructure, these conventions might become objects of study for the definition of more formal standards. Currently, however, the conventions are intended to support experiments in knowledge sharing among interested parties.

In the next section describes the activities of our four working groups on these foundation-laying activities. For each group, we summarize the problem being addressed, the approach being taken, and the outcomes sought.

## Interlingua

The Interlingua Working Group is headed by Richard Fikes and Mike Genesereth, both of Stanford University.

**Problem Formulation.** The Interlingua Working Group focuses on the problems posed by the heterogeneity of knowledge representation languages. Specifically, to interchange knowledge among disparate programs (written by different programmers, at different times, in different languages), effective means need to be developed for translating knowledge bases from one specialized representation language into another. The goal of this group is to specify a language for communicating knowledge between computer programs (as opposed to a language for the internal representation of knowledge within computer programs). This language needs

(1) an agreed-on declarative semantics that is independent of any given interpreter,

(2) sufficient expressive power to represent the declarative knowledge contained in typical application system knowledge bases, and

(3) structure that enables semiautomatic translation into and out of typical representation languages.

**Approach.** This group is specifying a language (KIF [knowledge interchange format]) that is a form of predicate calculus extended to include facilities for defining terms, representing knowledge about knowledge, reifying functions and relations, specifying sets, and encoding commonly used nonmonotonic reasoning policies. The group is also conducting knowledge-interchange experiments to substantially test the viability and adequacy of the language. The experiments focus on developing and testing a methodology for semiautomatic translation to and from typical representation languages and the use of the interchange format as an intermodule communication language to facilitate interoperability.

**Outcomes.** The specification for interlingua will evolve in a set of layers. The innermost layer will be a core, analogous to the primitives in Lisp, providing basic representational and language extension functions. The next layer will provide idioms and extensions that make the language more usable, analogous to the set of functions provided by Common Lisp. This working group will be responsible for developing these specifications. Its output will be (1) a living document containing the current KIF specification, describing open issues, and presenting current proposals for modification and (2) a corpus of documented microexamples, using fragments of knowledge bases to illustrate how they translate into KIF and to point out open issues.

dialects of the KL-One family of languages support KRSS standard. A copy of the specification is included as Appendix C.

## External Interfaces

The External Interfaces Working Group, cochaired by Tim Finin of the Unisys Center for Advanced Information Technology and Gio Wiederhold of Stanford University, focuses on interfaces that provide interoperability between a knowledge representation system and other software systems.

**Problem Formulation.** The time is ending when an intelligent system can exist as a single, monolithic program that provides all the functions necessary to do a complex task. Intelligent systems will be used and deployed in environments that require them to interact with a complex of other software components. These components will include conventional software modules, operating system functions, and database servers as well as other intelligent agents. There is a strong need to develop standard interface modules and protocols to make it easier to achieve this interoperability. The working group is concerned with three aspects of this problem: providing interoperability with other intelligent agents, conventional (for example, relational) database management systems, and object-oriented database systems.

**Approach.** To provide run-time interoperability between knowledge representation systems, we need a language or protocol that allows one system to pose queries or provide data to another. The group has begun the specification of such a language, KQML. The intent is that KQML will be to knowledge representation systems what sql has become to database management systems—a high-level, portable protocol for which all systems will provide interfaces. The current specification is organized as a protocol stack in which the lowest information-conveying layer is based on the interlingua. Higher layers in this stack provide for modality (for example, assert, retract, query), transmission (for example, the specification of the recipient agent or agents), and complex transactions (for example, the efficient transmission of a block of data).

The integration of ai and database management system technologies promises to play a significant role in shaping the future of computing. As noted by Brodie (1988), this integration is crucial not only for next-generation computing but also for the continued development of database management system technology and the effective application of much of ai technology. The need exists for (1) access to large amounts of existing shared data for knowledge processing, (2) the efficient management of data as well as knowledge, and (3) the intelligent processing of data. The working group is studying the many existing interfaces between knowledge representation systems and relational databases (for example, Mckay, Finin, and O'Hare [1990]) and is attempting to develop specifications for a common one. The primary issues here are the various ways in which the data in the databases can best be mapped into the knowledge representation objects.

The third task that the group is looking at is providing interfaces between knowledge representation systems and object-oriented databases. The goal here is to be able to use an object-oriented database as a substrate under the knowledge representation system to provide a persistent object store for knowledge base objects. This work is exploratory, but the potential benefits in the long run are significant. They include (1) building and managing knowledge bases much larger than the current technology will support and (2) providing controls for transactions and concurrent access to knowledge bases at an object level.

**Outcomes.** The External Interfaces Working Group is concentrating on the development of the KQML protocol as its first goal. It hopes that an early implementation will be used to help build test beds for several distributed, cooperative demonstration systems. With regard to database interfaces, several working group members are attempting to integrate existing models for interfaces between knowledge representation systems and relational databases and to produce a specification of a common one. The working group is also planning an experiment in which a simple interface will be built to allow an existing object-oriented database to be used as a substrate under one of the representation systems being investigated by the KRSS working group. A draft specification of the KQML protocol is included in Appendix D. An on-line version of the specification and related publications can be accessed via the web from knowledge-sharing home page (http://www.isi.edu/isd/KRSharing/).

## Shared, Reusable Knowledge Bases

The Shared, Reusable Knowledge Bases Group is headed by Tom Gruber of Stanford University and Marty Tenenbaum of EITech, Inc.

**Problem Formulation.** This group is working on mechanisms to enable the development of libraries of shareable knowledge modules and the reuse of their knowledge-level contents. Today's knowledge bases are structured as monolithic networks of highly interconnected symbols, designed for specific tasks in narrow domains. As a result, it is difficult to adapt existing knowledge bases to new uses or even to identify the shareable contents. To enable the accumulation of shareable knowledge and the use of this knowledge by multiple systems, we need a means for designing composable modules of knowledge. The working group is chartered to identify the barriers to the building and use of shared knowledge modules, characterize potential approaches to overcoming these barriers, and conduct experiments exploring mechanisms for knowledge sharing.

**Approach.** The working group supports three kinds of activity. One is the identification of important research issues for knowledge sharing, including problems of methodology (for example, multidisciplinary, collaborative knowledge base design) as well as engineering (for example, scalability, shareability). A second activity is the development of ontologies that define terminology used to represent bodies of shareable knowledge. The task includes (1) identifying bodies of knowledge worth the effort to formally represent and make shareable and (2) defining coherent sets of terms that characterize the ontological commitments and representational choices for modeling these bodies of knowledge. A third type of working group activity is the coordination of collaborative experiments in knowledge sharing in which multiple research groups attempt to share and use each other's knowledge bases (for example, libraries of device models). Some experiments will evaluate the use of ontologies as a mechanism for sharing (that is, for modularity and composability of knowledge modules and the specification of their contents).

**Outcomes.** To these ends, the working group is concentrating on three objectives. The first is a survey of the state of the art in research on knowledge sharing and reuse, which identifies the techniques currently being explored and recommends research on the critical problems to be solved. A second outcome is a set of results from the collaborative experiments on knowledge sharing, including the ontologies used for each experiment and lessons learned about tools and methodologies for developing them. An immediate subgoal for this outcome is to develop a mechanism for representing these ontologies in portable form, building on the work of the other three working groups. The third, more long-term objective is to develop a suite of exemplary shared ontologies and the knowledge bases using them.

The ontology working group has developed Ontolingua: a language for describing sharable ontologies, and a libraries of foundational ontologies. Under separate funding from DARPA, the Stanford ontology group maintains an ontology library containing ontologies developed by the knowledge sharing effort. These ontologies, relevant publications and the Ontolingua system can be accessed over the web through the Stanford Knowledge System Laboratory home page. (http://www-ksl.stanford.edu/).

## Dissemination

One of the key objectives of this project is to raise the awareness in the research and application community to the need for sharing and reuse, as well as to make them aware of the results produced by the DARPA knowledge sharing effort. Towards this end, ISI has organized a number of panels at major conferences, publications, and supported talks by researchers. A few of the major activities are listed below:

- Plenary session on Knowledge Sharing at the International conference on Knowledge Representation and Reasoning, 1991, Cambridge, MA
- Lead article in AAAI Magazine on The DARPA Knowledge Sharing Effort (see Appendix A)
- Invited talk by Ramesh Patil at the International conference on Knowledge Representation and Reasoning, 1992 (see Appendix B)

In addition many of the researchers involved in this effort (not funded by the grant) have published papers related to these activities.

## Conclusion

Attempting to move beyond the capabilities of current knowledge-based systems mandates knowledge bases that are substantially larger than those we have today. However, representing and acquiring knowledge is a difficult and time-consuming task. Knowledge-acquisition tools and current development methodologies will not make this problem go away because the root of the problem is that knowledge is inherently complex and the task of capturing it is correspondingly complex. Thus, we cannot afford to waste whatever knowledge we do succeed in acquiring. We will be hard pressed to make knowledge bases much bigger than we have today if we continue to start from scratch each time we construct a new system. Building qualitatively bigger knowledge-based systems will be possible only when we are able to share our knowledge and build on each other's labor and experience.

## References

Bennett, J. S. 1984. roget: Acquiring the Conceptual Structure of a Diagnostic Expert System. In Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems, 83–88. Washington, D.C.: IEEE Computer Society.

Brachman, R. J. 1990. The Future of Knowledge Representation. In Proceedings of the Eighth National Conference on Artificial Intelligence, 1082–1092. Menlo Park, Calif.: American Association for Artificial Intelligence.

Brachman, R. J., and Levesque, H. J. 1984. The Tractability of Subsumption in Frame-Based Description Languages. In Proceedings of the Third National Conference on Artificial Intelligence, 34–37. Menlo Park, Calif.: American Association for Artificial Intelligence.

Brodie, M. 1988. Future Intelligent Information Systems: ai and Database Technologies Working Together. In Readings in Artificial Intelligence and Databases, eds. J. Mylopoulos and M. Brodie, 623–641. San Mateo, Calif.: Morgan Kaufmann.

Cargill, C. F. 1989. Information Technology Standardization: Theory, Process, and Organizations. Bedford, Mass.: Digital.

Chandrasekaran, B. 1986. Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design. IEEE Expert 1(3): 23–30.

Chandrasekaran, B. 1983. Toward a Taxonomy of Problem-Solving Types. AI Magazine 4(4): 9–17.

Cutkosky, M. R., and Tenenbaum, J. M. 1990. A Methodology and Computational Framework for Concurrent Product and Process Design. Mechanism and Machine Theory 25(3): 365–381.

Feiner, S. K., and McKeown, K. R. 1990. Coordinating Text and Graphics in Explanation Generation. In Proceedings of the Eighth National Conference on Artificial Intelligence, 442–449. Menlo Park, Calif.: American Association for Artificial Intelligence.

Finin, T., and Fritzson, R. 1989. How to Serve Knowledge—Notes on the Design of a Knowledge Server. Presented at the AAAI Spring Symposium on Knowledge System Tools and Languages, 28–30 March, Stanford, Calif.

Forbus, K. 1990. The Qualitative Process Engine. In Readings in Qualitative Reasoning about Physical Systems, eds. D. Weld and J. de Kleer, 220–235. San Mateo, Calif.: Morgan Kaufmann.

Gruber, T. R. 1991. An Experiment in the Collaborative Development of Shared Ontology, Technical Report, KSL 91-51, Knowledge Systems Laboratory, Stanford University. Forthcoming.

Guha, R. V., and Lenat, D. B. 1990. cyc: A Mid-Term Report. AI Magazine 11(3): 32–59.

Harp, B.; Aberg, P.; Benjamin, D.; Neches, R.; and Szekely, P. 1991. drama: An Application of a Logistics Shell. In Proceedings of the Annual Conference on Artificial Intelligence Applications for Military Logistics, 146-151. Williamsburg, Va.: American Defense Preparedness Association. Forthcoming.

Johnson, W. L., and Harris, D. R. 1990. Requirements Analysis Using aries: Themes and Examples. In Proceedings of the Fifth Annual Knowledge-Based Software Assistant Conference, 121–131. Liverpool, N.Y.: Rome Air Development Center.

Kahn, R. E., and Cerf, V. E. 1988. An Open Architecture for a Digital Library System and a Plan for Its Development, Volume I: The World of Knowbots, Technical Report, Corporation for National Research Initiatives, Reston, Virginia.

Kahn, G.; Nowlan, S.; and McDermott, J. 1984. A Foundation for Knowledge Acquisition. In Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems, 89–96. Washington, D.C.: IEEE Computer Society.

McDermott, J. 1990. Developing Software Is Like Talking to Eskimos about Snow. In Proceedings of the Eighth National Conference on Artificial Intelligence, 1130–1133. Menlo Park, Calif.: American Association for Artificial Intelligence.

Mckay, D.; Finin, T.; and O'Hare, A. 1990. The Intelligent Database Interface. In Proceedings of the Eighth National Conference on Artificial Intelligence, 677–684. Menlo Park, Calif.: American Association for Artificial Intelligence.

Neches, R., et. al. 1991. Enabling Technology for Knowledge Sharing. AI Magazine, 12(3): 36-56.

Pan, J. Y.-C.; Tenenbaum, J. M.; and Glicksman, J. 1989. A Framework for Knowledge-Based Computer-Integrated Manufacturing. IEEE Transactions on Semiconductor Manufacturing 2(2): 33–46.

Patil, R. S., et. al. 1992. The DARPA Knowledge Sharing Effort: Progress Report. In the Proceedings of the Third Internatinal Conference on Principles of Knowledge Representation and Reasoning. Morgan Kauffman Publishers, Inc. San Mateo, CA.

Roth, S., and Mattis, J. 1990. Data Characterization for Intelligent Graphics Presentation. In Proceedings of CHI-90, The Annual Conference on Computer-Human Interaction, 193–200. New York: Association of Computing Machinery.

Sathi, A.; Fox, M. S.; and Greenberg, M. 1990. Representation of Activity Knowledge for Project Management. IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-7(5): 531–552.

Smith, D. R. 1990. A Semiautomatic Program Development System. IEEE Transactions on Software Engineering SE-16(9): 1024–1043.

Steele, G. R. 1984. Common Lisp: The Language. Bedford, Mass.: Digital.

Steels, L. 1990. Components of Expertise. AI Magazine 11(2): 29–49.

Stefik, M. 1986. The Next Knowledge Medium. AI Magazine 7(1): 34–46.

Swartout, W. R., and Smoliar, S. 1989. Explanation: A Source for Guidance in Knowledge Representation. In Knowledge Representation and Organization in Machine Learning, ed. K. Morik, 1–16. New York: Springer-Verlag.

Waterman, D. A. 1986. A Guide to Expert Systems. Reading, Mass.: Addison-Wesley.

## BIBLIOGRAPHY OF PUBLICATIONS

1.Robert Neches, Richard Fikes, Tim Finin, Thomas Gruber, Ramesh Patil, Ted Senator, and William Swartout. Enabling Technology for Knowledge Sharing. AI Magazine, 12(3): 36-56. 1991.

2.Ramesh Patil, Richard Fikes, Peter Patel-Schneider, Don Mckay, Tim Finin, Thomas Gruber, and Robert Neches. The DARPA knowledge Sharing Effort: Progress Report. In the Proceedings of the Third Internatinal Conference on Principles of Knowledge Representation and Reasoning (KR'92) Morgan Kauffman Publishers, Inc. San Mateo, CA. 1992

3.Michael Genesereth, and Richard Fikes. Knowledge Interchange Format Version 3.0 Reference Manual. Stanford University Department of Computer Science Logic Group Technical Report: Logic-92-1. June 1992.

4. Peter F. Patel-Schneider and William Swartout. Description-Logic Knowledge Representation System Specification from the KRSS Group of the ARPA Knoweledge Sharing Effort. Working Draft (Appendix D).

5.Tim Finin, Jay Weber, Geo Weiderhold, Michael Geneserath, Richard Fritzson, Richard Pelavin, James McGuire, Stuart Shapiro, and Chris Beck. Draft Specification of the KQML Agent Commnication Language. Draft (Appendix E).

## APPENDIX A

# Management of Knowledge Representation Standards Activities NASA-Ames Contract NCC 2-719

**Final Report covering the period of 6/1/1991 - 5/31/1993**

**Principal Investigator**

Ramesh S. Patil
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
Phone: (310) 822-1511
ramesh@isi.edu

# Enabling Technology for Knowledge Sharing

Robert Neches
Information Sciences Institute
University of Southern California

Richard Fikes
Knowledge Systems Laboratory
Stanford University

Tim Finin
Computer Science Department
University of Maryland

Thomas Gruber
Knowledge Systems Laboratory
Stanford University

Ramesh Patil
Information Sciences Institute
University of Southern California

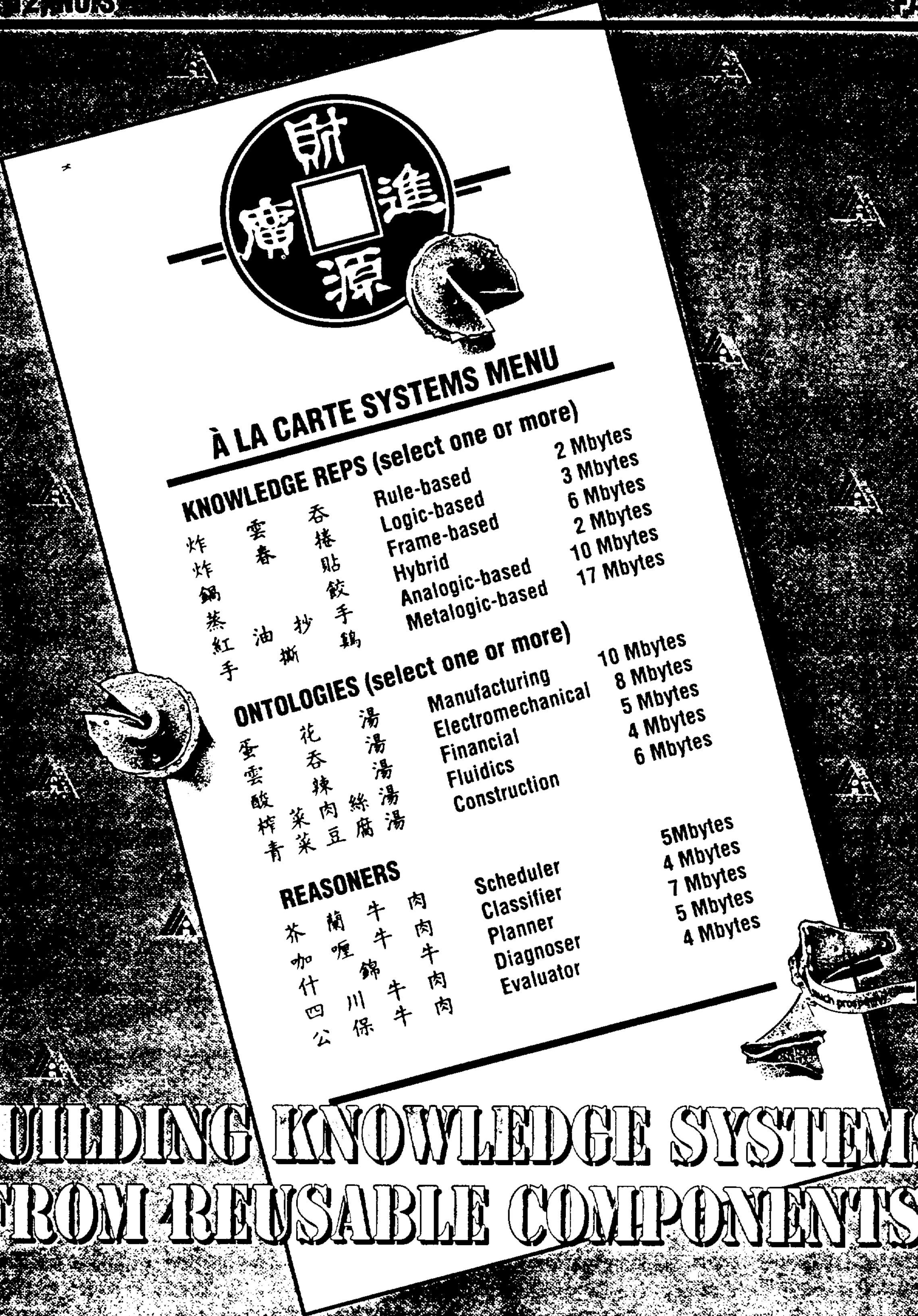
Ted Senator
Financial Crimes Enforcement Network
US Treasury Department

William Swartout.
Information Sciences Institute
University of Southern California

**Citation:** Robert Neches, Richard Fikes, Tim Finin, Thomas Gruber, Ramesh Patil, Ted Senator, and William Swartout. Enabling Technology for Knowledge Sharing. AI Magazine, 12(3): 36-56. 1991.

# AI magazine

VOLUME 12, NO 3 FALL 1991

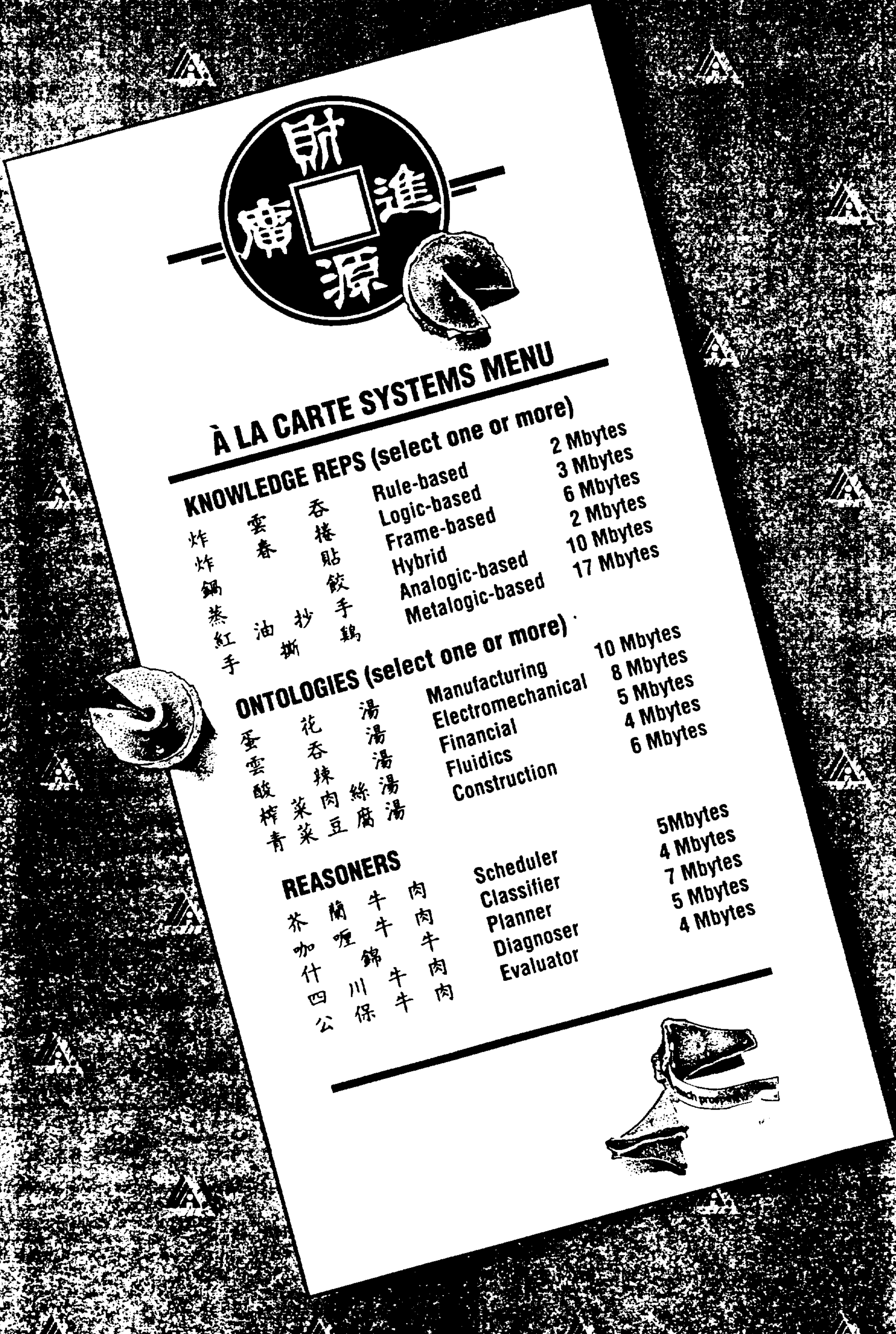

財源廣進
À LA CARTE SYSTEMS MENU
KNOWLEDGE REPS (select one or more)
炸雲吞 Rule-based 2 Mbytes
炸春捲 Logic-based 3 Mbytes
鍋貼 Frame-based 6 Mbytes
蒸餃 Hybrid 2 Mbytes
紅油抄手 Analogic-based 10 Mbytes
手撕雞 Metalogic-based 17 Mbytes
ONTOLOGIES (select one or more)
蛋花湯 Manufacturing 10 Mbytes
雲吞湯 Electromechanical 8 Mbytes
酸辣湯 Financial 5 Mbytes
榨菜肉絲湯 Fluidics 4 Mbytes
青菜豆腐湯 Construction 6 Mbytes
REASONERS
芥蘭牛肉 Scheduler 5Mbytes
咖喱牛肉 Classifier 4 Mbytes
什錦牛 Planner 7 Mbytes
四川牛肉 Diagnoser 5 Mbytes
公保牛肉 Evaluator 4 Mbytes

## BUILDING KNOWLEDGE SYSTEMS FROM REUSABLE COMPONENTS

An official publication of the American Association for Artificial Intelligence

# Enabling Technology for Knowledge Sharing

Robert Neches, Richard Fikes, Tim Finin, Thomas Gruber, Ramesh Patil, Ted Senator, and William R. Swartout



*Building new knowledge-based systems today usually entails constructing new knowledge bases from scratch. It could instead be done by assembling reusable components. System developers would then only need to worry about creating the specialized knowledge and reasoners new to the specific task of their system. This new system would interoperate with existing systems, using them to perform some of its reasoning. In this way, declarative knowledge, problem-solving techniques, and reasoning services could all be shared among systems. This approach would facilitate building bigger and better systems cheaply. The infrastructure to support such sharing and reuse would lead to greater ubiquity of these systems, potentially transforming the knowledge industry. This article presents a vision of the future in which knowledge-based system development and operation is facilitated by infrastructure and technology for knowledge sharing. It describes an initiative currently under way to develop these ideas and suggests steps that must be taken in the future to try to realize this vision.*

Ever since the mid-seventies, researchers have recognized that capturing knowledge is the key to building large and powerful AI systems. In the years since, we have also found that representing knowledge is difficult and time consuming. Although we have developed tools to help with knowledge acquisition, knowledge base construction remains one of the major costs in building an AI system: For almost every system we build, a new knowledge base must be constructed from scratch. As a result, most systems remain small to medium in size. Even if we build several systems within a general area, such as medicine or electronics diagnosis, significant portions of the domain must be rerepresented for every system we create.

The cost of this duplication of effort has been high and will become prohibitive as we attempt to build larger and larger systems. To overcome this barrier and advance the state of the art, we must find ways of preserving existing knowledge bases and of sharing, reusing, and building on them.

This article describes both near- and long-term issues underlying an initiative to address these concerns. First, we discuss four bottlenecks to sharing and reuse, present a vision of a future in which these bottlenecks have been ameliorated, and touch on the efforts of the initiative's four working groups to address these bottlenecks. We then elaborate on the vision by describing the model it implies for how knowledge bases and knowledge-based systems could be structured and developed. This model involves both infrastructure and supporting technology. The supporting technology is the topic of our near-term interest because it is critical to enabling the infrastructure. Therefore, we return to discussing the efforts of the four working groups of our initiative, focusing on the enabling technology that they are working to define. Finally, we consider topics of longer-range interest by reviewing some of the research issues raised by our vision.

0738-4602/91/$4.00 © 1991 AAAI

## Sharing and Reuse

There are many senses in which the work that went into creating a knowledge-based system can be shared and reused. Rather than mandating one particular sense, the model described in this article seeks to support several of them. One mode of reuse is through the exchange of techniques. That is, the content of some module from the library is not directly used, but the approach behind it is communicated in a manner that facilitates its reimplementation. Another mode of reuse is through the inclusion of source specifications. That is, the content of some module is copied into another at design time and compiled (possibly after extension or revision) into the new component. A third mode is through the run-time invocation of external modules or services. That is, one module invokes another either as a procedure from a function library or through the maintenance of some kind of client-server relationship between the two (Finin and Fritzson 1989).

*Attempting to move beyond the capabilities of current knowledge-based systems mandates knowledge bases that are substantially larger than those we have today.*

These modes of reuse do not work particularly smoothly today. Explaining how to reproduce a technique often requires communicating subtle issues that are more easily expressed formally; whether stated formally or in natural language, the explanations require shared understanding of the intended interpretations of terms. The reuse of source specifications is only feasible to the extent that their model of the world is compatible with the intended new use. The reuse of external modules is feasible only to the extent that we understand what requests the modules are prepared to accept. Let us consider these complexities in more detail by reviewing four critical impediments to sharing and reuse.

**Impediment 1. Heterogeneous Representations:** There are a wide variety of approaches to knowledge representation, and knowledge that is expressed in one formalism cannot

*Instead, the process of building a knowledge-based system will start by assembling reusable components.*

directly be incorporated into another formalism. However, this diversity is inevitable—the choice of one form of knowledge representation over another can have a big impact on a system's performance. There is no single knowledge representation that is best for all problems, nor is there likely to be one. Thus, in many cases, sharing and reusing knowledge will involve translating from one representation to another. Currently, the only way to do this translating is by manually recoding knowledge from one representation to another. We need tools that can help automate the translation process.

**Impediment 2. Dialects within Language Families:** Even within a single family of knowledge representation formalisms (for example, the KL-One family), it can be difficult to share knowledge across systems if the knowledge has been encoded in different dialects. Some of the differences between dialects are substantive, but many involve arbitrary and inconsequential differences in syntax and semantics. All such differences, substantive or trivial, impede sharing. It is important to eliminate unnecessary differences at this level.

**Impediment 3. Lack of Communication Conventions:** Knowledge sharing does not necessarily require a merger of knowledge bases. If separate systems can communicate with one another, they can benefit from each other's knowledge without sharing a common knowledge base. Unfortunately, this approach is not generally feasible for today's systems because we lack an agreed-on protocol specifying how systems are to query each other and in what form answers are to be delivered. Similarly, we lack standard protocols that would provide interoperability between knowledge representation systems and other, conventional software, such as database management systems.

**Impediment 4. Model Mismatches at the Knowledge Level:** Finally, even if the language-level problems previously described are resolved, it can still be difficult to combine two knowledge bases or establish effective communications between them. These remaining barriers arise when different primitive terms are used to organize them; that is, if they lack shared vocabulary and domain terminology. For example, the type hierarchy of one knowledge base might split the concept Object into Physical-Object and Abstract-Object, but another might decompose Object into Decomposable-Object, Nondecomposable-Object, Conscious-Being, and Non-Conscious-Thing. The absence of knowledge about the relationship between the two sets of terms makes it difficult to reconcile them. Sometimes these differences reflect differences in the intended purposes of the knowledge bases. At other times, these differences are just arbitrary (for example, different knowledge bases use Isa, Isa-kind-of, Subsumes, AKO, or Parent relations, although their real intent is the same). If we could develop shared sets of explicitly defined terminology, sometimes called *ontologies*, we could begin to remove some of the arbitrary differences at the knowledge level. Furthermore, shared ontologies could provide a basis for packaging knowledge modules—describing the contents or services that are offered and their ontological commitments in a composable, reusable form.

## A Vision: Knowledge Sharing

In this article, we present a vision of the future in which the idea of knowledge sharing is commonplace. If this vision is realized, building a new system will rarely involve constructing a new knowledge base from scratch. Instead, the process of building a knowledge-based system will start by assembling reusable components. Portions of existing knowledge bases would be reused in constructing the new system, and special-purpose reasoners embodying problem-solving methods would similarly be brought in. Some effort would go into connecting these pieces, creating a "custom shell" with preloaded knowledge. However, the majority of the system development effort could become focused on creating only the specialized knowledge and reasoners that are new to the specific task of the system under construction. In our vision, the new system could interoperate with existing systems and pose queries to them to perform some of its reasoning.

Furthermore, extensions to existing knowledge bases could be added to shared repositories, thereby expanding and enriching them.

Over time, large rich knowledge bases, analogous to today's databases, will evolve. In this way, declarative knowledge, problem-solving techniques and reasoning services could all be shared among systems. The cost to produce a system would decrease. To the extent that well-tested parts were reused, a system's robustness would increase.

For end users, this vision will change the face of information systems in three ways. First, it will provide sources of information that serve the same functions as books and libraries but are more flexible, easier to update, and easier to query. Second, it will enable the construction and marketing of prepackaged knowledge services, allowing users to invoke (rent or buy) services. Third, it will make it possible for end users to tailor large systems to their needs by assembling knowledge bases and services rather than programming them from scratch.

We also expect changes and enhancements in the ways that developers view and manipulate knowledge-based systems. In particular, we envision three mechanisms that would increase their productivity by promoting the sharing and reuse of accumulated knowledge. First among these are libraries of multiple layers of reusable knowledge bases that could either be incorporated into software or remotely consulted at execution time. At a level generic to a class of applications, layers in such knowledge bases capture conceptualizations, tasks, and problem- solving methods. Second, system construction will be facilitated by the availability of common knowledge representation systems and a means for translation between them. Finally, this new reuse-oriented approach will offer tools and methodologies that allow developers to find and use library entries useful to their needs as well as preexisting services built on these libraries. These tools will be complemented by tools that allow developers to offer their work for inclusion in the libraries.

## The Knowledge-Sharing Effort

We are not yet technically ready to realize this vision. Instead, we must work toward it incrementally. For example, there is no consensus today on the appropriate form or content of the shared ontologies that we envision. For this consensus to emerge, we need to engage in exercises in building shared knowledge bases, extract generalizations from the set of systems that emerge, and capture these generalizations in a standard format that can be interpreted by all involved. This process requires the development of some agreed-on formalisms and conventions at the level of an interchange format between languages or a common knowledge representation language.

Simply enabling the ability to share knowledge is not enough for the technology to have full impact, however. The development and use of shared ontologies cannot become cost effective unless the systems using them are highly interoperable with both AI and conventional software, so that large numbers of systems can be built. Thus, software interfaces to knowledge representation systems are a crucial issue.

The Knowledge-Sharing Effort, sponsored by the Air Force Office of Scientific Research, the Defense Advanced Research Projects Agency, the Corporation for National Research Initiatives, and the National Science Foundation (NSF), is an initiative to develop the technical infrastructure to support the sharing of knowledge among systems. The effort is organized into four working groups, each of which is addressing one of the four impediments to sharing that we outlined earlier. The working groups are briefly described here and in greater detail later in the article.

The Interlingua Working Group is developing an approach to translating between knowledge representation languages. Its approach involves developing an intermediary language, a *knowledge interchange format* or *interlingua*, along with a set of translators to map into and out of it from existing knowledge representation languages. To map a knowledge base from one representation language into another, a system builder would use one translator to map the knowledge base into the interchange format and another to map from the interchange format back out to the second language.

The Knowledge Representation System Specification (KRSS) Working Group is taking another, complementary tack toward promoting knowledge sharing. Rather than translating between knowledge representation languages, the KRSS group is seeking to promote sharing by removing arbitrary differences among knowledge representation languages within the same paradigm. This group is currently working on a specification for a knowledge representation system that brings together the best features of languages developed within the KL-One paradigm. Similar efforts for other families of languages are expected to follow.

The External Interfaces Working Group is

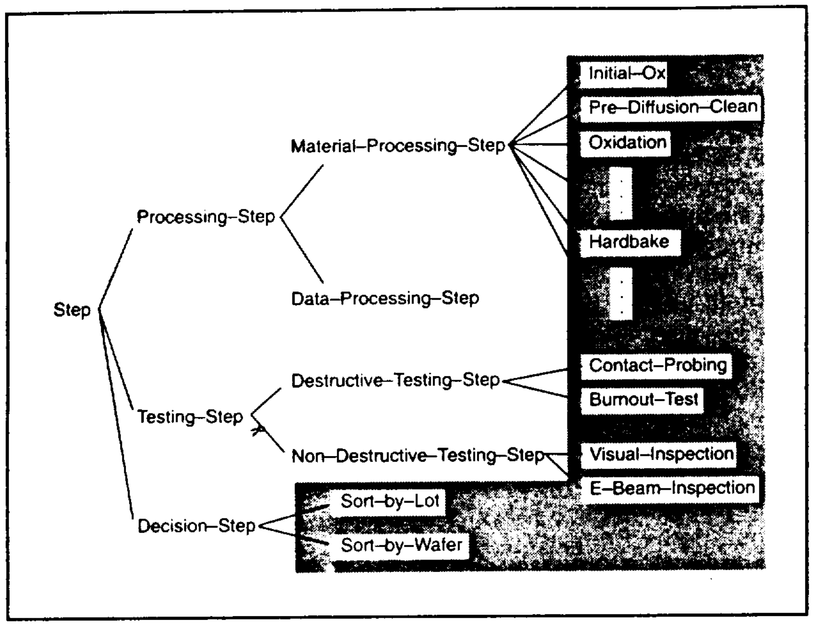


*Figure 1. The MKS Ontology of Manufacturing Operations, Elaborated with Knowledge Specific to Semiconductor Manufacturing.*

*Ontologies such as this one, in effect, lay the ground rules for modeling a domain by defining the basic terms and relations that make up the vocabulary of this topic area. These ground rules serve to guide system builders in fleshing out knowledge bases, building services that operate on knowledge bases, and combining knowledge bases and services to create larger systems. For one system to make use of either the knowledge or reasoners of another system, the two must have consistent ontologies.*

investigating yet another facet of knowledge sharing. It is developing a set of protocols for interaction that would allow a knowledge-based system to obtain knowledge from another knowledge-based system (or, possibly, a conventional database) by posting a query to this system and receiving a response. The concerns of this group are to develop the protocols and conventions through which such an interaction could take place.

Finally, the Shared, Reusable Knowledge Bases Working Group is working on overcoming the barriers to sharing that arise from lack of consensus across knowledge bases on vocabulary and semantic interpretations in domain models. As mentioned earlier, the ontology of a system consists of its vocabulary and a set of constraints on the way terms can be combined to model a domain. All knowledge systems are based on an ontology, whether implicit or explicit. A larger knowledge system can be composed from two smaller ones only if their ontologies are consistent. This group is trying to ameliorate problems of inconsistency by fostering the evolution of common, shareable ontologies. A number of candidate reusable ontologies are expected to come from this work. However, the ultimate contribution of the group lies in developing an understanding of the group processes that evolve such products and the tools and infrastructure needed to facilitate the creation, dissemination, and reuse of domain-oriented ontologies.

## Architectures of the Future

In this section, we elaborate on our vision by describing what we hope to enable concerning both knowledge bases and the systems that use them. In doing so, we look at how they are structured and the process by which they will be built. We also consider the relationship of this vision to other views that have been offered, such as Guha and Lenat's (1990) Cyc effort, Stefik's (1986) notion of Knowledge Media, and Kahn's notion of Knowbots (Kahn and Cerf 1988). Finally, we offer a view of the range of system models that this approach supports.

### Structural and Development Models for Knowledge Bases

In a AAAI-90 panel on software engineering, John McDermott (1990) described how AI could make software development easier: Write programs to act as frameworks for handling instances of problem classes.

Knowledge-based systems can provide such frameworks in the form of top-level declarative abstraction hierarchies, which an application builder elaborates to create a specific system. Essentially, hierarchies built for this purpose represent a commitment to provide specific services to applications that are willing to adopt their model of the world.

When these top-level abstraction hierarchies are represented with enough information to lay down the ground rules for modeling a domain, we call them ontologies. An ontology defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for combining terms and relations to define extensions to the vocabulary. An example is the MKS generic model of manufacturing steps (Pan, Tenenbaum, and Glicksman 1989), illustrated in figure 1 along with a set of application-specific extensions for semiconductor manufacturing. The frame hierarchy in MKS defines classes of concepts that the system's reasoning modules (for example, schedulers and diagnosers) are prepared to operate on. The slots and slot restrictions on these frames define how one must model a particular manufacturing domain to enable the use of these modules.

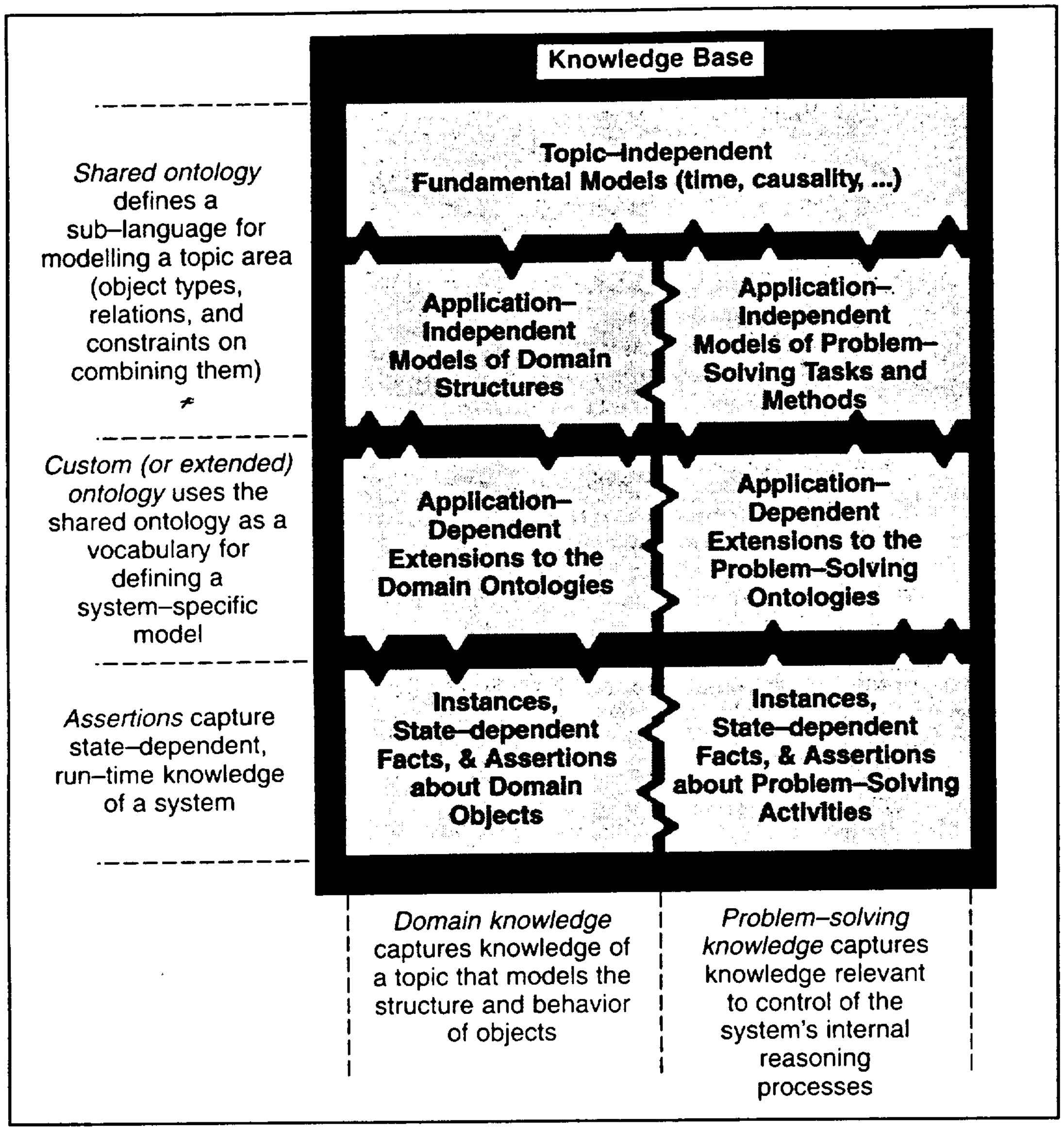

Knowledge Base
Topic–Independent Fundamental Models (time, causality, ...)
Application–Independent Models of Domain Structures
Application–Independent Models of Problem–Solving Tasks and Methods
Application–Dependent Extensions to the Domain Ontologies
Application–Dependent Extensions to the Problem–Solving Ontologies
Instances, State–dependent Facts, & Assertions about Domain Objects
Instances, State–dependent Facts, & Assertions about Problem–Solving Activities
*Shared ontology* defines a sub–language for modelling a topic area (object types, relations, and constraints on combining them)
*Custom (or extended) ontology* uses the shared ontology as a vocabulary for defining a system–specific model
*Assertions* capture state–dependent, run–time knowledge of a system
*Domain knowledge* captures knowledge of a topic that models the structure and behavior of objects
*Problem–solving knowledge* captures knowledge relevant to control of the system's internal reasoning processes

*Figure 2. The Anatomy of a Knowledge Base.*

*Application systems contain many different kinds and levels of knowledge. At the top level are ontologies, although often represented only implicitly in many of today's systems. The top-level ontologies embody representational choices ranging from topic independent (for example, models of time or causality) to topic specific but still application-independent knowledge (for example, domain knowledge about different kinds of testing operations represented in a manufacturing system or problem-solving knowledge about hypothesis classes in a diagnostic system). This top level of knowledge is elaborated by more application-specific models (for example, knowledge about chip-testing operations in a specific manufacturing application or failure modes in a circuit diagnosis system). Together, they define how a particular application describes the world. At the bottom level, assertions using the vocabulary of these models capture the current state of the system's knowledge. Knowledge at the higher levels, being less specialized, is easier to share and reuse. Knowledge at the lower levels can only be shared if the other system accepts the models in the levels above.*

The MKS example is hardly unique. A number of systems have been built in a manner consistent with this philosophy, for example, FIRST-CUT and NEXT-CUT (Cutkosky and Tenenbaum 1990), QPE (Forbus 1990), Cyc (Guha and Lenat 1990), ARIES (Johnson and Harris 1990), SAGE (Roth and Mattis 1990), Carnegie Mellon University's factory scheduling and project management system (Sathi, Fox, and Greenberg 1990), KIDS (Smith 1990), and EES (Swartout and Smoliar 1989). The notion that generic structure can be exploited in building specialized systems has been argued for a long time by Chandrasekaran (1983, 1986) and more recently by Steels (1990). The notion has also long been exploited in knowledge-

*An ontology defines the basic terms and relations comprising the vocabulary of a topic area…*

acquisition work, for example, in systems such as MORE (Kahn, Nowlan, and McDermott 1984) and ROGET (Bennett 1984).

The range of knowledge captured with ontologies is described in figure 2. There is some fuzziness in the figure's distinction between shared and custom ontologies because they are relative terms—any given knowledge base is custom with respect to the knowledge it extends and is shared with respect to the knowledge that extends it. Nevertheless, the essential idea is that application knowledge bases should consist of layers of increasingly specialized, less reusable knowledge. As is argued later, explicitly organizing knowledge bases in this fashion is a step on the critical path to enabling sharing and reuse.

There are a number of uses for ontologies. The primary uses today are in defining knowledge-based system frameworks in the spirit advocated by John McDermott (1990). However, a number of additional possibilities open if *libraries* of these ontologies can be developed because these libraries define the common models needed for combining knowledge bases or successfully communicating between independent modules. Ontologies and knowledge bases can also be viewed as ends in themselves, that is, as repositories of information in the spirit suggested by Stefik's (1986) discussion of knowledge media.

Although every knowledge-based system implicitly or explicitly embodies an ontology, ontologies are rarely shared across systems. Commonalities among existing systems can be identified and made shareable. For example, Stanford's Summer Ontology Project found that several of the collaborators had built systems that used models of mechanical devices based on concepts such as module, port, and connection (Gruber 1991). However, each system used slightly different names and formalisms. An ontology for lumped element models that defines these concepts with consistent, shareable terminology is under construction. A library of such shared ontologies would facilitate building systems by reducing the effort invested in reconciliation and reinvention.

## Structural and Development Models for Knowledge-Based Systems

Figure 3 illustrates the envisioned organization of an individual knowledge-based application system. The local system consists of a set of services that operate on combined (or indistinguishable) knowledge bases and databases. One uniform user interface management system mediates interactions with humans, and a set of agents mediates interaction with other systems.

The services that make up the application consist of various reasoning modules, which can be defined at either the level of generic reasoning and inference mechanism (for example, forward chaining, backward chaining, unification) or the task level (for example, planners, diagnosers). These modules would typically be acquired off the shelf. At either level, however, they can be augmented by special-purpose reasoning modules that provide custom capabilities for exploiting particular characteristics of the application domain. In addition to these reasoning modules, the application's services are also likely to include a number of modules providing conventional capabilities, such as spreadsheets, electronic mail, hypermedia, calendar systems, statistical packages, and accounting systems.

To perform the tasks of the overall system, modules will need to interact internally (that is, knowledge base to knowledge base) and externally (that is, knowledge base–database to other knowledge-based systems or arbitrary outside software applications). For external interaction, the modules will need a language for encoding their communications. SQL serves this function for conventional database interactions, and it appears likely that it will continue to be used in the future. We call the analogous programmatic interface for knowledge-based applications KQML (knowledge query and manipulation language). KQML will consist of a language for specifying wrappers that define messages communicated between modules. These wrappers will surround declarations that will express whatever knowledge

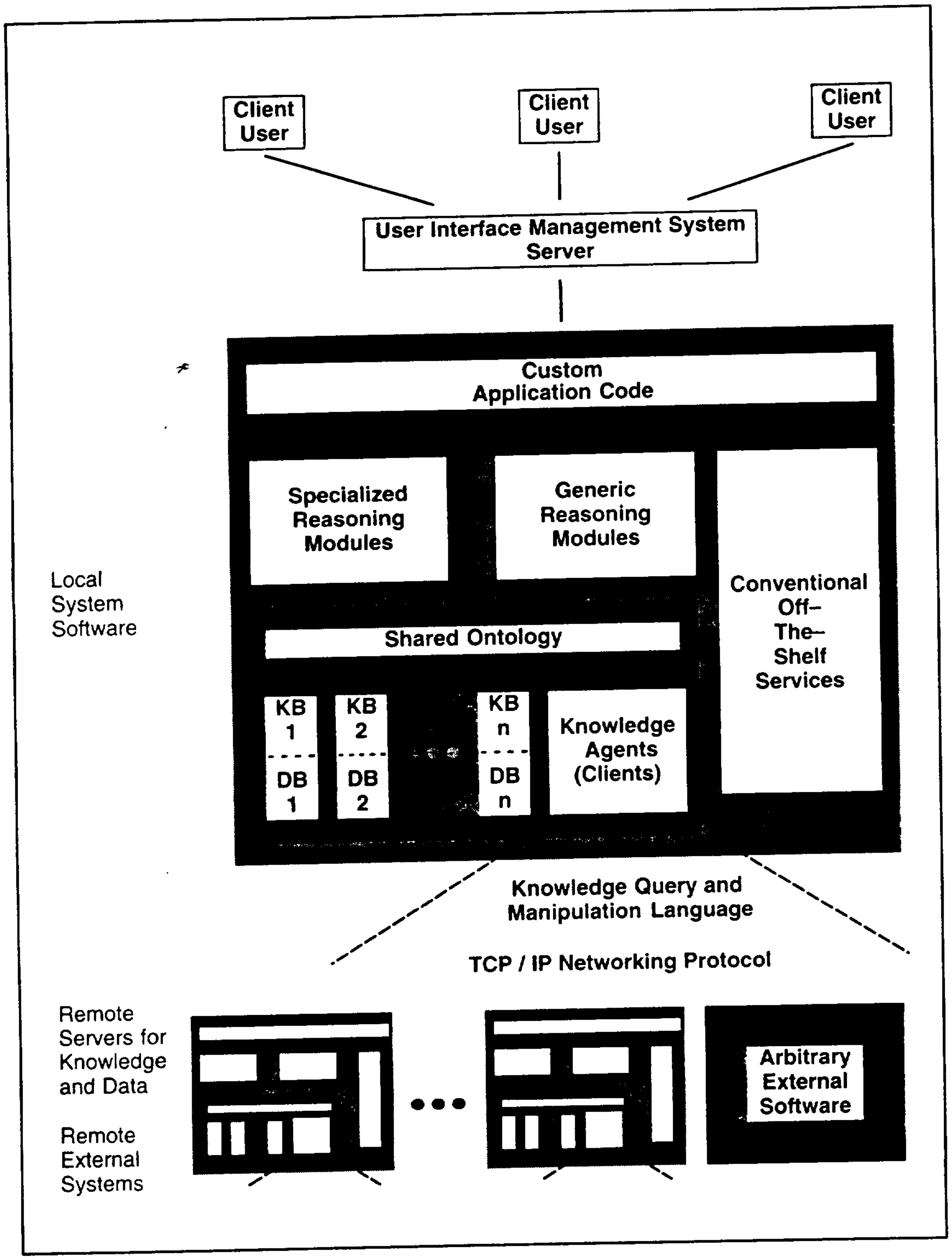

Client User
Client User
Client User
User Interface Management System Server
Custom Application Code
Specialized Reasoning Modules
Generic Reasoning Modules
Conventional Off–The–Shelf Services
Local System Software
Shared Ontology
KB 1
DB 1
KB 2
DB 2
KB n
DB n
Knowledge Agents (Clients)
Knowledge Query and Manipulation Language
TCP / IP Networking Protocol
Remote Servers for Knowledge and Data
Remote External Systems
Arbitrary External Software

*Figure 3. Architecture of a Knowledge-Based System.*

*We envision that knowledge-based systems will be assembled from components rather than built from scratch. The components include a framework for local system software in which one or more local knowledge bases are tied to a shared ontology. Remote knowledge bases can be accessed and are understood by the local system by virtue of being tied in to the ontology. Specialized reasoning modules (for example, a diagnostic system) and generic reasoning systems (that is, a representation system) are also tied to the knowledge base(s) through the ontology. In turn, these systems are glued together with conventional services through specialized, custom application code. Larger systems can be obtained from smaller ones in this architecture by either expanding the contents of a local system or interlinking multiple systems built in this fashion.*

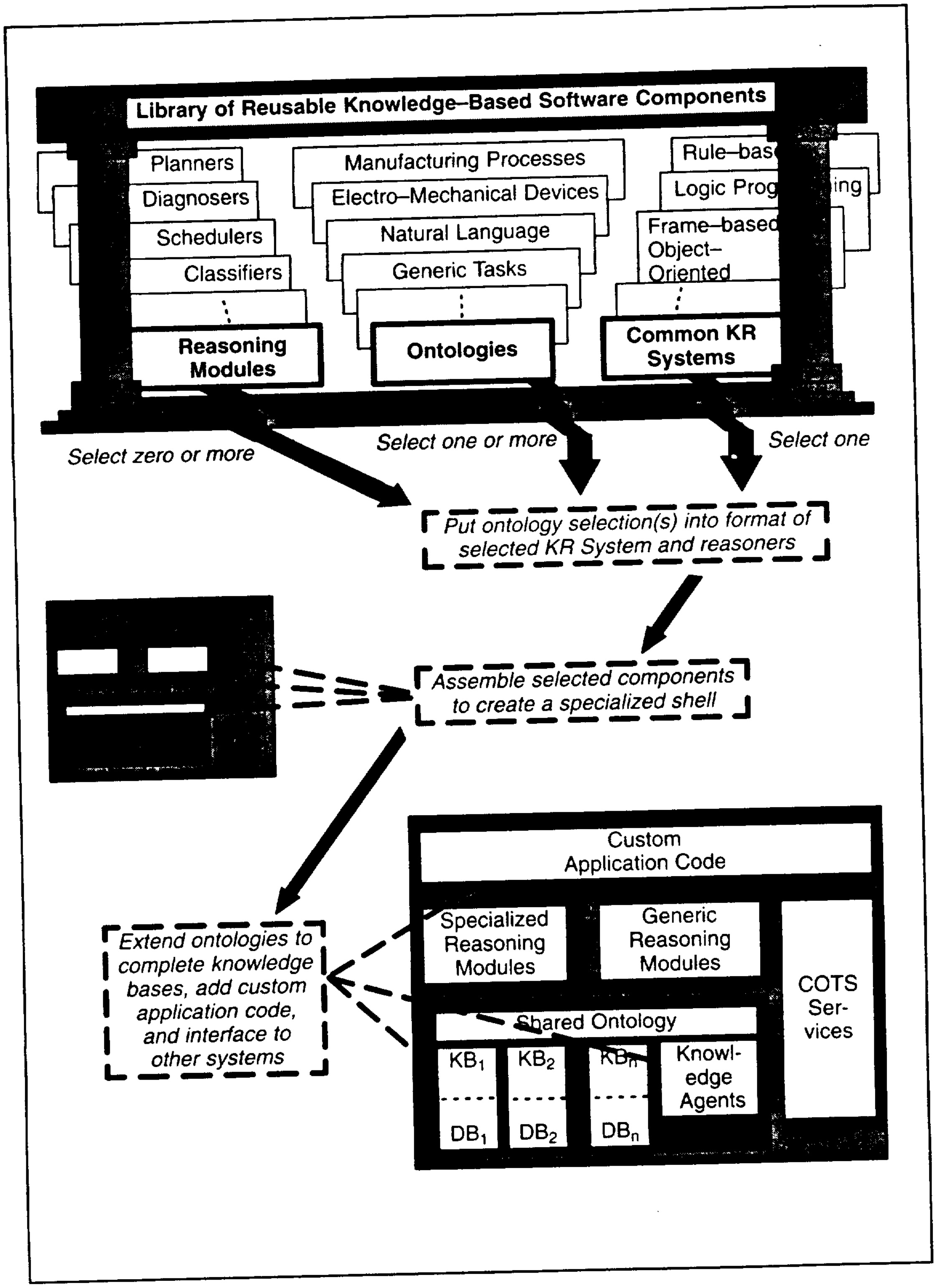

Library of Reusable Knowledge–Based Software Components
Planners
Diagnosers
Schedulers
Classifiers
Reasoning Modules
Manufacturing Processes
Electro–Mechanical Devices
Natural Language
Generic Tasks
Ontologies
Rule–bas
Logic Prog
ing
Frame–based Object–Oriented
Common KR Systems
Select zero or more
Select one or more
Select one
Put ontology selection(s) into format of selected KR System and reasoners
Assemble selected components to create a specialized shell
Extend ontologies to complete knowledge bases, add custom application code, and interface to other systems
Custom Application Code
Specialized Reasoning Modules
Generic Reasoning Modules
COTS Services
Shared Ontology
KB1
KB2
KBn
DB1
DB2
DBn
Knowledge Agents

*Figure 4. Envisioned Phases in Defining a Knowledge-Based System.*

*With libraries of reusable knowledge-based software components, building an application could become much more of a configuration task and, correspondingly, less of a programming activity. System builders could select specialized reasoning modules and ontologies, converting them to the format required by the selected knowledge representation if they were not already in this format. This approach gives them a specialized shell with some knowledge built in. This custom shell can then be utilized to build the application from a higher-level starting point.*

the sending module must pass to the receiving module.

The uniform KQML communication protocol will facilitate modular construction of system components and, thus, facilitate the development of libraries of reusable components. Such libraries will contain generic reasoners such as truth maintenance systems, task-specific reasoners (planners, diagnosers, and so on), and domain-specific reasoners (for example, electric circuit simulators). The libraries will also contain a number of ontologies covering both structural knowledge and problem-solving knowledge. Some will be domain oriented, and some will correspond to particular reasoners. Also residing in these libraries will be a collection of knowledge representation system implementations to choose from. We expect that several different representational paradigms, or families, might be available in the library. For each family, we expect multiple implementations of a common core language, similar to Common Lisp (Steele 1984), which has a core specification and several competing implementations of this specification, each of which offers various performance, environment, and function enhancements.

As figure 4 illustrates, application-specific systems will be developed by assembling components from a library into a customized shell, which is then used to develop the application. The first task of system developers would be to configure this specialized shell. This process will involve selecting ontologies, specialized reasoning modules, and a knowledge representation system from the library. As in any configuration task, there are likely to be constraints that must be respected. For example, a particular specialized scheduling module might assume that tasks to be scheduled are modeled according to a particular ontology. The entries in the library must provide enough information about themselves to allow system developers to understand these constraints.

Once the ontologies, specialized reasoning modules, and a knowledge representation system have been selected and assembled, the system developers have a specialized shell for their application. This shell differs from today's shells in that it will come with built-in knowledge, not just specialized programming features. The system developers' tasks are then to flesh out the knowledge base, add whatever custom application code is necessary, and write software interfaces to any other systems the new application will work with.

In configuring this specialized shell, it would be highly desirable if there were support for translation between representations. Then, developers would not be obliged to use the original implementation if their application had different performance or function requirements. For example, a common ontology of electromechanical devices could serve DRAMA (Harp et al. 1991), which analyzes logistical implications of design changes, and COMET (Feiner and McKeown 1990), which generates multimedia how-to presentations about maintenance tasks. COMET needs device models to access related device components and primarily requires efficient storage and retrieval from its representation language. In contrast, DRAMA analyzes implications of large amounts of changing knowledge and, therefore, demands efficient inference mechanisms.

Note that the approach we have been describing scales with the evolution of infras-

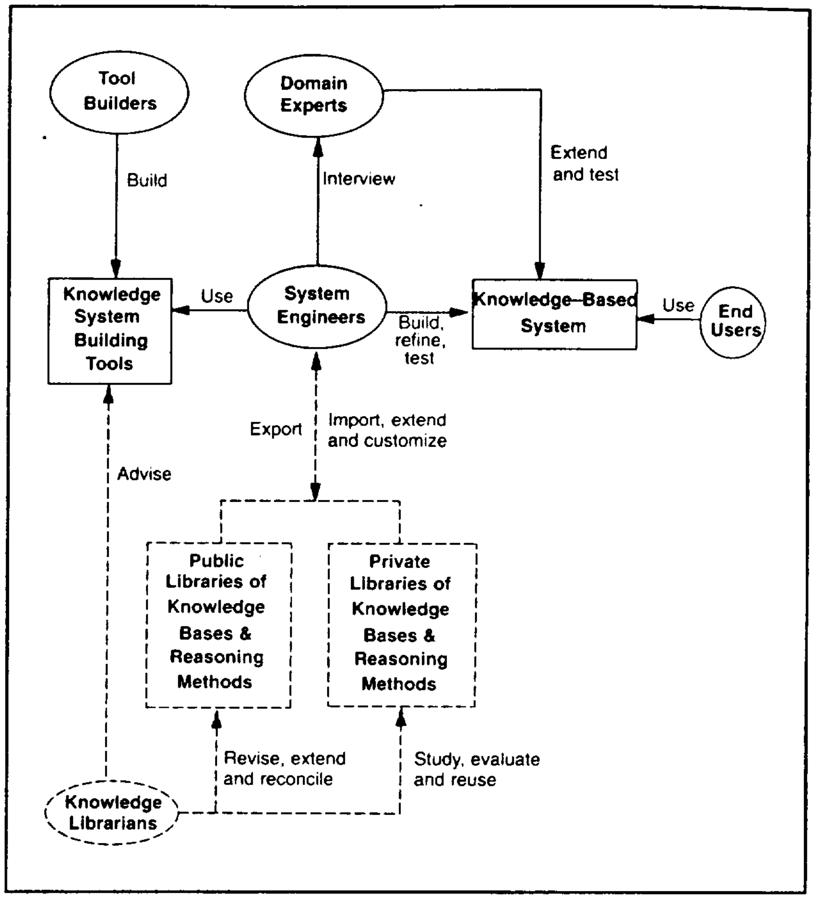

Tool Builders
Domain Experts
Build
Interview
Extend and test
Knowledge System Building Tools
Use
System Engineers
Build, refine, test
Knowledge-Based System
Use
End Users
Export
Import, extend and customize
Advise
Public Libraries of Knowledge Bases & Reasoning Methods
Private Libraries of Knowledge Bases & Reasoning Methods
Revise, extend and reconcile
Study, evaluate and reuse
Knowledge Librarians

*Figure 5. Current versus Envisioned Models of the AI Software Life Cycle.*

*Adding knowledge libraries represents a significant change in methodology for building knowledge-based systems. Knowledge librarians, a new category of participant in the process, would ensure that submitted ontologies and reasoners were ready for release. They would help system engineers browse the library and select modules. They would also help tool builders create tools and development environments to assist in these activities.*

*Adding a knowledge library represents a significant change to knowledge base methodology.*

tructure for knowledge sharing. It is feasible with existing technology. If and when common knowledge representation systems become available, their use would broaden the portability and reusability of a given library. Similarly, the development and dissemination of techniques for translation between different representation languages would also broaden the use of a library.

Figure 5 contrasts our model of the life cycle for knowledge-based software with the approach followed in expert system software today. The notion of libraries is a key difference. In today's models of expert system software development, exemplified by Waterman's (1986) book on expert systems, there are at least four classes of participants in the life cycle. Tool builders create shells and development environments. System engineers take these tools and create an initial version of a knowledge-based system by interviewing domain experts. Then, together with the domain experts, the system engineers test, extend, and refine the system. Finally, end users put the system to use. The vision we propose changes the nature of the system engineer's work and has the potential to merge roles by giving domain experts more ability to act as system engineers. It also adds a new participant to the infrastructure: a knowledge librarian, who serves as a keeper of reusable ontologies and implementations of specialized reasoning methods.

The knowledge librarian works with tool builders on tools and development environments to help system engineers browse and select modules from the library. System engineers import, extend, and customize these modules. They can retain the customized elements in private libraries for sharing and reuse within a subcommunity. Alternatively, they can offer their developments to the library for possible inclusion as extensions to the existing set of knowledge.

One of the crucial functions that must be performed in the management of a knowledge library is the determination of when submitted knowledge is ready for release. If knowledge is simply added and edited by all participants without some discipline, then it will be difficult to achieve the level of reliability and stability needed for practical software development.

Adding a knowledge library represents a significant change to knowledge base methodology. It transforms the system-building process away from using power tools for constructing essentially custom, one-of-a-kind systems. Instead, system building becomes a process of selecting and combining a range of components. The system engineer becomes much more like a builder of homes and much less like a sculptor. A home builder has a range of components and materials, from bricks and two-by-fours to prefabricated walls or even rooms. A home builder has a choice of building each new home from small components that give a lot of design flexibility or from larger components that trade off reduced design options for reduced construction effort. Similarly, system engineers would find a range of grain sizes among objects in the knowledge library and would be empowered to make analogous choices.

## Comparison with Other Visions

The model of knowledge-based systems that we just described bears significant relationships to other notions that have been offered.

One recent example is MCC Corporation's Cyc Project (Guha and Lenat 1990), which seeks to develop an extremely large knowledge base of commonsense knowledge under which all applications would be built. The Cyc Project provides a language, Cyc-L, for implementing its ontology and developing an application-specific knowledge base. Because its scope is so broad, Cyc represents one extreme in the range of efforts compatible with the model we propose. In our scheme, Cyc's knowledge base could be one large entry in the library of components (or, perhaps, it might be broken into several smaller modules, or *microtheories*). Its implementation language, Cyc-L, would be one of the entries in the library of representation systems. If one chose to build a system entirely within Cyc, our model of the development process and that of the Cyc Project are largely consistent. If one wishes to go outside Cyc, our model is complementary. Potential users might use our knowledge-interchange format to translate other knowledge bases into Cyc-L or, conversely, to translate Cyc's knowledge into some other representation system. Alternatively, they might use external protocols to access a module built in Cyc and build other modules through other means.

If successful, our approach would help make the knowledge in Cyc accessible even to those system developers who for one reason or another do not choose to use the whole Cyc knowledge base or represent their systems in Cyc-L.

However, the model we propose also differs significantly from the Cyc effort. Among other things, it allows the development of large systems without having to first commit to a particular knowledge representation formalism; users do not even have to commit to homogeneity. Furthermore, this approach allows for the development and use of ontologies and knowledge bases under a more conservative, bottom-up development model as an alternative to committing to a large, broad knowledge base. Thus, an alternative use of this model aims for the evolution of smaller, topic-specific ontologies intended for sharing by specialized communities. (Over time, it is possible that these topic-specific ontologies would grow and merge with others, so that an eventual end result might be a broadly encompassing ontology such as that sought by the Cyc effort.)

Our vision owes a great deal to Mark Stefik's (1986) view of knowledge bases as the next mechanism of information exchange. Stefik was interested, as are we, in both how systems of the future will be built and what uses will be made of them. He suggested that expert systems were parts and precursors of a new knowledge medium, a set of interactive multidisciplinary tools for information manipulation:

> A knowledge medium based on AI technology is part of a continuum. Books and other passive media can simply store knowledge. At the other end of the spectrum are expert systems which can store and also apply knowledge. In between are a number of hybrid systems in which the knowledge processing is done mostly by people. There are many opportunities for establishing human-machine partnerships and for automating tasks incrementally.

In these interactive tools, the computer provides storage, retrieval, and group communications services as well as languages and tools that enable precision and explicitness in manipulating knowledge about a topic.

In producing the new knowledge medium, Stefik argued that expert systems should be viewed as complex artifacts, such as automobiles or airplanes, which are assembled from highly sophisticated materials and subassemblies. A marketplace that supports the specialized suppliers of component technologies benefits the manufacturer of the end product. The marketplace provides economies of scale that make it possible to develop component technologies of a quality and sophistication that would be unaffordable if the costs had to be borne entirely within a single, built-from-scratch, one-of-a-kind product development effort. Thus, by analogy, Stefik concluded that

> ...the "goods" of a knowledge market are elements of knowledge... To reduce the cost of building expert systems, we need to be able to build them using knowledge acquired from a marketplace. This requires setting some processes in place and making some technical advances.

In particular, Stefik urged combining work on expert system shells with work on standard vocabularies and ways of defining things in terms of primitives. This suggestion is similar to the notion of ontologies proposed in this article. However, Stefik questioned (as do we) the feasibility of relying entirely on the development of standard vocabularies or ontologies for domains. The key to their effectiveness lies in how these ontologies are analyzed, combined, and integrated to create large applications. Tools and methods are needed to support this process. Our vision seeks to extend Stefik's by trying to further define the process as well as the supporting tools and methods.

Our particular extensions also have some kinship to the architecture of the national information infrastructure envisioned by Kahn and Cerf (1988) for the Digital Library System. Their vision of the library of the future consists of information collections bearing some analogies to Stefik's knowledge media. The Digital Library System provides a distributed network architecture for accessing information. The architecture contains database servers, various accounting and billing servers, servers to support placing knowledge into the library and extracting knowledge from it, and servers for translating knowledge flowing to and from the library into different forms. The Digital Library System suggests a model for how our vision of development, distribution, and dissemination of knowledge-based systems might be realized in the future. At the same time, our vision proposes supporting technology—for example, *There are many unanswered questions about how large-scale systems can best be built.*

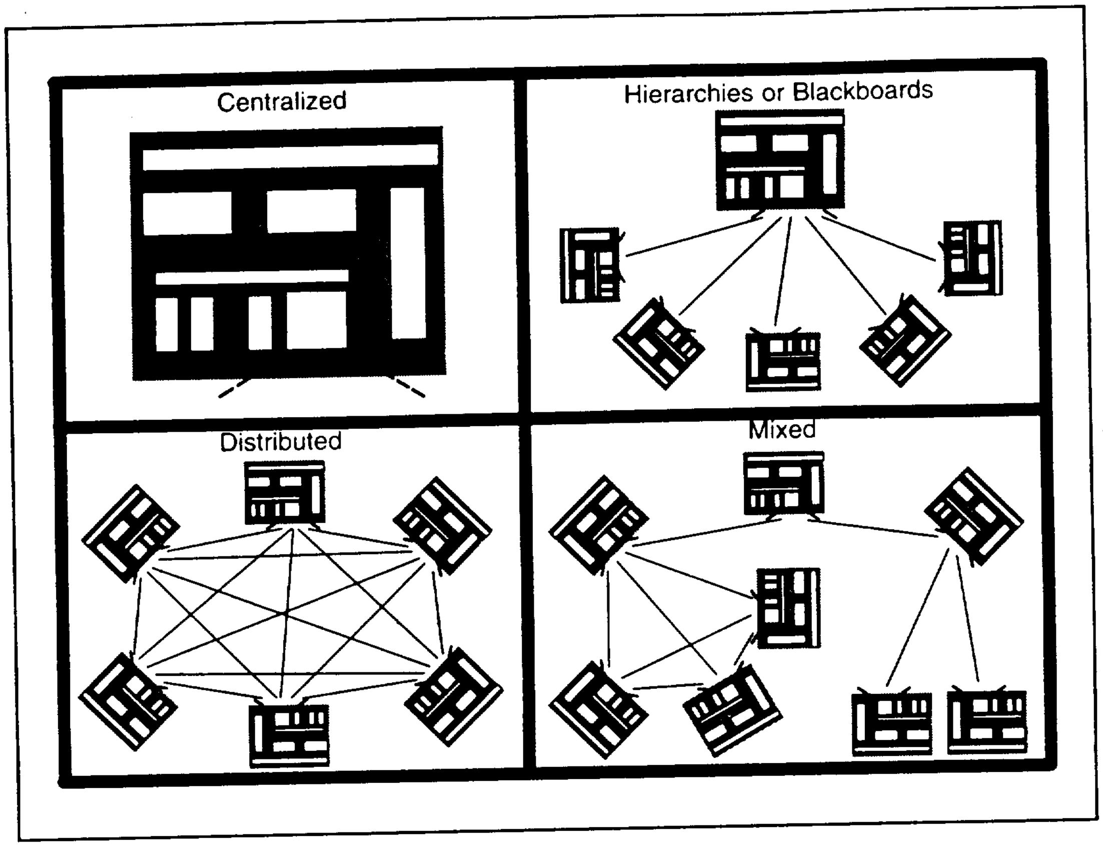


*Figure 6. A Range of (potentially heterogeneous) System Models.*
*The technology described in this article enables experimentation with alternative models for architectures of knowledge-based systems. Shared ontologies provide the basis for protocols that link separate modules.*

representation systems and translators— that could help realize the Digital Library System.

## Supporting a Range of System Architectures

The technology described in this article is intended to provide enabling tools for building a range of architectures for large, knowledge-based systems. Figure 6 illustrates one dimension of this range. Within the scope of our vision, a system might be built as a single, large, integrated, centralized package (as depicted in figure 4). In this model, sharing and reuse occur at design time. Software-engineering concerns such as modularity can be achieved by partitioning the knowledge in this system into multiple bases linked by shared ontologies. Alternatively, however, the large system could be factored into multiple separate modules, communicating according to any of a number of alternative control regimes. Sharing and reuse in this fashion occur at run time. Modularity for software-engineering reasons can be handled by the separation of modules as well as by the same internal structuring, as in the first case. Some control regimes seemingly require that each module know a great deal about the others; the duplication of information in each module that is entailed is usually regarded as a potential maintenance bottleneck. However, the architecture provides natural mechanisms (ontologies and standard communications components) for encapsulating the knowledge needed for intermodule communication.

Over and above module structuring, system models can vary along a number of other dimensions in this scheme. These dimensions include the choice of representation languages used within modules, the homogeneity or heterogeneity of representation systems across modules, the use of specialized reasoning modules, the nature of ontologies, the content of the knowledge bases, the partitioning of the knowledge bases, the tightness of coupling with databases, the degree of transparency of modules (black boxes versus glass boxes), and the locus of control.

There are many unanswered questions about how large-scale systems can best be built. Although sharing and reuse are clearly essential principles, the best way to make them operational remains to be understood. What are the trade-offs between sharing services at run time versus sharing knowledge bases at design time? On what scale and under what circumstances is translation viable? When are shared formalisms, rather than translation, required? How do domain-specific considerations drive choices of system models? What are the constraints on the usability of knowledge originally recorded for different purposes? What mechanisms best facilitate convergence between multiple users on a mutually satisfactory representation?

The intent behind the framework we outlined is not to enshrine a particular set of answers to such questions. Rather, our goal is to identify enabling technologies that make it easier to search the space of possible answers. The right answers will emerge only if we first make it easier for the AI community to explore the alternatives empirically—by building a number of large systems.

## Working Groups in the Knowledge-Sharing Initiative

The desire to collaborate through knowledge sharing and reuse has arisen within a segment of the broad knowledge representation community that is interested in scaling up to larger systems and that views the sharing and reuse of knowledge bases as a means to this end. Closely related to this effort is a concern for building embedded systems in which knowledge representation systems support certain functions rather than act as ends in themselves.

In particular, our goal is to support researchers in areas requiring systems bigger than a single person can build. These areas include engineering and design domains, logistics and planning domains, and various integrated modality areas (for example, multimedia interfaces). Researchers working on such topics need large knowledge bases that model complex objects; because these models drive complex systems, they cannot be skeletons. Putting together much larger systems, of which various stand-alone systems being built today are just components, is an interesting challenge.

The creation of such knowledge resources requires communitywide effort. This effort engenders a need for agreed-on conventions to enable us to build pieces that fit together. Eventually, in pursuing the goal of large, shared knowledge bases as part of a nationwide information infrastructure, these conventions might become objects of study for the definition of more formal standards. (For those interested in the role of standards in computing infrastructure, Cargill [1989] is a useful entry point into the topic.) Currently, however, the conventions are intended to support experiments in knowledge sharing among interested parties.

The next part of this article focuses on making our vision operational by developing these conventions. Doing so is an essential precursor to larger aspects of our vision, such as libraries of ontologies, reasoning modules, and representation systems. This section describes the activities of our four working groups on these foundation-laying activities. For each group, we summarize the problem being addressed, the approach being taken, and the outcomes sought.

*The creation of… knowledge resources requires community-wide effort.*

### Interlingua

The Interlingua Working Group is headed by Richard Fikes and Mike Genesereth, both of Stanford University.

**Problem Formulation.** The Interlingua Working Group focuses on the problems posed by the heterogeneity of knowledge representation languages. Specifically, to interchange knowledge among disparate programs (written by different programmers, at different times, in different languages), effective means need to be developed for translating knowledge bases from one specialized representation language into another. The goal of this group is to specify a language for communicating knowledge between computer programs (as opposed to a language for the internal representation of knowledge within computer programs). This language needs (1) an agreed-on declarative semantics that is independent of any given interpreter, (2) sufficient expressive power to represent the declarative knowledge contained in typical application system knowledge bases, and (3) a structure that enables semiautomated translation into and out of typical representation languages.

**Approach.** This group is specifying a language (KIF [knowledge interchange format]) that is a form of predicate calculus extended to include facilities for defining terms, representing knowledge about knowledge, reifying functions and relations, specifying sets, and encoding commonly used nonmonotonic reasoning policies. The group is also conducting knowledge-interchange experiments to substantially test the viability and adequacy of

*The goal... is to specify a language for communicating knowledge between computer programs...*

the language. The experiments focus on developing and testing a methodology for semiautomatic translation to and from typical representation languages and the use of the interchange format as an intermodule communication language to facilitate interoperability.

**Outcomes.** The specification for interlingua will evolve in a set of layers. The innermost layer will be a core, analogous to the primitives in Lisp, providing basic representational and language extension functions. The next layer will provide idioms and extensions that make the language more usable, analogous to the set of functions provided by Common Lisp. This working group will be responsible for developing these specifications. Its output will be (1) a living document containing the current KIF specification, describing open issues, and presenting current proposals for modification and (2) a corpus of documented *microexamples*, using fragments of knowledge bases to illustrate how they translate into KIF and to point out open issues.

By the time this group has completed its work, we expect that the documented interlingua specification will define a language in which the sharing and reuse of the contents of individual knowledge bases can be accomplished by transmitting specifications using KIF as the medium for expressing these languages. The language will be oriented toward supporting translation performed with a human in the loop, and we expect that several prototype translation aids will be developed during the course of this work.

## Knowledge Representation System Specifications

The KRSS working group is headed by Bill Swartout of University of Southern California/Information Sciences Institute (USC/ISI) and Peter Patel-Schneider of AT&T Bell Labs.

**Problem Formulation.** This group is concerned with specification on a separate family-by-family basis of the common elements within individual families of knowledge representation system paradigms. The intent is not to develop a "be-all, end-all" knowledge representation system. The group is not trying to develop one language that encompasses all approaches to knowledge representation; rather, it seeks to create specifications that incorporate the good features within families and develop a common version that is reasonably comprehensive yet pragmatic for each individual family. An example of one such family is the set of KL-One descendents, that is, object-centered languages with definitional constructs, such as CLASSIC, LOOM, BACK, SB-One, and OMEGA. The effort should be viewed more as an attempt to develop a Common Lisp rather than a PL-1 or Ada. The analogy to Common Lisp is imperfect, however. Knowledge representation systems perform inference on the information represented within them; programming languages do not. Thus, specifying what inferences should be performed is an additional issue in specifying a knowledge representation system.

**Approach.** The goal of this group is to allow system builders working within a family to provide a common set of features that have consensus within this paradigm in addition to the augmentations that they regard as their research contribution. The resulting language will serve as a medium for sharing knowledge bases as well as a means for communicating ideas and issues among researchers. Potential benefits are the abilities of users to more readily convert between systems and to borrow models originally built to run in other systems. The trade-offs assumed by this group are the mirror image of those faced by the Interlingua Working Group: They eliminate the problem of translating knowledge bases between systems but require one to work within a given formalism to obtain this benefit.

Specifying a knowledge representation system poses the interesting challenge of specifying just what kinds of inference the system should perform. From Brachman and Levesque (1984), we know that if the system is expressive enough to be useful, the inferences that its reasoner draws will be incomplete. How should this incomplete inference be specified? The group's approach is to construct a layered specification. There will be an inner core of the language, which will consist of a few constructs. Because this inner core will have limited expressivity, it will be possible to perform complete inference on it. Another layer, the outer core, will be built on the inner core. This layer will significantly extend the expressivity of the language, but inference on it will be incomplete. To specify the inferences that should be performed, the group will provide an abstract algorithm for drawing just those inferences that are required by the specification. Implementers of the specification must provide a reasoner at least as powerful as the one specified by this algorithm.

**Outcomes.** The group is seeking to develop a published specification and at least one initial implementation of a common language.

## External Interfaces

The External Interfaces Working Group, cochaired by Tim Finin of the Unisys Center for Advanced Information Technology and Gio Wiederhold of Stanford University, focuses on interfaces that provide interoperability between a knowledge representation system and other software systems.

**Problem Formulation.** The time is ending when an intelligent system can exist as a single, monolithic program that provides all the functions necessary to do a complex task. Intelligent systems will be used and deployed in environments that require them to interact with a complex of other software components. These components will include conventional software modules, operating system functions, and database servers as well as other intelligent agents. There is a strong need to develop standard interface modules and protocols to make it easier to achieve this interoperability. The working group is concerned with three aspects of this problem: providing interoperability with other intelligent agents, conventional (for example, relational) database management systems, and object-oriented database systems.

**Approach.** To provide run-time interoperability between knowledge representation systems, we need a language or protocol that allows one system to pose queries or provide data to another. The group has begun the specification of such a language, KQML. The intent is that KQML will be to knowledge representation systems what SQL has become to database management systems—a high-level, portable protocol for which all systems will provide interfaces. The current specification is organized as a protocol stack in which the lowest information-conveying layer is based on the interlingua. Higher layers in this stack provide for modality (for example, assert, retract, query), transmission (for example, the specification of the recipient agent or agents), and complex transactions (for example, the efficient transmission of a block of data).

The integration of AI and database management system technologies promises to play a significant role in shaping the future of computing. As noted by Brodie (1988), this integration is crucial not only for next-generation computing but also for the continued development of database management system technology and the effective application of much of AI technology. The need exists for (1) access to large amounts of existing shared data for knowledge processing, (2) the efficient management of data as well as knowledge, and (3) the intelligent processing of data. The working group is studying the many existing interfaces between knowledge representation systems and relational databases (for example, Mckay, Finin, and O'Hare [1990]) and is attempting to develop specifications for a common one. The primary issues here are the various ways in which the data in the databases can best be mapped into the knowledge representation objects.

The third task that the group is looking at is providing interfaces between knowledge representation systems and object-oriented databases. The goal here is to be able to use an object-oriented database as a substrate under the knowledge representation system to provide a persistent object store for knowledge base objects. This work is exploratory, but the potential benefits in the long run are significant. They include (1) building and managing knowledge bases much larger than the current technology will support and (2) providing controls for transactions and concurrent access to knowledge bases at an object level.

**Outcomes.** The External Interfaces Working Group is concentrating on the development of the KQML protocol as its first goal. It hopes that an early implementation will be used to help build test beds for several distributed, cooperative demonstration systems. With regard to database interfaces, several working group members are attempting to integrate existing models for interfaces between knowledge representation systems and relational databases and to produce a specification of a common one. The working group is also planning an experiment in which a simple interface will be built to allow an existing object-oriented database to be used as a substrate under one of the representation systems being investigated by the KRSS working

*... the goals proposed in this article suggest a number of high-payoff research questions for the entire research community.*

group. This approach will provide a feasibility test and generate data for further exploration.

### Shared, Reusable Knowledge Bases

The Shared, Reusable Knowledge Bases Group is headed by Tom Gruber of Stanford University and Marty Tenenbaum of EITech, Inc.

**Problem Formulation.** This group is working on mechanisms to enable the development of libraries of shareable knowledge modules and the reuse of their knowledge-level contents. Today's knowledge bases are structured as monolithic networks of highly interconnected symbols, designed for specific tasks in narrow domains. As a result, it is difficult to adapt existing knowledge bases to new uses or even to identify the shareable contents. To enable the accumulation of shareable knowledge and the use of this knowledge by multiple systems, we need a means for designing composable modules of knowledge. The working group is chartered to identify the barriers to the building and use of shared knowledge modules, characterize potential approaches to overcoming these barriers, and conduct experiments exploring mechanisms for knowledge sharing.

**Approach.** The working group supports three kinds of activity. One is the identification of important research issues for knowledge sharing, including problems of methodology (for example, multidisciplinary, collaborative knowledge base design) as well as engineering (for example, scalability, shareability). A second activity is the development of ontologies that define terminology used to represent bodies of shareable knowledge. The task includes (1) identifying bodies of knowledge worth the effort to formally represent and make shareable and (2) defining coherent sets of terms that characterize the ontological commitments and representational choices for modeling these bodies of knowledge. A third type of working group activity is the coordination of collaborative experiments in knowledge sharing in which multiple research groups attempt to share and use each other's knowledge bases (for example, libraries of device models). Some experiments will evaluate the use of ontologies as a mechanism for sharing (that is, for modularity and composability of knowledge modules and the specification of their contents).

**Outcomes.** To these ends, the working group is concentrating on three objectives. The first is a survey of the state of the art in research on knowledge sharing and reuse, which identifies the techniques currently being explored and recommends research on the critical problems to be solved. A second outcome is a set of results from the collaborative experiments on knowledge sharing, including the ontologies used for each experiment and lessons learned about tools and methodologies for developing them. An immediate subgoal for this outcome is to develop a mechanism for representing these ontologies in portable form, building on the work of the other three working groups. The third, more long-term objective is to develop a suite of exemplary shared ontologies and the knowledge bases using them. These ontologies will serve as research test beds, playing a role analogous to the E. *coli* bacterium in biology: a well-understood, complete example on which to perform a variety of experiments.

## Long-Term Research Issues

The preceding description of the working groups focused on near-term issues. However, the goals proposed in this article suggest a number of high-payoff research questions for the entire research community. In this section, we want to focus on longer-term concerns by reviewing some of these questions.

A number of issues raised by this vision were also identified by Ron Brachman (1990) in his AAAI-90 address on the future of knowledge representation. High on his list of issues were knowledge representation services, knowledge base management, and the usability of knowledge representation systems. Brachman pointed out that the idea of a single, general-purpose knowledge representation system with optimal expressive power might not meet real needs. Instead, he urged, we must look at different levels of service and understand what it means to offer them, how to characterize the capabilities of different services, and what the cost is of invoking them. The management of very large knowledge bases poses some fascinating research questions, including methods for handling globally inconsistent but locally reasonable knowledge, higher-level principles for organizing knowledge, and techniques for belief revision as knowledge bases evolve over time. As large knowledge bases come into widespread use, they will need to be built, extended, and used by a wider range of users. Because these users will likely be less forgiving than academic researchers using products of their own making, a number of questions arise concerning presenting knowledge, browsing, querying, and explaining.

Other relevant issues raised by Brachman include the integration of multiple paradigms, extensible representation systems, and efficiency concerns. In addition, each of the four working groups has questions that present challenges to the research community.

The notion of translation implied by interlingua raises a number of questions about tools and methodologies for translation. As the work on interlingua has progressed, a much better understanding has grown about the distinction between communication codes and representation codes (these two terms were introduced into the discussion by Pat Hayes). A *communication code* captures knowledge for the purposes of exchanging it or talking about it, and a *representation code* attempts to capture knowledge for the purpose of supporting the efficient implementation of inference mechanisms. As the effort proceeds, it is likely to spawn a great deal of work on understanding and recording the design rationale behind representation codes to facilitate greater automation in translating them into and out of communication codes.

A number of issues in representation languages remain as research topics. Belief has already been mentioned. Others include defaults and inheritance, negation, disjunction, metadescriptions, higher-order logics, description of inference, predicate reification, causality, and time.

External interfaces present a range of both short- and long-term research issues. Examples include the verification of protocols for asynchronous operation of multiple end user applications with many knowledge bases, the assessment of possible degrees of parallelism, and deeper investigation into requirements for supporting interactive user interfaces. Longer-range issues include mechanisms for specifying the quantity and quality of information to be returned by a knowledge base in response to a request, means for dealing with uncertainty, and methods for optimizing the performance of distributed systems.

Finally, the notion of shared, reusable knowledge bases provides a tremendous amount of grist for the research mill. Most obviously, researchers will be exploring questions about the content of specific reusable ontologies for some time to come. In addition, there are a number of ancillary questions: How is consensus on a group ontology best achieved? How can consensus be maintained as needs change over time? What kinds of automated assistance and interactive tools will be beneficial? How can we verify compatibility with an ontology? How can we support correspondence theories that allow knowledge base developers to express and reason about mappings between different ontologies?

The efforts described in this article to develop conventions for sharing and reuse represent the start—rather than the culmination—of a large body of research activity. We believe that this area is one in which conventions will serve to focus and enable research, not codify and terminate it.

## Conclusion

Attempting to move beyond the capabilities of current knowledge-based systems mandates knowledge bases that are substantially larger than those we have today. However, representing and acquiring knowledge is a difficult and time-consuming task. Knowledge-acquisition

*The time is ripe to start building the infrastructure for integrating AI software at the knowledge level.*

tools and current development methodologies will not make this problem go away because the root of the problem is that knowledge is inherently complex and the task of capturing it is correspondingly complex. Thus, we cannot afford to waste whatever knowledge we do succeed in acquiring. We will be hard pressed to make knowledge bases much bigger than we have today if we continue to start from scratch each time we construct a new system. Building qualitatively bigger knowledge-based systems will be possible only when we are able to share our knowledge and build on each other's labor and experience.

The time is ripe to start building the infrastructure for integrating AI software at the knowledge level, independent of particular implementations. Today, there is a significant body of ongoing work in AI and application domains on pieces of the problem, such as basic knowledge representation; knowledge-acquisition tools; task-specific architectures; and domain-oriented, multiuse models. What is lacking is an integrating framework, the means for describing, connecting, and reusing knowledge-based technology.

Addressing these concerns will open the doors to the development of much larger-scale systems, structured in a fashion that facilitates their development, maintenance, and extension. Furthermore, it will eliminate barriers to embedding AI components in larger, mostly conventional software systems. This approach will lead to a great expansion in the range of applicability for AI technology, which, in turn, will greatly enhance its utility and significantly expand the commercial marketplace.

The knowledge representation technology that supports these goals will have four key characteristics:

First, it will offer libraries of reusable ontologies, that is, knowledge base frameworks consisting of (1) formal definitions of the terms that can be used to model a domain or class of domains and (2) assertions that govern the requirements and constraints for creating valid models within a domain by combining and relating terms and term instances.

Second, it will offer powerful, expressive, and efficient interpreters and compilers for knowledge representation systems (knowledge bases combined with inference mechanisms) in which these ontologies are embedded. These systems will likely be structured as services oriented toward a client-server model of interaction.

Third, it will provide system builders with tools for translating between alternative representation systems. These tools will enable them to create efficient, optimized systems by making choices about alternative implementations without losing the opportunity to reuse representational work from other, different formalisms.

Fourth, it will embed these interpreters and compilers in architectures that support complete interoperability not just between multiple knowledge-based systems but also with conventional software systems. In particular, there will be a convenient, standard application programming interface and tight interconnection with databases.

This article attempted to articulate a vision of the necessary knowledge representation system technology and a path to achieving it. It also argued that progress in this area will dramatically change for the better the way that knowledge-based systems are built and the way they are perceived by their users. Central to this vision is the notion of establishing an information infrastructure for promoting the sharing and reuse of knowledge bases in the development and application of large, enterprise-level software systems.

## Acknowledgments

This effort is supported by NSF grant IRI-900-6923, NASA-Ames contract NCC 2-520, and a cooperative agreement between USC/ISI and the Corporation for National Research Initiatives. Steve Cross, Mike Genesereth, Bob Kahn, Bill Mark, Ron Ohlander, Peter Patel-Schneider, Marty Tenenbaum, and Gio Wiederhold deserve particular acknowledgment for their respective important roles in shaping the efforts presented in this article. The authors would like to acknowledge beneficial discussions with Brad Allen, Giuseppe Attardi, R. Bhaskar, Danny Bobrow, B. Chandrasekaran, Randy Davis, Helen Gigley, Pat Hayes, Phil Hayes, Alan Lawson, Bob Mac Gregor, Elaine Rich, Luc Steels, Mark Stefik, Abe Waksman, Chuck Williams, and Mike Williams. Sheila Coyazo's copyediting assistance was extremely valuable.

## References

Bennett, J. S. 1984. ROGET: Acquiring the Conceptual Structure of a Diagnostic Expert System. In Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems, 83–88. Washington, D.C.: IEEE Computer Society.

Brachman, R. J. 1990. The Future of Knowledge Representation. In Proceedings of the Eighth National Conference on Artificial Intelligence, 1082–1092. Menlo Park, Calif.: American Association for Artificial Intelligence.

Brachman, R. J., and Levesque, H. J. 1984. The Tractability of Subsumption in Frame-Based Description Languages. In Proceedings of the Third National Conference on Artificial Intelligence, 34–37. Menlo Park, Calif.: American Association for Artificial Intelligence.

Brodie, M. 1988. Future Intelligent Information Systems: AI and Database Technologies Working Together. In *Readings in Artificial Intelligence and Databases*, eds. J. Mylopoulos and M. Brodie, 623–641. San Mateo, Calif.: Morgan Kaufmann.

Cargill, C. F. 1989. *Information Technology Standardization: Theory, Process, and Organizations*. Bedford, Mass.: Digital.

Chandrasekaran, B. 1986. Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design. *IEEE Expert* 1(3): 23–30.

Chandrasekaran, B. 1983. Toward a Taxonomy of Problem-Solving Types. *AI Magazine* 4(4): 9–17.

Cutkosky, M. R., and Tenenbaum, J. M. 1990. A Methodology and Computational Framework for Concurrent Product and Process Design. *Mechanism and Machine Theory* 25(3): 365–381.

Feiner, S. K., and McKeown, K. R. 1990. Coordinating Text and Graphics in Explanation Generation. In Proceedings of the Eighth National Conference on Artificial Intelligence, 442–449. Menlo Park, Calif.: American Association for Artificial Intelligence.

Finin, T., and Fritzson, R. 1989. How to Serve Knowledge—Notes on the Design of a Knowledge Server. Presented at the AAAI Spring Symposium on Knowledge System Tools and Languages, 28–30 March, Stanford, Calif.

Forbus, K. 1990. The Qualitative Process Engine. In *Readings in Qualitative Reasoning about Physical Systems*, eds. D. Weld and J. de Kleer, 220–235. San Mateo, Calif.: Morgan Kaufmann.

Gruber, T. R. 1991. An Experiment in the Collaborative Development of Shared Ontology, Technical Report, KSL 91-51, Knowledge Systems Laboratory, Stanford University. Forthcoming.

Guha, R. V., and Lenat, D. B. 1990. CYC: A Mid-Term Report. *AI Magazine* 11(3): 32–59.

Harp, B.; Aberg, P.; Benjamin, D.; Neches, R.; and Szekely, P. 1991. DRAMA: An Application of a Logistics Shell. In Proceedings of the Annual Conference on Artificial Intelligence Applications for Military Logistics, 146-151. Williamsburg, Va.: American Defense Preparedness Association. Forthcoming.

Johnson, W. L., and Harris, D. R. 1990. Requirements Analysis Using ARIES: Themes and Examples. In Proceedings of the Fifth Annual Knowledge-Based Software Assistant Conference, 121–131. Liverpool, N.Y.: Rome Air Development Center.

Kahn, R. E., and Cerf, V. E. 1988. An Open Architecture for a Digital Library System and a Plan for Its Development, Volume I: The World of Knowbots, Technical Report, Corporation for National Research Initiatives, Reston, Virginia.

Kahn, G.; Nowlan, S.; and McDermott, J. 1984. A Foundation for Knowledge Acquisition. In Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems, 89–96. Washington, D.C.: IEEE Computer Society.

McDermott, J. 1990. Developing Software Is Like Talking to Eskimos about Snow. In Proceedings of the Eighth National Conference on Artificial Intelligence, 1130–1133. Menlo Park, Calif.: American Association for Artificial Intelligence.

Mckay, D.; Finin, T.; and O'Hare, A. 1990. The Intelligent Database Interface. In Proceedings of the Eighth National Conference on Artificial Intelligence, 677–684. Menlo Park, Calif.: American Association for Artificial Intelligence.

Pan, J. Y.-C.; Tenenbaum, J. M.; and Glicksman, J. 1989. A Framework for Knowledge-Based Computer-Integrated Manufacturing. *IEEE Transactions on Semiconductor Manufacturing* 2(2): 33–46.

Roth, S., and Mattis, J. 1990. Data Characterization for Intelligent Graphics Presentation. In Proceedings of CHI-90, The Annual Conference on Computer-Human Interaction, 193–200. New York: Association of Computing Machinery.

Sathi, A.; Fox, M. S.; and Greenberg, M. 1990. Representation of Activity Knowledge for Project Management. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-7(5): 531–552.

Smith, D. R. 1990. A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering* SE-16(9): 1024–1043.

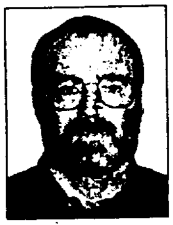
Steele, G. R. 1984. *Common Lisp: The Language*. Bedford, Mass.: Digital.

Steels, L. 1990. Components of Expertise. *AI Magazine* 11(2): 29–49.

Stefik, M. 1986. The Next Knowledge Medium. *AI Magazine* 7(1): 34–46.

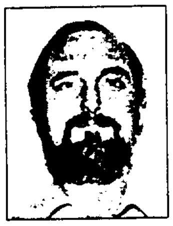
Swartout, W. R., and Smoliar, S. 1989. Explanation: A Source for Guidance in Knowledge Representation. In *Knowledge Representation and Organization in Machine Learning*, ed. K. Morik, 1–16. New York: Springer-Verlag.

Waterman, D. A. 1986. *A Guide to Expert Systems*. Reading, Mass.: Addison-Wesley.



**Robert Neches** is head of the Integrated User-Support Environments Group at USC/Information Sciences Institute and a research associate professor in the Computer Science Department at USC. His current research topics include intelligent interfaces and development methodologies for large systems integrating AI and conventional technology. Since November 1989, he has been serving as coordinator for a university, government, and private industry effort to develop conventions that facilitate the sharing and reuse of AI knowledge bases and knowledge representation system technology.

**Richard Fikes** is a research professor in the Computer Science Department at Stanford University and is co-director of the Heuristic Programming Project in Stanford's Knowledge Systems Laboratory. Fikes is a fellow of the American Association for Artificial Intelligence and is prominently known as co-developer of the STRIPS automatic planning system and one of the principal architects of IntelliCorp's KEE system.



**Tim Finin** is professor and chairman of the Computer Science Department at the University of Maryland. Prior to joining the University of Maryland, he was a technical director at the Unisys Center for Advanced Information Technology, a member of the faculty of the University of Pennsylvania, and on the research staff of the Massachusetts Institute of Technology AI Laboratory. He has had over 20 years of experience in the applications of AI to problems in database and knowledge base systems, expert systems, natural language processing, and intelligent interfaces. Finin holds a B.S. degree in electrical engineering from the Massachusetts Institute of Technology and a Ph.D in computer science from the University of Illinois.

**Thomas Gruber** is a research scientist in the Knowledge Systems Laboratory, Stanford University. His current research interests include the development of shared ontologies for knowledge sharing and reuse, the acquisition and generation of design rationale, and the computer-assisted formulation of engineering models. His earlier work at the University of Massachusetts was in the areas of knowledge-based communication prosthesis and automated knowledge acquisition for expert systems.

**Ramesh Patil** recently joined the Intelligent Systems Division of USC/ Information Sciences Institute as a project leader after serving as an associate professor of computer science at MIT. His Ph.D. work on multilevel causal reasoning systems for medical diagnosis, performed in the Computer Science Department at MIT, was the basis for his receiving the American Association for Medical Systems and Informatics Young Investigator Award in Medical Knowledge Systems. His current research interests include knowledge representation, signal interpretation, qualitative reasoning, and diagnostic reasoning in medicine and electronics.

**Ted Senator** is chief of the Artificial Intelligence Division in the Financial Crimes Enforcement Network of the U.S. Treasury Department. In his prior position with the Department of the Navy, he managed the Navy's applications under the Strategic Computing Initiative.

**William R. Swartout** is director of the Intelligent Systems Division and an associate research professor of computer science at USC/Information Sciences Institute. He is primarily known for his work on explanation and new expert system architectures. Swartout was co–program chair for the 1990 National Conference on Artificial Intelligence and is currently serving on the Board of Councilors of the American Association of Artificial Intelligence.

# APPENDIX B

## Management of Knowledge Representation Standards Activities
## NASA-Ames Contract NCC 2-719

## Final Report covering the period of
## 6/1/1991 - 5/31/1993

## Principal Investigator

Ramesh S. Patil
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
Phone: (310) 822-1511
ramesh@isi.edu

# The DARPA knowledge Sharing Effort: Progress Report

Ramesh Patil
Information Sciences Institute
University of Southern California
Marina del Rey, CA

Richard Fikes
Knowledge Systems Laboratory
Stanford University
Palo Alto, CA

Peter Patel-Schneider
AT&T Bell Laboratories
Murry Hill, New Jersey

Don Mckay
Paramax Systems Corp.
Paoli, PA

Tim Finin
Computer Science Department
University of Maryland
Baltimore, MD

Thomas Gruber
Knowledge Systems Laboratory
Stanford University
Palo Alto, CA

Robert Neches
Information Sciences Institute
University of Southern California
Marina del Rey, CA

**Citation:** Ramesh Patil, Richard Fikes, Peter Patel-Schneider, Don Mckay, Tim Finin, Thomas Gruber, and Robert Neches. The DARPA knowledge Sharing Effort: Progress Report. In the Proceedings of the Third Internatinal Conference on Principles of Knowledge Representation and Reasoning (KR'92) Morgan Kauffman Publishers, Inc. San Mateo, CA. 1992

# The DARPA Knowledge Sharing Effort: Progress Report

**Ramesh S. Patil**
USC Info. Sci. Inst.
Marina del Rey, California

**Richard E. Fikes**
Stanford University
Palo Alto, California

**Peter F. Patel-Schneider**
AT&T Bell Labs
Murray Hill, New Jersey

**Don Mckay**
Paramax Systems Corp.
Paoli, Pennsylvania

**Tim Finin**
Univ. of Maryland
Baltimore, Maryland

**Thomas Gruber**
Stanford University
Palo Alto, California

**Robert Neches**
USC Info. Sci. Inst.
Marina del Rey, California

Building knowledge-based systems today usually entails constructing a new knowledge base from scratch. Even if several groups of researchers are working in the same general area, such as medicine or electronic diagnosis, each team must develop its own knowledge base from scratch. The cost of this duplication of effort has been high and will become prohibitive as we build larger and larger systems. Furthermore, lack of methodology for sharing and communicating knowledge poses a significant road-block in developing large multi-center research projects such as DARPA/Rolm Laboratory Planning and Scheduling Initative [21]. To overcome these barrier and advance the state of the art, we must find ways of preserving existing knowledge bases, and sharing, reusing, and building on them.

The Knowledge-Sharing Effort, sponsored by the Defense Advanced Research Projects Agency (DARPA), The Air Force Office of Scientific Research (AFOSR), the Corporation for National Research Initiative (NRI), and the National Science Foundation (NSF), is an initiative to develop the technical infrastructure to support the sharing of knowledge among systems. [27] The goal of this effort is to develop a technology that will enable researchers to develop new systems by selecting components from library of reusable modules and assembling them together. Their effort will be focused on creating specialized knowledge and reasoners specific to the task of their system. Their new system would inter-operate with existing systems, using them to perform some of its reasoning. In this way, declarative knowledge, problem solving techniques and reasoning services could all be shared among systems. The reusable modules in the library them-selves will benefit from refinements that are only possible through extensive use. This would facilitate building larger systems cheaply and reliably. The infrastructure to support such sharing and reuse would lead to greater ubiquity of these systems, potentially transforming the knowledge industry.

The work in the Knowledge-Sharing Effort began with the identification of the impediments to knowledge sharing and corresponding needs for the development of technology to overcome these impediments. Four key areas were identified for the initial effort. They are: (1) mechanisms for translation between knowledge bases represented in different languages; (2) common versions of languages and reasoning modules within families of representational paradigm; (3) protocols for communication between separate knowledge-based modules, as well as between knowledge-based systems and databases; and, (4) libraries of "ontologies," i.e., pre-fabricated foundations for application-specific knowledge bases in a particular topic area.

A detailed discussion of the impediments, and an analysis of the issues that motivated us to focus on these four types, appears in [27]. That article also describes the working groups (comprised of researchers from the DARPA AI community and other volunteers) that were established to address these issues. The next four sections describe the progress made by each of the four working groups in addressing these issues through the development of draft specifications, implementations and experiments.

## 1 AN INTERLINGUA FOR KNOWLEDGE INTERCHANGE

For a knowledge-based system to incorporate encoded knowledge from a library or to interchange knowledge with another system, the knowledge must either be represented in the receiving system's representation language or be translatable in some practical way into that language. Since an important means of achieving efficiency in application systems is to use specialized representation languages that directly support the knowledge processing requirements of the application, we cannot expect a standard knowledge representation language to emerge that would be used generally in application systems. Thus, we are confronted with a *heterogeneous language problem*. We may, however, be able to deal with that problem by developing a knowledge interchange language that would be commonly used as an *interlingua* for communicating knowledge between computer programs.

Given such an interlingua, a sending system could translate knowledge from its application-specific representation into the interlingua for communication purposes and a receiving system could translate knowledge from the interlingua into its application-specific representation before use. In addition, the interlingua could be the language in which libraries would provide reusable knowledge bases. An interlingua eases the translation problem in that without an interlingua one must write N pairs of translators in order to communicate knowledge to and from N other languages. With an interlingua, one need only write one pair of translators into and out of the interlingua.

## 1.1 KIF – A KNOWLEDGE INTERCHANGE FORMAT

The Interlingua Working Group, chaired by Richard Fikes and Michael Genesereth, is attacking the heterogeneous language problem by developing and testing a language for use as an interlingua called the Knowledge Interchange Format (KIF)[16]. The group began its work by observing that an interlingua needs to be a language with the following general properties:

- A formally defined declarative semantics;
- Sufficient expressive power to represent the declarative knowledge contained in typical application system knowledge bases; and
- A structure that enables semi-automatic translation into and out of typical representation languages.

The working group then merged ongoing language design efforts to produce a preliminary version of the KIF language which could be used as a straw man interlingua in knowledge interchange experiments and design discussions. Since then, the language has been continually evolved and expanded based on feedback from ongoing e-mail discussions, formal design reviews, translation of example knowledge bases, and interoperation experiments.

KIF is an extended version of first order predicate logic. The current 3.0 version of KIF has the following features:

- Simple list-based linear ASCII syntax suitable for transmission on serial media. For example, the following is a KIF sentence:

  ```
  (forall ?x (=> (P ?x) (Q ?x)))
  ```

- Model-theoretic semantics with axiomatic characterization of a large vocabulary of object, function, and relation constants.
- Function and relation vocabulary for numbers, sets, and lists.
- Support for expression of knowledge about the properties of functions and relations. Functions and relations are included in the universe of discourse as sets of lists so that they can be arguments to relations (e.g, **transitive** and **one-one**) and functions (e.g., **inverse** and **range**). In addition, a **holds** relation is included that is true when its first argument denotes a relation that has as a member the list consisting of the items denoted by the remaining arguments. So, for example, one could define transitivity as follows:

  ```
  (<=> (transitive ?r)
       (=> (holds ?r ?x ?y)
           (holds ?r ?y ?z)
           (holds ?r ?x ?z)))
  ```

- A sublanguage for defining objects, n-ary relations, and n-ary functions that enables augmentation of the representational vocabulary and specification of domain ontologies. Definitions can be *complete* in that they specify an equivalent expression or *partial* in that they specify an axiom that restricts the possible denotations of the constant being defined. For example, the following is a complete definition of the unary relation **bachelor**:

  ```
  (defrelation bachelor (?x) :=
     (and (man ?x) (not (married ?x))))
  ```

  and the following is a partial definition of a binary relation **above** which specifies that **above** is transitive and holds only for "located objects":

  ```
  (defrelation above (?b1 ?b2)
     :=> (and (located-object ?b1)
              (located-object ?b2))
     :axiom (transitive above))
  ```

- Support for expression of knowledge about knowledge. KIF expressions are included as objects (i.e., lists) in the universe of discourse, and functions are available for changing level of denotation. For example, the following sentence says that Lisa has the same belief as John about the material of which things are made:

  ```
  (=> (believes john '(material ,?x ,?y))
      (believes lisa '(material ,?x ,?y))
  ```

  and the following sentence says that every sentence of the form (=> $\phi$ $\phi$) is true:

  ```
  (=> (sentence ?p) (true '(=> ,?p ,?p)))
  ```

- A sublanguage for stating both monotonic and nonmonotonic inference rules. For example:

  ```
  (<<= (flies ?x)
       (bird ?x) (consis (flies ?x)))
  ```

A KIF reference manual describing the entire language in detail is available through anonymous FTP from

**hudson.stanford.edu**[17]. The working group expects the current language design to remain relatively stable and for future versions to be essentially extensions to the existing language. Extensions under active consideration include support for uncertain knowledge and contexts, and additional support for default knowledge.

KIF is intended to be a core language which is expandable by defining additional representational primitives. For example, one can define a frame language vocabulary of classes, slots, number restrictions, value restrictions, etc. (as Gruber has done in [19]) so that knowledge can be expressed in a form directly analogous to a frame language. Thus, given suitable definitions, one could define a "guest meal" as being a meal in which there is at least one guest and the food is gourmet as follows:

```
(defrelation guest-meal (?m)
   :=> (and (meal ?m)
            (at-least-fillers ?m guest 1)
            (all-fillers ?m food
                         gourmet-food)))
```

## 1.2 KNOWLEDGE INTERCHANGE EXPERIMENTS USING KIF

The problems involved in interchanging knowledge bases are not yet well understood, and there is open debate as to whether a generally useful interlingua can be specified. The Interlingua Working Group is attempting to inform that debate by developing KIF as a candidate interlingua and by promoting knowledge interchange experiments designed to substantially test the viability and adequacy of KIF as an interlingua. Several small scale experiments have been conducted thus far and multiple projects are underway to build and test KIF translators. These activities, though still in preliminary stages, have already been very productive in identifying issues that need to be resolved and technology that needs to be developed in order for knowledge interchange to be a practical reality. We describe three examples of such activities below.

Ramesh Patil built translators to an early version of KIF from CLASSIC [4] and from LOOM [22]. He then used those translators to produce KIF versions of simple CLASSIC and LOOM knowledge bases. As expected, such translation experiments highlighted weaknesses in KIF and motivated evolution of the language. In general, producing KIF translations of a wide range of sample knowledge bases is an effective means of evaluating the expressive adequacy of KIF and focusing its continuing development. Building the translators themselves does not appear to be problematical. The primary issue is whether KIF has sufficient expressive power to represent the declarative knowledge expressible in the source language.

Translating knowledge *out of* KIF is in general an intractable problem because any given proposition can be expressed in KIF in many equivalent but syntactically different forms and the recognition grammar for a target language will only be able to recognize some subset of those forms. The translation task, therefore, involves applying equivalence preserving rewrite rules. to transform unrecognizable sentences into recognizable forms. Despite the worst-case complexity of logically complete translation, effective translation may be achievable in most situations by logically incomplete techniques combined with interactive direction from the user. To explore that hypothesis, Fikes and Van Baalen are building a translator development "shell" which will contain a grammar-based recognizer, a goal-directed rewrite rule interpreter, a library of general-purpose rewrite rules, facilities for hand translation of problematic sentences, etc. [12]. Initial versions of the basic components of that shell have been implemented and have been used to successfully translate simple KIF knowledge bases into CLASSIC.

A knowledge interchange capability is important both to enable *incorporation* of knowledge into a knowledge-based system (e.g., during system development) and to enable *interoperation* of knowledge-based systems so that they can cooperatively perform tasks and solve problems. KIF is being used as the knowledge level inter-agent communication language in multiple interoperation experiments, including those conducted by Mike Genesereth using the Designworld system [15] and those being conducted by participants in the Palo Alto Collaborative Testbed (PACT).

Designworld is an automated prototyping system for small scale electronic circuits built from standard parts (TTL chips and connectors on prototyping boards). The design for a product is entered into the system via a multi-media design workstation; the product is built by a dedicated robotic cell; and, if necessary, the product, once built, can be returned to the system for diagnosis and repair. The system consists of eighteen processes on six different machines. Each of the eighteen programs is implemented as a distinct agent that communicates with its peers via messages in a KQML-like Agent Communication Language (ACL) that uses KIF as the "content" language.

PACT is a laboratory for exploring the use of knowledge sharing technology and agent-based system integration architectures to support concurrent engineering. Participants include research groups at Stanford University, Lockheed AI Laboratory, Hewlett-Packard Laboratories, and Enterprise Integration Technologies. The initial experiments integrated four preexisting concurrent engineering systems into a common computational framework and explored engineering knowledge exchange in the context of a distributed simulation and simple incremental redesign scenario [9]. In those experiments, each of the individual systems was used to model one or more components of an example programmable electro-mechanical device, a small robotic manipulator. The systems interact via software agents which use KQML as the "performative" language and KIF as the "content" language during knowledge interchange.

Although these experiments have not yet placed severe demands on KIF as an interlingua, KIF successfully provided what was needed, namely a clearly specified logical sentence language for interchange of assertions, queries, and simulation inputs and outputs.

## 2 THE KNOWLEDGE REPRESENTATION SYSTEM SPECIFICATION

Even within a single family of knowledge representation systems (e.g. KL-ONE) minor differences in syntax and semantics between systems pose significant barriers to knowledge sharing. The goal of the Knowledge Representation System Specification (KRSS) group is to develop common specifications for the representational component of families of knowledge representation systems. These specifications will help facilitate the transfer of collections of knowledge between knowledge representation systems in the same family, by reducing the representational differences among systems in the family. The intent of the group is to produce, by-and-large, descriptive specifications, although reconciliation of some syntactic differences will almost certainly be required.

Specifications produced by the group will concentrate on the representational components of the family of knowledge representation systems. Thus, they will provide a complete definition of the representation language underlying the family, but will not include a complete definition of the interface functions that are required in a useful knowledge representation system. Instead the specifications will only define a minimal interface, one that is sufficient to create knowledge bases and query them in limited ways. Also, specifications will completely ignore user-interface issues.

These specifications will definitely not be interlinguas. The representation formalism in the specifications will be specific to the family of representation systems under consideration, and will not be general-purpose representation logics. The specifications also have to be concerned with the computational properties of the formalism they define (i.e., how hard inference in the formalism is), as the aim of the group is to specify knowledge representation *systems*, and not just abstract formalisms.

The initial effort of the KRSS group is the development of a specification for knowledge representation systems based on what are now called description logics (also known as frame-based description languages, terminological logics, etc.). These systems include BACK [31], CLASSIC [6], KRIS [2], and LOOM [22]. This group of systems was chosen partly because there is a large number of systems that are based on description logics (see above), partly because there was already some interest in the community of developers of such systems for a common specification [1], partly because many of the people in the initial group gathered together at the start of the DARPA Knowledge Sharing Initiative were working with such systems, and partly because such systems have a formal basis that is readily amenable to a well-defined specification. There has also been considerable study of the formal properties of reasoning in systems based on description logics. This includes studies of how reasoning should proceed in such systems [26] and the computational complexity and decidability of reasoning in description logics [5, 25, 10, 30]. The presence of such a large body of formal work makes the specification process much easier.

Although there is a common background for all knowledge representation systems based on description logics, there is surprising variance in several dimensions in the systems. First, different systems have different input syntaxes. One goal of the initial KRSS effort is to minimize differences in this dimension. Second, different systems have different interfaces, both functional and user interfaces. Another goal of the initial KRSS effort is to minimize differences in the portion of the functional interface used to construct and directly query knowledge bases. However, the rest of the interfaces of the various systems will not be incorporated into the specification, as it is outside the goals of the KRSS group.

The main difference between the various systems is that they take different positions in the trade-offs among expressive power, completeness of inference, and resource consumption. Some systems try to be as complete as possible in a less-expressive description logic while consuming as few resources as possible, trading off expressive power for computational benefits. Some systems implement complete inference in a moderately-expressive but decidable description logic, trading off possible resource consumption for better expressive power. Some systems implement only partial inference in an expressively-powerful description logic, trading off completeness for expressive power.

Many points in this set of trade-offs are reasonable, so a specification has to allow for both the current set of trade-offs, and also for possible future trade-offs. This means that the specification will not be a complete specification nor even a nearly complete specification.

The approach that has been taken in the specification is to define an expressively powerful description logic, including both a syntax and a semantics, incorporating those constructs whose meaning has been generally agreed upon by the community. Along with the description logic is a set of interface functions that allow for the construction, manipulation, and querying of description-logic knowledge bases. These functions allow

- the formation of descriptions and sentences;
- the definition of concepts and roles from descriptions;
- the assertion of sentences, including ground facts about individuals and simple rules about concepts;
- the creation of individuals and reasoning about their identity;
- the making of local closed-world statements;
- the making of default statements about instances of concepts;
- the retracting of previously-told assertions; and
- the querying of knowledge bases.

The non-query functions are defined by their effect on an abstract knowledge base, which is a collection of statements in the description logic. The results of the query functions are (mostly) defined by semantic relationships between the knowledge base and the query.

Because it is impossible to efficiently perform inference in the full description logic, conforming systems are not required to completely implement it. Conforming systems are free to recognize only a subset of the syntax of the logic, and need not even perform complete reasoning in the subset that they do recognize. However, such systems must use this logic as the ideal meaning of their knowledge bases, and must perform "sound" reasoning with respect to the logic.

Conforming systems are not completely free in what portion of the logic they choose to address. There is a core portion of the logic that all conforming systems are required to implement; in this way a minimal competence is required for all conforming systems. The core is not just a syntactic subset of the full logic—complete inference on even very minimal subsets of the logic is very difficult—but is instead a set of constructs that must be recognized, along with a set of inferences that must be performed on these constructs.

Most of the debate on the specification has involved the details of this core. The constructs to include in the core, the inferences to perform on them, and how to specify these inferences have all been subjects of debate. This was to be expected, as the specification of the core is where the specification is making decisions on matters that have been decided in different ways by different systems. Devising a core that is both reasonable and non-trivial is an interesting exercise in how to balance various representation and implementation concerns.

There is now (July 1992) a second draft of the complete specification that has been distributed to interested parties. Some changes still need to be made to this draft. First, formal work in description logics has advanced, and should be incorporated into the specification. Second, there are portions of the draft, particularly in the inferences required in the core, that are objectionable to some parties. By September 1992, there should be a third draft prepared and discussed, and by the end of October 1992 a final version of the specification should be available. Also, a method for demonstrating compliance with the specification will be developed.

Future work in the KRSS group effort on description-logic based systems will then consist of augmenting the specification as new formal work on description logic produces relevant results and as new implementation techniques make it possible to extend the core. Also, other families of knowledge representation systems may be given the same treatment, provided that developers are interested.

## 3 KNOWLEDGE QUERY AND MANIPULATION LANGUAGE (KQML)

The *External Interfaces* working group was originally charged with addressing the general problem of defining standard high-level interfaces for knowledge representation systems. This was seen as including such diverse interfaces as those to other KR systems, DBMSs, active sensors, and human users. Over the past two years, this working group has focused on and experimented with a somewhat narrowed and more focused problem – designing a common high-level language (KQML) and associated protocol which can be used by software systems for the run-time sharing of information and knowledge. This section briefly describes the current status of the effort to specify KQML and experiment with its use in several testbeds.

### 3.1 OVERVIEW

The Knowledge Query and Manipulation Language (KQML) is both a message format and a message-handling protocol to support run-time knowledge sharing among agents. KQML can be used as a language for an application program to interact with an intelligent system or for two or more intelligent systems to share knowledge in support of cooperative problem solving. KQML focuses on an extensible set of *performatives*, which defines the permissible operations that agents may attempt on each other's knowledge and goal stores. The performatives comprise a substrate on which to develop higher-level models of interagent interaction such as contract nets and negotiation [8, 33].

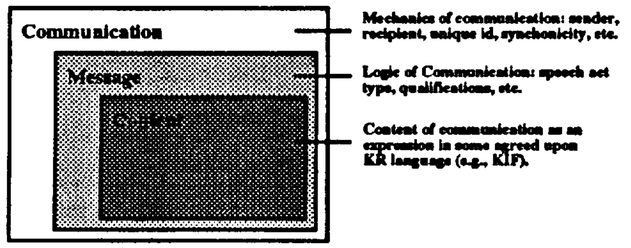

Communication
Message
Content
Mechanics of communication: sender, recipient, unique id, synchonicity, etc.
Logic of Communication: speech act type, qualifications, etc.
Content of communication as an expression in some agreed upon KR language (e.g., KIF).

Figure 1: KQML expressions can be thought of as consisting of a content expression encapsulated in a message wrapper which is in turn encapsulated in a communication wrapper.

In addition, KQML provides a basic architecture for knowledge sharing through a special class of agent called *communication facilitators.* These agents coordinate the interactions of other agents by providing such functions as:

- identification of other agents with which to communicate both explicitly via "names" or "addresses" or implicitly via declared topics of interest or capabilities,
- maintaining registration databases of knowledge services offered and sought by agents,
- communication services (e.g., forwarding information from one agent to other interested agents), and
- content translation to bridge semantic and ontologic differences between end agents.

These functions are embodied in special performatives (which take messages as arguments), and in the way that facilitators treat messages received from application agents.

The ideas which underly the evolving design of KQML are currently being explored through experimental prototype systems which are being used to support two testbeds: the Palo Alto Collaborative Testbed (PACT) [9] which is focused in the concurrent engineering domain, and the DARPA/Rome Planning Initiative (DRPI) which deals with military transportation planning [13].

## 3.2 KQML EXPRESSIONS ARE LAYERED

KQML expressions consist of a content expression encapsulated in a message wrapper which is in turn encapsulated in a communication wrapper, as shown in Figure 1. Thus the language is thought of as being divided into three layers: content, message and communication. The content layer contains an expression in some language which encodes the knowledge to be conveyed. The format of this expression is unimportant to KQML; it can carry any type of content expressed in any representation language which follows some general syntactic constraints (currently, the content expression must be an s-expression). However, there are emerging conventions for knowledge representation (e.g., Interlingua, KIF [17], etc) and standards for *persistent objects* (e.g., the OMG Object Request Broker) which may prove to be very valuable in the near future.

The primary purpose of the message layer is to identify the speech act or performative that the sender attaches to the content, such as an assertion, a query or a command, and any of a small set of qualifiers that may be appropriate to that performative. In addition, since the the content is opaque to KQML, this layer also includes optional features describing the content's language, the ontology it assumes and a descriptor naming a topic within the ontology. These features make it possible for the protocol implementation to analyze, route and properly deliver messages even though their content may be inaccessible.

The final communication level adds a second layer of features to the message which describe the lower level communication parameters, such as the identity of the sender and recipient, a unique identifier associated with the communication and whether the communication is meant to be synchronous or asynchronous. These are used by the network layer which provides reliable transfer of bytes between processes on a network.

## 3.3 KQML PERFORMATIVES

The message layer is used to encode a message that one application would like to have transmitted to another application and forms the core of the language, determining the kinds of interactions one can have with a KQML-speaking agent. It can be thought of as a "speech act layer", since an important attributes to specify about the content is what kind of "speech act" it represents – an assertion, a query, a response, an error message, etc.

**Structure.** Conceptually, a KQML message consists of an operator or *performative*, its associated arguments which constitute the real *content* of the message and a set of optional arguments which describe the content in a standard, language-independent manner. For example, a message representing a query about the location of an particular airport might be encoded as:

```
(ask (geoloc lax (?long ?lat))
     :number_answers 1
     :ontology drpi_geo)
```

In this message, the KQML performative is *ask*, the content (i.e., knowledge being sought) is $(geoloclax(?long?lat))$, the number of answers requested is 1, the language in which the content is expressed is (by default) *kif* and the ontology to be assumed is that named by the token $drpi_geo$. The same general query could be conveyed in using standard Prolog as the content language in a form that requests the set of all answers as:

```
(ask "geoloc(lax, (Long,Lat))"
     :language standard_Prolog
     :number_answers all
     :ontology drpi_geo)
```

**Semantics.** It is our intention to allow the set of KQML performatives to be extensible. We will identify a core set of performatives that will have a well defined meaning. An KQML-speaking agent need not implement or handle all of the performatives in this core, but for those it does, it must adhere to the standard semantics. Moreover, it is our goal to provide a standard mechanism by which one can define the semantics of new performatives, allowing the set to be extended. The semantics of the core performatives will be defined in terms of a smaller set of *primitive performatives.* The semantics of these primitive performatives are defined with respect to a simple and general model of agents in which each agent as a store of information structures (i.e., "belief" like items) and a store of goals structures (i.e., items which may effect the agent's future behavior).

**Primitive Performatives.** We are currently working with a set of four primitive performatives from which we believe the core and various interesting extensions can be defined. These four primitives provide operators to present an agent with items to add ($ADVISE$) and remove ($UNADVISE$) from its information store and to add ($ACHIEVE$) and remove from ($FORGET$) its goal store. These four performatives are primarily used as a means to specify the semantics of the larger core performatives.

**Core Performatives.** The core set of performatives is expected to include several dozen operators which most KQML-speaking agents will support. If an agent accepts a message with a core performative, it must adhere to its agreed upon semantics. Some of these performatives will accept optional arguments which serve as qualifier. Figure 2 shows some examples of performatives that are in the current specification.

**Messaging via Facilitators.** Any substantial collection of interacting agents will require some structure on information flow [20, 28, 32]. For this reason, KQML introduces a class of communication facilitator agents that help manage the message traffic among application agents. Facilitator agents can route performatives to appropriate agents (MONITOR performatives in particular), record the performative-processing abilities of new agents, and bridge the capabilities of superficially incompatible agents (through buffering, translation, and problem decomposition). These facilitation functions will be reflected in new core performatives, e.g., (**FORWARD** *agent-name message*) and (**DISTRIBUTE** *message*).

- **(ASSERT P)** - Add P to the agent's information store, performing whatever reasoning the agent can perform.
- **(RETRACT P)** - Remove P from the agent's information store if present, signalling an error if not present and performing whatever reasoning the agent can perform.
- **(ASK P)** - Query the agent's information store to find answers matching query P. The number of answers returned is governed by an optional argument.
- **(GENERATOR P)** - Reply with a *generator* that the recipient can use to elicit a stream of answers to the query P.
- **(MONITOR P)** - Modify the agent's goal store to cause it to inform the sender whenever a sentence matching P becomes true.

Figure 2: These are a few of the core KQML performatives.

**Software Architecture.** As Figure 3 shows, a typical KQML-speaking agent will be built using two reusable pieces – an interface between the agent's system language (e.g., LOOM or Prolog) which ties communication actions to system actions, and a router which handles the low-level communication chores necessary to talk to other agents. These might all be done within a single process (e.g., in Lisp) or might include several processes (e.g., the router might be done in C or Perl).

## 3.4 STATUS AND OPEN ISSUES

The ideas which underly the evolving design of KQML are currently being explored through experimental prototype systems which are being used to support

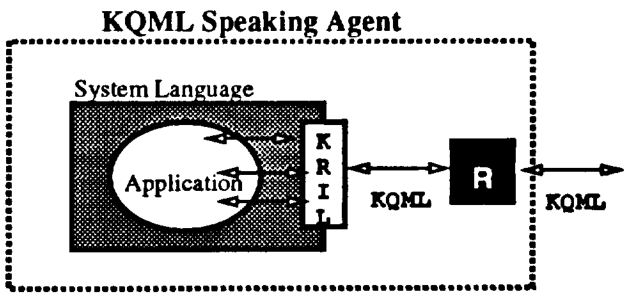


Figure 3: A typical KQML-speaking agent will be built using two reusable pieces – an interface between the agent's system language (e.g., LOOM or Prolog) which ties communication actions to system actions, and a router which handles the low-level communication chores necessary to talk to other agents.

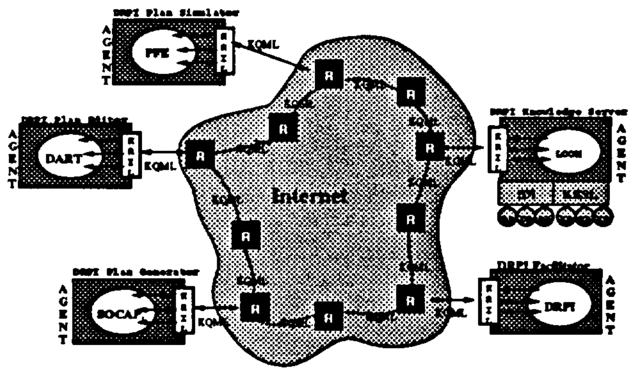


Figure 4: KQML will be used as communication language among the various agents which make up the DRPI testbed. It will be used, for example, to support the interchange of knowledge among the planner, the plan simulator, the plan editor and the DRPI knowledge server which is the repository for the shared ontology and access point for common databases.

two testbeds: the Palo Alto Collaborative Testbed (PACT) [9] which is focused in the concurrent engineering domain, and the DARPA/Rome Planning Initiative (DRPI) which deals with military transportation planning.

**KQML use in PACT.** The Palo Alto Collaborative Testbed (PACT) uses KQML as its medium for agent interaction in support of concurrent engineering. PACT participants modified several existing knowledge-based engineering systems to speak KQML and thereby exchange design and manufacturing knowledge of mutual interest. (For example, the mechanical modeler sends the controls modeler knowledge regarding the dynamics of the design; the power modeler sends the manufacturing process planner knowledge regarding a motor replacement.) These agents find each other in part through facilitators, which handle message forwarding, content-based routing, and simple format translations.

**KQML use in DRPI.** The DARPA/Rome Planning Initiative is using KQML as the communication language among the various agents that make up the testbed and feasibility demonstrations. Figure 4 shows KQML being used, for example, to support the interchange of knowledge among the planner, the plan simulator, the plan editor and the DRPI knowledge server, which is the repository for the shared ontology [21] and access point to common databases through the *Intelligent Database Interface* [23, 29]

**Open Issues.** The design of KQML has continued to evolve as the ideas are explored and feedback is received from the prototypes and the attempts to use them in real testbed situations. We mention here a few of the important issues that we expect to be addressed in the coming year.

The core set of performatives is still undergoing revision as we experiment with its use. This set needs to be stabilized and well specified. In particular, we need to refine the model of what a communication facilitator is and what services it might offer so that we develop a good set of performatives to support their effective use.

A method for defining new extensions to the core set needs to be worked out. This includes a method for defining them for humans as well as a method to allow one agent to define a new performative to another.

The basic model of a knowledge representation agent that we have been working with is quite simple. One of several extensions that may be needed, for example, is a mechanism to define contexts within an agents information and goal stores.

An important part of KQML will be the protocols associated with the different performatives. There are some general issues which go beyond defining the semantics of particular performatives that must be addressed. These general protocols include such things as refusing to accept a message, error reporting, security, and transaction oriented processing.

## 4 SHARED, REUSABLE KNOWLEDGE BASES

The SRKB Working Group (Shared, Reusable Knowledge Bases) of the DARPA Knowledge effort is working on the problem of sharing the *content* of formally represented knowledge. Sharing content requires more than a formalism (KIF) and communication protocol (KQML). Of course, understanding the nature of what needs to be held in common between communicating agents, or between the author of a book and its reader, is a fundamental question for philosophy and science. The SRKB group is focusing on the practical problem of building knowledge-based software that can be shared and reused as off-the-shelf technology. The charter of the group is to identify the technical barriers to the sharing and reuse of formally represented knowledge by AI programs, and to provide a forum for experimentation with possible approaches.

### 4.1 STRATEGY: COMMON ONTOLOGIES AS A SHARING MECHANISM

The strategy is to focus on common ontology as the sharing mechanism [27, 18]. What is a common ontology? Every knowledge-based system is based on some conceptualization of the world: those objects, processes, qualities, distinctions, and relationships that matter for performing some task. A program (or its programmer) makes ontological commitments to a conceptualization by embodying these concepts, distinctions, etc. in a formal representation and using knowledge formulated in that representation during problem solving. By **common ontology** we mean an explicit specification of a the ontological commitments of a set of programs. Such a specification is an objective description—interpretable outside of the programs—of the concepts and relationships that the programs assume and use when interacting with other programs, knowledge bases, and human users.

Operationally, a common ontology can be specified as a set of definitions of representational terms used to construct expressions in a knowledge base, such as classes, relations, slots, and object constants. To make a common ontology shareable, the definitions should consist of human-readable text and machine-enforceable, declarative constraints (i.e., axioms) on the well-formed use of the terminology. The set of terms in a common ontology need not include all the terms used internally in participating programs. Rather, the shared vocabulary defined in a ontology is used for specifying the coupling between programs and knowledge bases (at design time) and for knowledge-level communication among agents (at run time). We hope to enable large-scale sharing and reuse of knowledge bases and knowledge based systems by making common ontologies available as open specifications, much like interchange formats and communication protocols.

The initial activities of the working group have been to explore the research issues in knowledge sharing, and to identify areas where it might be practical and useful to specify common ontologies. The Summer Ontology Project, held at Stanford in 1990, studied the collaborative, multi-disciplinary development of reusable ontologies for describing electromechanical devices and their designs. One outcome was the observation that several approaches to device modeling, from digital circuit modeling to rigid body dynamics, seemed to make commitments to lumped-element models of physical devices. In a lumped-element model, the behavior of a device is described in terms of values of functions (state variables) that map a single independent variable (e.g., time, but not space) to physical quantities (position, force, etc.). A preliminary ontology was proposed to formalize these concepts.

In March of 1991, the SRKB group met at Pajaro Dunes to characterize some of the research issues. There was some controversy about whether it is premature to "standardize" ontologies of any sort, especially those designed to be comprehensive over tasks and domains. Instead, a series of collaborative, grassroots experiments were proposed, in which two or more research groups identify potential candidates for knowledge sharing.

In the past year, several collaborations have begun, and a set of ad hoc subgroups have been formed to study these ontological niches. Each subgroup is tasked with identifying, collecting, making available, and analyzing ontologies for knowledge sharing. We will describe the efforts of these groups within a framework of models of sharing and reuse.

## 4.2 MODELS OF KNOWLEDGE SHARING AND REUSE

Three models of sharing and reuse are being explored, and in each, common ontologies play an enabling role.

First is the **library model**, in which bodies of formally represented knowledge are available as off-the-shelf products, like books in a library. In this model, knowledge bases are designed artifacts, and the role of SRKB to help make them available and reusable.

Two ad hoc subgroups are currently active within the library model of sharing. One is an effort by representatives of projects in qualitative physics to specify a common language for model fragments. Model fragments are conceptual building blocks for programs that formulate and assemble engineering models of device behavior, using techniques such as compositional modeling [11]. For example, idealized components such as resistors and physical processes such as liquid flow are represented by model fragments, which are composed to produce simulation models of complete systems. The language under development is a unification of model formulation and simulation systems such as QPE, DME, and QPC, and should enable a community library of model fragments that can be directly executed by these systems. The axiomatic semantics of the language will be expressed in KIF, and the ontological commitments of these programs will be specified as an ontology.

A second subgroup, following up on the Summer Ontology Project, is developing a family of ontologies for specifying various styles of engineering modeling. It is formalizing the classes of algebras used in constraints (e.g., with or without differential equations; qualitative operators), the assumptions underlying component/connection topologies, and the various styles of dynamics analysis (e.g., Newtonian, LaGrangian, Kane's method). This work is complementary to the composition modeling effort; any of these styles of modeling can be formulated using the model fragment language.

A preliminary finding is that the ontological commitments of a given approach to modeling may be factored into separate ontologies. These ontologies form an inclusion hierarchy, where each ontology can inherit (by set inclusion) the definitions of included ontologies. For example, the original proposal for a lumped-element ontology has since been divided into several ontologies, including *continuous-state-space* (commits to describing behavior using state variables) and *hierarchical-component-assembly* (objects are structured into components related by connections and part-of relations). To specify how state variables are associated with components, one writes a third ontology that includes the other two, adding a few additional constraints. To support this sort of modular partitioning of ontologies, the interlingua committee is considering context mechanisms such as Cyc's microtheories.

A second mode of sharing and reuse under investigation is the **software engineering model**. A standard approach to making software reusable is to decompose complex programs into modular pieces, and to provide a formal specification of the inputs, outputs, and function computed by each piece. Knowledge-based systems are like other software in this respect, except that they operate on a special input called the "background knowledge base" or "domain theory." Reusable modules are designed so that the same code can be used on several knowledge bases. However, to write these knowledge bases the developer needs to understand the ontological assumptions and commitments made in the code. An ontology that defines the vocabulary with which to write the knowledge bases can help determine which software module to use on a given problem, how to provide it the necessary domain knowledge, and whether the knowledge base meets the input requirements of the software.

A significant effort is under way in the knowledge acquisition community to formally characterize the tasks being performed by knowledge based systems, and to design modular problem-solving methods that can be combined to address these tasks [24]. For example, complex, amorphous tasks such as diagnosis and planning have been decomposed into more generic subtasks that can be solved with reusable methods such as simple classification, abductive assembly, and varieties of constraint satisfaction. An subgroup led by Mark Musen is studying ways to describe these tasks and methods, and has begun to define ontologies that specify the input and output assumptions of reusable methods.

A second subgroup, headed by Ed Hovy and Doug Skuce, is identifying and analyzing the comprehensive, *top-level* ontologies that are intended to be general across domains and tasks. A motivating application for such ontologies is natural language processing. NLP techniques needs a way to couple to domain knowledge bases (for something to talk about) without committing the programs to particular subject matter areas. For example, the Penman language generation system's "Upper Structure" ontology [3] divides the world up according to the major type distinctions made in English and German (Objects of various types, Processes and Relations of various types, Qualities, etc.). A developer customizes Penman to a particular application domain by defining the domain's concepts as specializations of the appropriate Upper Structure concepts. As a result, the domain concepts inherit the necessary linguistic annotations from their Upper Structure ancestors. In general, such top-level ontologies can be viewed as a software reuse mechanism for programs parameterized by large knowledge bases.

Another subgroup is looking at ontologies that specify semiformal representations of decision making and design rationale (Jeff Bradshaw, Jin Tae Lee, and Charles Petrie). In semiformal rationale support systems, users organize text describing design decisions in to a hypertext document that supports a fixed vocabulary of node types (classes) and link types (relations). For example, in the gIBIS ontology [7], decisions are described in terms of *issues*, *arguments*, and *positions*, and these node types are linked by relations such as *supports* and *objects-to*. The documents structured by these terms are called semiformal or semistructured, since only the node and link types are machine interpretable and the contents of the nodes are not formalized. Several methodologies for developing semiformal documents, and tools to support them, are based on these ontologies of node and link types.

A third kind of sharing and reuse is the **reference model**, typically used to define an integration framework for a family of application programs. A reference model defines the concepts in a domain and/or problem area that are common to the set of application tasks. For example, a reference model for digital circuits includes a formalism for describing the netlist, which is a representation of circuit topology. The reference model ontology commits the participating tools to the existence of shared objects such as components connected by ports in a netlist; this is necessary to enable tools to exchange data.

An international standards effort called PDES/STEP is working on a family of reference-model ontologies for product data, starting by defining primitives for geometry and working toward high level descriptions of behavior and functionality. The DARPA knowledge sharing effort is exploring avenues for collaboration with the PDES organization.

Within the SRKB working group, ad hoc subgroups are studying reference-model ontologies for user interface toolkits (Jim Foley and Bob Neches), manufacturing enterprise models (Mark Fox), and planning/scheduling (Don McKay, Masahiro Hori).

## 4.3 TECHNICAL SUPPORT FOR ONTOLOGIES – ONTOLINGUA

Each of the subgroups of the SRKB are charged with identifying and collecting ontologies, and making them available in a form amenable to analysis and possible reuse. However, existing ontologies are either incompletely formalized or written in a specific knowledge representation tool. To address this problem, a system called Ontolingua has been developed [19]. Ontolingua is a mechanism for defining ontologies *portably*, that is, independent of specific representation systems. It allows the definition of classes, relations, and distinguished objects using KIF sentences, and translates these definitions into several implemented representation systems.

Ontolingua's design demonstrates the use of a common ontology to facilitate sharing and reuse (in this case, of ontologies). Translation from a very expressive language (KIF) into restricted languages is inherently incomplete. Therefore, Ontolingua supports a subset of legal sentences that can be translated into a class of commonly-used representation systems: the object-centered or frame-based systems. These implemented systems commit to particular ways of organizing and specifying knowledge about objects, such as inheritance hierarchies and slot descriptions. These ontological commitments are captured in the Frame Ontology, which defines a vocabulary for describing classes, binary relations, and second-order relationships among them (e.g., subclass, instance, class partitions, slot-value restrictions). Ontolingua recognizes the use of Frame Ontology concepts in KIF sentences, and translates them into the special syntax of each target representation system. The Frame Ontology, on top of a syntactically restricted KIF, defines a language for portable ontologies. The Ontolingua software operationalizes the language by providing automatic translation into implemented representation systems.

## 5 SUMMARY

Moving beyond the capabilities of current knowledge-based systems will require development of knowledge bases that are substantially larger than those we have today. It will require knowledge-based systems to communicate with other knowledge-based systems and conventional software systems in carrying out their functions. Meeting these challenges on a broad scale will require development new knowledge-sharing technology and shared conventions. The on-going efforts in the Knowledge Sharing Effort represent steps in this directions. The efforts underway are neither complete nor comprehensive – they represent an initial first steps that will result in valuable experience and understanding, will identify shortcomings in current methods and point to new research directions, will encourage others to focus on solving problems encountered in knowledge sharing, to explore alternatives and to enhance the state of the art.

### Acknowledgments

This effort is supported by NSF grant IRI-9006923, DARPA/NASA-Ames contract NCC 2-719, and a cooperative agreement between USC/ISI and the Corporation for National Research Initiative. We would also like to acknowledge members of the Knowledge Sharing Effort, too numerous to name individually.

## References

[1] Franz Baader, Hans-Jürgen Bürckert, Jochen Heinsohn, Bernhard Hollunder, Jürgen Müller, Bernhard Nebel, Werner Nutt, and Hans-Jürgen Profitlich. Terminological knowledge representation: A proposal for a terminological logic. A DFKI note., June 1991.

[2] Franz Baader and Bernhard Hollunder. KRIS: Knowledge Representation and Inference System—system description. Technical Memo TM-90-13, Deutsches Forschungszentrum für Künstliche Intelligenz, November 1990.

[3] John A. Bateman. Upper modeling: A general organization of knowledge for natural language processing. Penman development note, USC/Information Sciences Institute, 1989.

[4] A. Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnick. CLASSIC: A structural data model for objects. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, Portland, Oregon, 1989.

[5] Ronald J. Brachman and Hector J. Levesque. The tractability of subsumption in frame-based description languages. In *Proceedings of the Fourth National Conference on Artificial Intelligence*, pages 34–37, Austin, Texas, August 1984. American Association for Artificial Intelligence.

[6] Ronald J. Brachman, Deborah L. McGuinness, Peter F. Patel-Schneider, Lori Alperin Resnick, and Alex Borgida. Living with CLASSIC: When and how to use a KL-ONE-like language. In Sowa [34], pages 401–456.

[7] Jeff Conklin and M. L. Begeman. gIBIS: A hypertext tool for exploratory policy discussion. In *Proceedings of the 1988 Conference on Computer Supported Cooperative Work (CSCW-88)*, pages 140–152, Portland, Oregon, 1988. ACM.

[8] Susan E. Conry, Robert A. Meyer, and Victor R. Lesser. Multistage negotiation in distributed planning. In Alan H. Bond and Les Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 367–384. Morgan Kaufman, 1988.

[9] Mark Cutkosky, Robert Engelmore, Richard Fikes, Thomas Gruber, Micheal Genesereth, William Mark, Jay Tenenbaum, and Jay Weber. Pact: An experiment in integrating concurrent engineering systems. *IEEE Computer*, 1992. To appear in a special issue on computer-supported concurrent engineering.

[10] Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Werner Nutt. The complexity of concept languages. In *Proceedings of the Second In-*

*ternational Conference on Principles of Knowledge Representation and Reasoning*, pages 151–162. Morgan Kaufmann, May 1991.

[11] Brian Falkenhainer and Ken Forbus. Compositional modeling: Finding the right model for the job. *Artificial Intelligence*, 51:95–143, 1991.

[12] Richard Fikes, Mark Cutkosky, Tom Gruber, and Jeffrey Van Baalen. Knowledge sharing technology project overview. Technical Report KSL 91-71, Stanford University, Knowledge Systems Laboratory, 1991.

[13] T. Finin, R. Fritzson, and D. McKay et. al. A language and protocol to support intelligent agent interoperability. In *Proceedings of the CE & CALS Washington '92 Conference*, June 1992.

[14] T. Finin, R. Fritzson, and D. McKay et. al. An overview of KQML: A knowledge query and manipulation language. Technical report, Department of Computer Science, University of Maryland Baltimore County, 1992.

[15] Michael R. Genesereth. Designworld. In *Proceedings of the 1991 International Conference on Robotiocs and Automation*, pages 2785–2788, 1991.

[16] Michael R. Genesereth. Knowledge interchange format. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the Conference of the Principles of Knowledge Representation and Reasoning*, pages 599–600. Morgan Kaufmann Publishers, Inc., 1991.

[17] Michael R. Genesereth, Richard E. Fikes, and et al. Knowledge interchange format, version 3.0 reference manual. Technical Report Logic-92-1, Computer Science Department, Stanford University, 1992.

[18] Thomas R. Gruber. The role of common ontology in achieving sharable, reusable knowledge bases. In J. A. Allen, R. Fikes, and E. Sandewall, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference*, pages 601–602, Cambridge, MA, 1991. Morgan Kaufmann.

[19] Thomas R. Gruber. Ontolingua: A mechanism to support portable ontologies. Technical Report KSL 91-66, Stanford University, Knowledge Systems Laboratory, 1992. June 1992 Revision.

[20] Michael N. Huhns, David M. Bridgeland, and Natraj V. Arni. A DAI communication aide. Technical Report ACT-RA-317-90, Microelectronics and Computer Technology Corporation, Microelectronics and Computer Technology Corporation, 3500 West Balcones Center Drive, Austin TX 78759-6509, October 1990.

[21] Nancy Lehrer. DARPA/Rolm Laboratory Planning and Scheduling Initiative, Knowledge Representation Specification Language: KRSL Version 2.0. Language specification and manual, 1992.

[22] Robert MacGregor. Loom users manual. Working Paper ISI/WP-22, USC/Information Sciences Institute, 1990.

[23] Don McKay, Tim Finin, and Anthony O'Hare. The intelligent database interface. In *Proceedings of the $7^{th}$ National Conference on Artificial Intelligence*, August 1990.

[24] Mark A. Musen. Overcoming the limitations of role-limiting methods. *Knowledge Acquisition*, 4(2):165–170, 1992.

[25] Bernhard Nebel. Terminological reasoning is inherently intractable. *Artificial Intelligence*, 43(2):235–249, May 1990.

[26] Bernhard Nebel. Terminological cycles: Semantics and computational properties. In Sowa [34].

[27] Robert Neches, Richard Fikes, Tim Finin, Thomas Gruber, Ramesh Patil, Ted Senator, and William R. Swartout. Enabling technology for knowledge sharing. *AI Magazine*, 12(3):16–36, 1991.

[28] Mike P. Papazoglou and Timos K. Sellis. An organizational framework for cooperating intelligent information systems. *International Journal on Intelligent and Cooperative Information Systems*, 1(1), (to appear) 1992.

[29] J. Pastor, D. McKay, and T. Finin. Viewconcepts: Knowledge-based access to databases. In *Proceedings of the First International Conference on Information and Knowledgement*, November 1992.

[30] Peter F. Patel-Schneider. Undecidability of subsumption in NIKL. *Artificial Intelligence*, 39(2):263–272, June 1989.

[31] Christof Peltason, Albrecht Schmiedel, Carsten Kindermann, and Joachim Quantz. The BACK system revisited. KIT-Report 75, Department of Computer Science, Technische Universität Berlin, September 1989.

[32] Kirk Sayre and Michael A. Gray. Backtalk: A generalized dynamic communication system for DAI. Technical Report CSIS-91-004, The American University, Washington DC, August 1991.

[33] Sandip Sen and Edmund H. Durfee. A formal study of distributed meeting scheduling: preliminary results. In *Proceedings of the ACM Conference on Organizational Computing Systems*, pages 55–68, November 1991.

[34] John Sowa, editor. *Principles of Semantic Networks: Explorations in the representation of knowledge*. Morgan-Kaufmann, San Mateo, California, 1991.

# APPENDIX C

## Management of Knowledge Representation Standards Activities NASA-Ames Contract NCC 2-719

## Final Report covering the period of 6/1/1991 - 5/31/1993

## Principal Investigator

Ramesh S. Patil
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
Phone: (310) 822-1511
ramesh@isi.edu

# Knowledge Interchange Format
# Version 3.0
# Reference Manual

Michael R. Geneserath and Richard E. Fikes
Stanford University
Palo Alto, California

Logic Group
Report Logic-92-1

First Draft January 1992
Current Version June 1992

# Knowledge Interchange Format Version 3.0 Reference Manual

*by*

Michael R. Genesereth
Richard E. Fikes

*in collaboration with*

Daniel Bobrow
Ronald Brachman
Thomas Gruber
Patrick Hayes
Reed Letsinger
Vladimir Lifschitz
Robert MacGregor
John McCarthy
Peter Norvig
Ramesh Patil
Len Schubert

This manual is the "living document" of the Interlingua Working Group of the DARPA Knowledge Sharing Effort. As such, it represents work in progress toward a proposal for a standard knowledge interchange format.

Computer Science Department
Stanford University
Stanford, California 94305

Abstract: *Knowledge Interchange Format* (KIF) is a computer-oriented language for the interchange of knowledge among disparate programs. It has declarative semantics (i.e. the meaning of expressions in the representation can be understood without appeal to an interpreter for manipulating those expressions); it is logically comprehensive (i.e. it provides for the expression of arbitrary sentences in the first-order predicate calculus); it provides for the representation of knowledge about the representation of knowledge; it provides for the representation of nonmonotonic reasoning rules; and it provides for the definition of objects, functions, and relations.

## Table of Contents

# Chapter 1
# Introduction

*Knowledge Interchange Format* (KIF) is a formal language for the interchange of knowledge among disparate computer programs (written by different programmers, at different times, in different languages, and so forth).

KIF is *not* intended as a primary language for interaction with human users (though it can be used for this purpose). Different programs can interact with their users in whatever forms are most appropriate to their applications (for example frames, graphs, charts, tables, diagrams, natural language, and so forth).

KIF is also *not* intended to be an internal representation for knowledge *within* computer programs or within closely related sets of programs (though it can be used for this purpose as well). Typically, when a program reads a knowledge base in KIF, it converts the data into its own internal form (specialized pointer structures, arrays, etc.). All computation is done using these internal forms. When the program needs to communicate with another program, it maps its internal data structures into KIF.

The purpose of KIF is roughly analogous to that of Postscript. Postscript is commonly used by text and graphics formatting programs in communicating information about documents to printers. Although it is not as efficient as a specialized representation for documents and not as perspicuous as a specialized wysiwyg display, Postscript is a programmer-readable representation that facilitates the independent development of formatting programs and printers. While KIF is not as efficient as a specialized representation for knowledge nor as perspicuous as a specialized display (when printed in its list form), it too is a programmer-readable language and thereby facilitates the independent development of knowledge-manipulation programs.

The definition of KIF is highly detailed. Some of these details are essential; others are arbitrary. The following general features are essential in the definition of KIF.

1. The language has declarative semantics. It is possible to understand the meaning of expressions in the language without appeal to an interpreter for manipulating those expressions. In this way, KIF differs from other languages that are based on specific interpreters, such as Emycin and Prolog.
2. The language is logically comprehensive – it provides for the expression of arbitrary sentences in predicate calculus. In this way, it differs from relational database languages (many of which are confined to ground atomic sentences) and Prolog-like languages (that are confined to Horn clauses).
3. The language provides for the representation of knowledge about the representation of knowledge. This allows us to make all knowledge representation decisions explicit and permits us to introduce new knowledge representation constructs without changing the language.

In addition to these hard criteria, KIF is designed to maximize in a joint fashion the following somewhat softer measures as well (to the extent possible while satisfying the preceding criteria).

1. Translatability. A central operational requirement for KIF is that it enable practical means of translating declarative knowledge bases to and from typical knowledge representation languages.
2. Readability. Although KIF is not intended primarily as a language for interaction with humans, human readability facilitates its use in describing representation language semantics, its use as a publication language for example knowledge bases, its use in assisting humans with knowledge base translation problems, etc.
3. Useability as a representation language. Although KIF is not intended for use within programs as a representation or communication language, it *can* be used for that purpose if so desired.

This document supplies full technical details of KIF. Chapter 2 presents the formal syntax of the language. Chapter 3 discusses conceptualizations of the world. Chapter 4 defines the semantics of the language. Chapter 5 deals with lists; chapter 6, with sets; and chapter 7, with functions and relations. Chapter 8 describes how metaknowledge is encoded. Chapter 9 describes the formalization of monotonic and nonmonotonic rules of inference. Chapter 10 discusses definitions.

# Chapter 2
# Syntax

Like many computer-oriented languages, KIF has two varieties. In *linear* KIF, all expressions are strings of ASCII characters and, as such, are suitable for storage on serial devices (such as magnetic disks) and for transmission on serial media (such as phone lines). In *structured* KIF, the legal "expressions" of the language are structured objects. Structured KIF is of special use in communication between programs operating in the same address space.

Fortunately, there is a simple correspondence between the two varieties of KIF. For every character string, there is exactly one corresponding list structure; and, for every list structure, there is exactly one corresponding character string (once all unnecessary white space is eliminated).

In what follows, we first define the mapping between the linear and structured forms of the language; and, thereafter, we deal exclusively with the structured form.

## §2.1 Linear KIF

The alphabet of linear KIF consists of the 128 characters in the ASCII character set. Some of these characters have standard print representations; others do not. The characters with standard print representations (93 of the 128) are shown below.

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 ( ) [ ] { } < >
= + - * / \ & ^ ~ ' ` " _ @ # $ % : ; , . ! ?
```

KIF originated in a Lisp application and inherits its syntax from Lisp. The relationship between linear KIF and structured KIF is most easily specified by appeal to the Common Lisp reader [Steele]. In particular, a string of ascii characters forms a legal expression in linear KIF if and only if (1) it is acceptable to the Common Lisp reader (as defined in Steele's book) and (2) the structure produced by the Common Lisp reader is a legal expression of structured KIF (as defined in the next section).

## §2.2 Structured KIF

In structured KIF, the notion of *word* is taken as primitive. An *expression* is either a word or a finite sequence of expressions. In our treatment here, we use enclosing parentheses to bound the items in a composite expression.

`<word> ::=` *a primitive syntactic object*

`<expression> ::= <word> | (<expression>*)`

The set of all words is divided into the categories listed below. This categorization is disjoint and exhaustive. Every word is a member of one and only one category. (The categories defined here are used again in the grammatical rules of subsequent tables.)

`<indvar> ::=` *a word beginning with the character* `?`

`<seqvar> ::=` *a word beginning with the character* `@`

```
<termop> ::= listof | setof | quote | if | cond |
             the | setofall | kappa | lambda
```

`<sentop> ::= = | /= | not | and | or | => | <= | <=> | forall | exists`

`<ruleop> ::= =>> | <<= | consis`

`<defop> ::= defobject | defunction | defrelation | := | :=> | :&`

`<objconst> ::=` *a word denoting an object*

`<funconst> ::=` *a word denoting a function*

`<relconst> ::=` *a word denoting a relation*

`<logconst> ::=` *a word denoting a truth value*

From these fundamental categories, we can build up more complex categories, viz. variables, operators, and constants.

`<variable> ::= <indvar> | <seqvar>`

`<operator> ::= <termop> | <sentop> | <ruleop> | <defop>`

`<constant> ::= <objconst> | <funconst> | <relconst> | <logconst>`

A *variable* is a word in which the first character is ? or @. A variable that begins with ? is called an *individual variable*. A variable that begins with an @ is called a *sequence variable*. Individual variables are used in quantifying over individual objects. Sequence variables are used in quantifying over sequences of objects.

*Operators* are used in forming complex expressions of various sorts. There are four types of operators in KIF – *term operators*, *sentence operators*, *rule operators*, and *definition operators*. Term operators are used in forming complex terms. Sentence operators are used in forming complex sentences. Rule operators are using in forming rules. Definition operators are used in forming definitions.

A *constant* is any word that is neither a variable nor an operator. There are four categories of constants in KIF – object constants, function constants, relation constants, and logical constants. *Object constants* are used to denote individual objects. *Function constants* denote functions on those objects. *Relation constants* denote relations. *Logical constants* express conditions about the world and are either true or false.

Some constants are *basic* in that their type and meaning are fixed in the definition of KIF. All other constants are *non-basic* in that the language user gets to choose the type and the meaning. All numbers, characters, and strings are basic constants in KIF; the remaining basic constants are described in the remaining chapters of this document.

KIF is unusual among logical languages in that there is no way of determining the category of a non-basic constant (i.e. whether it is an object, function, relation, or logical constant) from its inherent properties (i.e. its spelling). The user selects the category of every non-basic constant for himself. The user need not declare that choice explicitly. However, the category of a constant determines how it can be used in forming expressions, and its category *can* be determined from this use. Consequently, once a constant is used in a particular way, its category becomes fixed.

There are four special types of expressions in the language – *terms*, *sentences*, *rules*, and *definitions*. Terms are used to denote objects in the world being described; sentences are used to express facts about the world; rules are used to express legal steps of inference; and definitions are used to define constants; and forms are either sentences, rules, or definitions.

The set of legal terms in KIF is defined below. There are ten types of terms – individual variables, object constants, function constants, relation constants, functional terms, list terms, set terms, quotations, logical terms, and quantified terms. Individual variables, object constants, function constants, and relation constants were discussed earlier.

```
<term> ::= <indvar> | <objconst> | <funconst> | <relconst>|
           <funterm> | <listterm> | <setterm> |
           <quoterm> | <logterm> | <quanterm>

<listterm> ::= (listof <term>* [<seqvar>])

<setterm> ::= (setof <term>* [<seqvar>])

<funterm> ::= (<funconst> <term>* [<seqvar>])

<quoterm> ::= (quote <expression>)

<logterm> ::= (if <sentence> <term> [<term>])|
              (cond (<sentence> <term>) ... (<sentence> <term>))

<quanterm> ::= (the <term> <sentence>)|
               (setofall <term> <sentence>)|
               (kappa (<indvar>* [<seqvar>]) <sentence>*)|
               (lambda (<indvar>* [<seqvar>]s) <term>)
```

A *functional term* consists of a function constant and an arbitrary number of *argument* terms, terminated by an optional sequence variable. Note that there is no syntactic restriction on the number of argument terms – the same function constant can be applied to different numbers of arguments; arity restrictions in KIF are treated semantically.

A *list term* consists of the `listof` operator and a finite list of terms, terminated by an optional sequence variable.

A *set term* consists of the `setof` operator and a finite list of terms, terminated by an optional sequence variable.

*Quotations* involve the `quote` operator and an arbitrary list expression. The embedded expression can be an arbitrary list structure; it need *not* be a legal expression in KIF. Remember that the Lisp reader converts strings of the form '$\sigma$ into (quote $\sigma$).

*Logical terms* involve the `if` and `cond` operators. The `if` form allows for the testing of a single condition only, whereas the `cond` form allows for the testing of a sequence of conditions.

*Quantified terms* involve the operators `the`, `setofall`, `kappa`, and `lambda`. A *designator* consists of the `the` operator, a term, and a sentence. A *set-forming term* consist of the `setof` operator, a term, and a sentence. A *relation-forming term* consists of `kappa`, a list of variables, and a sentence. A *function-forming term* consists of `lambda`, a list of variables, and a term. Strictly speaking, we do not need `kappa` and `lambda` – both can be defined in terms of `setof`; they are included in KIF for the sake of convenience.

The following BNF defines the set of legal sentences in KIF. There are six types of sentences. We have already mentioned logical constants.

```
<sentence> ::= <logconst>|<equation>|<inequality>|
               <relsent>|<logsent>|<quantsent>

<equation> ::= (= <term> <term>)

<inequality> ::= (/= <term> <term>)

<relsent> ::= (<relconst> <term>* [<seqvar>])|
              (<funconst> <term>* <term>)

<logsent> ::= (not <sentence>)|
              (and <sentence>*)|
              (or <sentence>*)|
              (=> <sentence>* <sentence>)|
              (<= <sentence> <sentence>*)|
              (<=> <sentence> <sentence>)

<quantsent> ::= (forall <indvar> <sentence>)|
                (forall (<indvar>*) <sentence>)|
                (exists <indvar> <sentence>)|
                (exists (<indvar>*) <sentence>)
```

An *equation* consists of the `=` operator and two terms.

An *inequality* consist of the `/=` operator and two terms.

A *relational sentence* consists of a relation constant and an arbitrary number of *argument* terms, terminated by an optional sequence variable. As with functional terms, there is no syntactic restriction on the number of argument terms in a relation sentence – the same relation constant can be applied to any finite number of arguments.

The syntax of *logical sentences* depends on the logical operator involved. A sentence involving the `not` operator is called a *negation*. A sentence involving the `and` operator is called a *conjunction*, and the arguments are called *conjuncts*. A sentence involving the `or` operator is called a *disjunction*, and the arguments are called *disjuncts*. A sentence involving the `=>` operator is called an *implication*; all of its arguments but the last are called *antecedents*; and the last argument is called the *consequent*. A sentence involving the `<=` operator is called a *reverse implication*; its first argument is called the *consequent*; and the remaining arguments are called the *antecedents*. A sentence involving the `<=>` operator is called an *equivalence*.

There are two types of *quantified sentences* – a *universally quantified sentence* is signalled by the use of the `forall` operator, and an *existentially quantified sentence* is signalled by the use of the `exists` operator.

The following BNF defines the set of legal KIF rules.

```
<rule> ::= (=>> <premise>* <sentence>) |
           (<<= <sentence> <premise>*)

<premise> ::= <sentence> | (consis <sentence>)
```

The last argument in a forward rule is called the *consequent* of the rule. Analogously, the first argument in a reverse rule is called the *consequent*. The premises that are sentences are its *prerequisites*, and the premises that have the form (`consis` $\phi$) are its *justifications*.

The following BNF defines the set of legal KIF definitions.

```
<definition> ::= <complete> | <partial>

<complete> ::=
 (defobject <objconst> := <term>) |
 (deffunction <funconst> (<indvar>* [<seqvar>]) := <term>) |
 (defrelation <relconst> (<indvar>* [<seqvar>]) := <sentence>)

<partial> :: <conservative> | <unrestricted>

<conservative> ::=
 (defobject <objconst> [:conservative-axiom <sentence>]) |
 (deffunction <funconst> [:conservative-axiom <sentence>]) |
 (defrelation <relconst> [:conservative-axiom <sentence>])|
 (defrelation <relconst> (<indvar>* [<seqvar>])
   :=><sentence> [:conservative-axiom <sentence>])

<unrestricted> ::=
 (defobject <objconst> <sentence>*) |
 (deffunction <funconst> <sentence>*) |
 (defrelation <relconst> <sentence>*) |
 (defrelation <relconst> (<indvar>* [<seqvar>])
   :=> <sentence> [:axiom <sentence>])
```

*Definitions* are used to make category declarations and specify *defining axioms* for constants (e.g. "A triangle is a polygon with 3 sides."). KIF definitions can be *complete* in that they specify an expression that defines the concept completely, or they can be *partial* in that they constrain the concept without necessarily giving a complete equivalence. Partial definitions can be either *conservative* or *unrestricted*. Conservative definitions are restricted in that their addition to a knowledge base does not result in the logical entailment of any additional sentences not containing the constant being defined.

Object constants are defined using the `defobject` operator by specifying (1) a term that is equivalent to the constant or (2) a sentence that provides a partial description of the object denoted by the constant. Function constants are defined using the `deffunction` operator by specifying (1) a term that is equivalent to the function applied to a given set of arguments or (2) a sentence that provides a partial description of the function denoted by the constant. Relation constants are defined using the `defrelation` operator by specifying (1) necessary and sufficient conditions for the relation to hold, (2) necessary conditions for the relation to hold, or (3) arbitrary sentences describing the relation.

A *form* in KIF is either a sentence, a rule, or a definition.

```
<form> ::= <sentence> | <definition> | <rule>
```

A *knowledge base* is a finite set of forms. It is important to keep in mind that a knowledge base is a *set* of sentences, not a *sequence*; the order of forms within the knowledge base is unimportant.

# Chapter 3
# Conceptualization

The formalization of knowledge in KIF, as in any declarative representation, requires a *conceptualization* of the world in terms of objects, functions, and relations.

## §3.1 Objects

A *universe of discourse* is the set of all objects presumed or hypothesized to exist in the world. The notion of *object* used here is quite broad. Objects can be concrete (e.g. a specific carbon atom, Confucius, the Sun) or abstract (e.g. the number 2, the set of all integers, the concept of justice). Objects can be primitive or composite (e.g. a circuit that consists of many subcircuits). Objects can even be fictional (e.g. a unicorn, Sherlock Holmes).

Different users of a declarative representation language, like KIF, are likely to have different universes of discourse. KIF is *conceptually promiscuous* in that it does *not* require every user to share the same universe of discourse. On the other hand, KIF is *conceptually grounded* in that every universe of discourse *is* required to include certain *basic* objects.

The following basic objects must occur in every universe of discourse.

- Words. Yes, the words of KIF are themselves objects in the universe of discourse, along with the things they denote.
- All complex numbers.
- All finite lists of objects in the universe of discourse.
- All sets of objects in the universe of discourse.
- $\bot$ (pronounced "bottom") – a distinguished object that occurs as the value of various functions when applied to arguments for which the functions make no sense.

Remember, however, that to these basic elements, the user can add whatever *non-basic* objects seem useful.

## §3.2 Functions and Relations

A function is one kind of interrelationship among objects. For every finite sequence of objects (called the *arguments*), a *function* associates a unique object (called the *value*). More formally, a function is defined as a set of finite lists of objects, one for each combination of possible arguments. In each list, the initial elements are the arguments, and the final element is the value. For example, the 1+ function contains the list $\langle 2, 3\rangle$, indicating that integer successor of 2 is 3.

A relation is another kind of interrelationship among objects in the universe of discourse. More formally, a *relation* is an arbitrary set of finite lists of objects (of possibly varying lengths). Each list is a selection of objects that jointly satisfy the relation. For example, the $<$ relation on numbers contains the list $\langle 2, 3\rangle$, indicating that 2 is less than 3.

Note that both functions and relations are defined as sets of lists. In fact, every function is a relation. However, not every relation is a function. In a function, there cannot be two lists that disagree on only the last element. This would be tantamount to the function having two values for one combination of arguments. By contrast, in a relation, there can be any number of lists that agree on all but the last element. For example, the list $\langle 2, 3\rangle$ is a member of the 1+ function, and there is no other list of length 2 with 2 as its first argument, i.e. there is only one successor for 2. By contrast, the $<$ relation contains the lists $\langle 2, 3\rangle$, $\langle 2, 4\rangle$, and so forth, indicating that 2 is less than 3, 4, and so forth.

Many mathematicians require that functions and relations have fixed arity, i.e they require that all of the lists comprising a function or relation have the same length. The definitions here allow for functions and relations with variable arity, i.e. it is perfectly acceptable for a function or a relation to contain lists of different lengths. For example, the $+$ function contains the lists $\langle 1, 1, 2\rangle$ and $\langle 1, 1, 1, 3\rangle$, reflecting the fact that the sum of 1 and 1 is 2 and the fact that the sum of 1 and 1 and 1 is 3. Similarly, the relation $<$ contains the lists $\langle 1, 2\rangle$ and $\langle 1, 2, 3\rangle$, reflecting the fact that 1 is less than 2 and the fact that 1 is less than 2 and 2 is less than 3. This flexibility is not essential, but it is extremely convenient and poses no significant theoretical problems.

# Chapter 4
# Semantics

Intuitively, the semantics of KIF is very simple. Unfortunately, the formal details are quite complex. Consequently, we proceed gradually in our presentation. In this chapter, we introduce the basic notions underlying the semantics of KIF (in particular, the notions of interpretation, variable assignment, semantic value, truth value, and various types of entailment).

The basis for KIF semantics is a correlation between the terms and sentences of the language and a conceptualization of the world. Every term denotes an object in the universe of discourse associated with the conceptualization, and every sentence is either true or false.

When we encode knowledge in KIF, we select constants on the basis of our understanding of their meanings. In some cases (e.g. the basic constants of the language), these meanings are fixed in the definition of the language. In other cases (i.e. the non-basic constants), the meanings can vary from one user to another.

Given exact meanings for the constants of the language (whether they are the meanings in the definition of the language or our own concoctions), the semantics of KIF tells us the meaning of its complex expressions. We can unambiguously determine the referent of any term, and we can unambiguously determine the truth or falsity of any sentence.

Unfortunately, few of us have complete knowledge about the world. In keeping with traditional logical semantics, this is equivalent to not knowing the exact referent for every constant in the language. In such situations, we write sentences that reflect all of the meanings consistent with whatever knowledge we have. In such situations, the semantics of the language cannot pick out exact meanings for all expressions in the language, but it does place constraints on the meanings of complex expressions.

And, of course, the meanings we ascribe to non-basic constants may differ from those ascribed by others. However, we can convey our meanings to others by writing sentences to constrain those meanings in accordance with our usage. By writing more and more sentences, the set of possible referents for our constants is decreased.

In the remainder of this section, we provide precise definitions for the ideas just introduced. We start off with a definition for the *interpretation* of constants, and we introduce the related notion of *variable assignment.* We then show how these concepts are used in defining the *semantic value* of terms and the *truth value* of sentences. Finally, we introduce several approaches to *entailment*, which eliminates the dependence of meaning on the interpretation of non-basic constants.

## §4.1 Interpretation

An *interpretation* is a function $i$ that associates the constants of KIF with the elements of a conceptualization. In order to be an interpretation, a function must satisfy the following two properties.

First, the function must map constants into concepts of the appropriate type. It must map object constants into objects in the universe of discourse. It must map function constants into functions on the universe of discourse. It must map relation constants into relations on the universe of discourse. Notice that we allow for functions and relations of variable, finite arity. The function must map logical constants into one of the boolean values *true* or *false* (which may or may not be members of the universe of discourse).

1. If $\sigma$ is an object constant, then $i(\sigma) \in O$.
2. If $\sigma$ is a function constant, then $i(\sigma) : O^* \longrightarrow O$.
3. If $\sigma$ is a relation constant, then $i(\sigma) \subseteq O^*$.
3. If $\sigma$ is a logical constant, then $i(\sigma) \in \{true, false\}$.

Second, $i$ must "satisfy" the conditions and axioms given in this chapter and the remaining chapters of this document. As a start, this includes the following conditions.

Every interpretation must map every numerical constant $\sigma$ into the corresponding number $n$ (assuming base 10).

$$i(\sigma) = n$$

Every interpretation must map the object constant `bottom` into $\perp$.

$$i(\texttt{bottom}) = \perp$$

Every interpretation must map the logical constant `true` into *true* and the logical constant `false` into *false*.

$$i(\texttt{true}) = true$$

$$i(\texttt{false}) = false$$

Note that, even with these restrictions, KIF is only a "partially interpreted" language. Although the interpretations of some constants (the basic constants) are constrained in the definition of the language, the meanings of other constants (the non-basic constants) are left open (i.e. left to the imaginations of the language users).

## §4.2 Variable Assignment

A *variable assignment* is a function that (1) maps individual variables $\mathcal{V}$ into objects in a universe of discourse $O$ and (2) maps sequence variables $\mathcal{W}$ into finite sequences of objects.

$$v : \mathcal{V} \longrightarrow O$$

$$v : \mathcal{W} \longrightarrow O^*$$

The notion of a variable assignment is important in defining the meaning of quantified terms and sentences and is discussed further below.

## §4.3 Semantic Value

Given an interpretation and a variable assignment, we can assign a *semantic value* to every term in the language. We formalize this as a function $s_{iv}$ from the set $\mathcal{T}$ of terms into the set $O$ of objects in the universe of discourse.

$$s_{iv} : \mathcal{T} \longrightarrow O$$

If an expression is an individual variable $\nu$, the semantic value is the object assigned to that variable by the given variable assignment.

$$s_{iv}(\nu) = v(\nu)$$

The semantic value of an object constant $\sigma$ is the object assigned to that constant by the given interpretation.

$$s_{iv}(\sigma) = i(\sigma)$$

The semantic value of a function constant $\pi$ is the set of tuples in the universe of discourse corresponding to the function denoted by $\pi$. Here, we use the operator `lambda` to denote this function. A full description of the semantics of expressions involving `lambda` is given later.

$$s_{iv}(\pi) = s_{iv}(\texttt{(lambda (@l) (}\pi\ \texttt{@l))})$$

The semantic value of a relation constant $\rho$ is the set of tuples in the universe of discourse corresponding to the relation denoted by $\rho$. Here, we use the operator `kappa` to denote this relation. A full description of the semantics of expressions involving `kappa` is given later.

$$s_{iv}(\rho) = s_{iv}(\texttt{(kappa (@l) (}\rho\ \texttt{@l))})$$

In most cases, the semantic value of a function or relation constant is the same as its interpretation. However, in order to avoid paradoxes, it must in some cases be different. See the chapter on sets for a fuller discussion of this subject.

The semantic value of a functional term without a terminating sequence variable is obtained by applying the function denoted by the function constant in the term to the objects denoted by the arguments.

$$s_{iv}(\texttt{(}\pi\ \tau_1 \ldots \tau_n\texttt{)}) = i(\pi)[s_{iv}(\tau_1), \ldots, s_{iv}(\tau_n)]$$

If a functional term has a terminating sequence variable, the semantic value is obtained by applying the function to the sequence of arguments formed from the values of the terms that precede the sequence variable and the values in the sequence denoted by the sequence variable. (The vertical bar | here means that the objects in the sequence following the bar are appended to the sequence of elements before the bar.)

$$s_{iv}(\texttt{(}\pi\ \tau_1 \ldots \tau_n\ \omega\texttt{)}) = i(\pi)[s_{iv}(\tau_1), \ldots, s_{iv}(\tau_n) | s_{iv}(\omega)]$$

A term that begins with *listof* refers to the sequence of objects denoted by the arguments in the term. There is no restriction on the objects in the sequence.

$$s_{iv}(\texttt{(listof}\ \tau_1\ \ldots\ \tau_k\texttt{)}) = \langle s_{iv}(\tau_1), \ldots, s_{iv}(\tau_k)\rangle$$

If a term that begins with `listof` ends with a sequence variable, the value of the term as a whole is the sequence consisting of the objects denoted by the terms prior to the sequence variable together with the objects in the sequence denoted by the sequence variable.

$$s_{iv}(\texttt{(listof}\ \tau_1\ \ldots\ \tau_k\ \omega\texttt{)}) = \langle s_{iv}(\tau_1), \ldots, s_{iv}(\tau_k) | s_{iv}(\omega)\rangle$$

A term that begins with *setof* refers to the set of "bounded" objects denoted by the arguments in the term. The concept of boundedness is discussed further in the chapter on sets.

$$s_{iv}(\texttt{(setof}\ \tau_1\ \ldots\ \tau_k\texttt{)}) = \{s_{iv}(\tau_1), \ldots, s_{iv}(\tau_k)\}$$

If a term that begins with `setof` ends with a sequence variable, the value of the term as a whole is the set consisting of the bounded objects denoted by the terms prior to the sequence variable together with the bounded objects in the sequence denoted by the sequence variable.

$$s_{iv}(\texttt{(setof}\ \tau_1\ \ldots\ \tau_k\ \omega\texttt{)}) = \{s_{iv}(\tau_1), \ldots, s_{iv}(\tau_k)\} \cup \{x | x = s_{iv}(\omega)_n\}$$

A quotation denotes the expression contained as argument of the `quote` operator. Remember that the universe of discourse for every interpretation must contain all list expressions and that the argument to `quote` can be any list expression, whether or not it is a legal expression in KIF.

$$s_{iv}(\texttt{(quote}\ \epsilon\texttt{)}) = \epsilon$$

Note that any KIF expression (other than a word) is a sequence of KIF expressions. Thus, there are two ways it can be denoted - with `quote` and with `listof`. This means we have the following equivalence.

$$s_{iv}(\texttt{(quote}\ \texttt{(}\epsilon_1\ \ldots\ \epsilon_n\texttt{)))} = s_{iv}(\texttt{(listof}\ \texttt{(quote}\ \epsilon_1\texttt{)}\ \ldots\ \texttt{(quote}\ \epsilon_n\texttt{)))}$$

The semantic value of a simple conditional term depends on the truth value of the embedded sentence (see next section). If the truth value of the embedded sentence is *true*, then the semantic value of the term as a whole is the semantic value of the first term; otherwise, it is the semantic value of the second term (if there is one).

$$s_{iv}(\texttt{(if}\ \phi\ \tau_1\ \tau_2\texttt{)}) = \begin{cases} s_{iv}(\tau_1) & t_{iv}(\phi) = true \\ s_{iv}(\tau_2) & \text{otherwise} \end{cases}$$

If a simple conditional has only one embedded term and the truth value of the embedded sentence is *true*, then the semantic value of the term is the semantic value of the embedded term. Otherwise, the value is $\perp$.

$$s_{iv}(\texttt{(if}\ \phi\ \tau_1\texttt{)}) = \begin{cases} s_{iv}(\tau_1) & t_{iv}(\phi) = \mathit{true} \\ \perp & \text{otherwise} \end{cases}$$

The semantic value of a complex conditional is the semantic value of the *first* term for which the truth value of the corresponding sentence is *true*. If none of the sentences are true, the semantic value is $\perp$.

$$s_{iv}(\texttt{(cond}\ \texttt{(}\phi_1\ \tau_1\texttt{)}\ \ldots\ \texttt{(}\phi_n\ \tau_n\texttt{))}) = \begin{cases} s_{iv}(\tau_1) & t_{iv}(\phi_1) = \mathit{true} \\ \cdots & \cdots \\ s_{iv}(\tau_2) & t_{iv}(\phi_n) = \mathit{true} \\ \perp & \text{otherwise} \end{cases}$$

The semantic value of a quantified term with an interpretation $i$ and variable assignment $v$ is determined by the semantic value of the embedded term or the truth value of the embedded sentence under the same interpretation but with various new versions of the variable assignment. We say that a variable assignment $v'$ is a *version* of variable assignment $v$ with respect to variables $\nu_1$, ...., $\nu_n$ if and only if $v'$ agrees with $v$ on all variables except for $\nu_1$, ...., $\nu_n$. The assignments for $\nu_1$, ...., $\nu_n$ can be the same as those in $v$ or can be completely different.

The referent of a designator with term $\tau$ as first argument and sentence $\phi$ can be one of two things. Consider all versions $v'$ of $v$ with respect to the free variables in $\tau$. If there is at least one version $v'$ that makes $\phi$ true and the semantic value of $\tau$ is the same in every $v'$ that makes $\phi$ true, then the semantic value of the designator as a whole is that value. If there is more than one such value, the semantic value is $\perp$.

$$s_{iv}(\texttt{(the}\ \tau\ \phi\texttt{)}) = \begin{cases} s_{iv'}(\tau) & t_{iv'}(\phi) = \mathit{true} \text{ and} \\ & s_{iv''}(\tau) = s_{iv'}(\tau) \text{ for all } v''\ t_{iv''}(\phi) = \mathit{true} \\ \perp & \text{otherwise} \end{cases}$$

A set-forming term with the term $\tau$ as first argument and the sentence $\phi$ as second argument denotes the set of objects in the universe of discourse with the following properties. (1) The object must be the semantic value of $\tau$ for some version $v'$ of $v$ that makes $\phi$ true. (2) The object must be *bounded.* A term is *bounded* if and only if it satisfies the interpretation of the `bounded` relation. See the chapter on sets for the axioms characterizing this relation.

$$s_{iv}(\texttt{(setofall}\ \tau\ \phi\texttt{)}) = \{s_{iv'}(\tau) | t_{iv'}(\phi) = \mathit{true}, s_{iv'}(\tau) \in i(\texttt{bounded})\}$$

A function-forming term denotes the set of tuples of bounded objects corresponding to the function that maps every tuple of objects matching the first argument of the term into the semantic value of the second argument.

$$s_{iv}(\texttt{(lambda}\ \texttt{(}\nu_1 \ldots \nu_n\texttt{)}\ \tau\texttt{)}) = s_{iv}(\texttt{(setofall}\ \texttt{(listof}\ \nu_1 \ldots \nu_n\ \nu\texttt{)}\ \texttt{(=}\ \tau\ \nu\texttt{))})$$

If the argument list of the function-forming term terminates in a sequence variable, the semantic value of the term is the union of the infinite series of sets of tuples corresponding to (1) the same term in which all occurrences of the sequence variable are dropped, (2) the same term in which all occurrences of the sequence variable are replaced by a single individual variable, (3) the same term in which all occurrences of the sequence variable are replaced by two individual variables, etc.

A relation-forming term denotes the set of all tuples of bounded objects that satisfy the embedded sentence.

$$s_{iv}(\texttt{(kappa (}\nu_1 \ldots \nu_n\ \texttt{[}\sigma\texttt{])}\ \phi\texttt{)}) = s_{iv}(\texttt{(setofall (listof}\ \nu_1 \ldots \nu_n\ \texttt{[}\sigma\texttt{])}\ \phi\texttt{)})$$

## §4.4 Truth Value

In a manner similar to that for terms, we define the *truth value* for sentences in the language as a function $t_{iv}$ that maps sentences $\mathcal{S}$ into the truth values *true* or *false*.

$$t_{iv} : \mathcal{S} \longrightarrow \{true, false\}$$

The truth value of a logical constant is the truth value assigned by the corresponding interpretation.

$$t_{iv}(\lambda) = i(\lambda)$$

An equation is true if and only if the terms in the equation refer to the same object in the universe of discourse.

$$t_{iv}(\texttt{(=}\ \tau_1\ \ \tau_2\texttt{)}) = \begin{cases} true & s_{iv}(\tau_1) = s_{iv}(\tau_2) \\ false & \text{otherwise} \end{cases}$$

An inequality is true if and only if the terms in the equation refer to distinct objects in the universe of discourse.

$$t_{iv}(\texttt{(/=}\ \tau_1\ \ \tau_2\texttt{)}) = \begin{cases} false & s_{iv}(\tau_1) = s_{iv}(\tau_2) \\ true & \text{otherwise} \end{cases}$$

The truth value of a simple relational sentence without a terminating sequence variable is *true* if and only if the relation denoted by the relation constant in the sentence is true of the objects denoted by the arguments. Equivalently, viewing a relation as a set of tuples, we say that the truth value of a relational sentence is *true* if and only if the tuple of objects formed from the values of the arguments is a member of the set of tuples denoted by the relation constant.

$$t_{iv}(\texttt{(}\rho\ \tau_1\ \ldots\ \tau_n\texttt{)}) = \begin{cases} true & \langle s_{iv}(\tau_1), \ldots, s_{iv}(\tau_n)\rangle \in i(\rho) \\ false & \text{otherwise} \end{cases}$$

If a relational sentence terminates in a sequence variable, the sentence is true if and only if the relation contains the tuple consisting of the values of the terms that precede the sequence variable together with the objects in the sequence denoted by the variable. Remember that the vertical bar | means that the objects in the sequence following the bar are appended to the sequence of elements before the bar.

$$t_{iv}(\texttt{(}\rho\ \tau_1\ \ldots\ \tau_n\ \omega\texttt{)}) = \begin{cases} true & \langle s_{iv}(\tau_1),\ldots,s_{iv}(\tau_n)|s_{iv}(\omega)\rangle \in i(\rho) \\ false & \text{otherwise} \end{cases}$$

The truth value of a negation is *true* if and only if the truth value of the negated sentence is *false*.

$$t_{iv}(\texttt{(not}\ \phi\texttt{)}) = \begin{cases} true & t_{iv}(\phi) = false \\ false & \text{otherwise} \end{cases}$$

The truth value of a conjunction is *true* if and only if the truth value of every conjunct is *true*.

$$t_{iv}(\texttt{(and}\ \phi_1\ \ldots\ \phi_n\texttt{)}) = \begin{cases} true & t_{iv}(\phi_j) = true \text{ for all } j\ 1 \le j \le n \\ false & \text{otherwise} \end{cases}$$

The truth value of a disjunction is *true* if and only if the truth value of at least one of the disjuncts is *true*.

$$t_{iv}(\texttt{(or}\ \phi_1\ \ldots\ \phi_n\texttt{)}) = \begin{cases} true & t_{iv}(\phi_j) = true \text{ for some } j\ 1 \le j \le n \\ false & \text{otherwise} \end{cases}$$

If the truth value of every antecedent in an implication is *true*, then the the truth value of the implication as a whole is *true* if and only if the truth value of the consequent is *true*. If any of the antecedents is *false*, then the implication as a whole is *true*, regardless of the truth value of the consequent.

$$t_{iv}(\texttt{(=>}\ \phi_1\ \ldots\ \phi_n\ \phi\texttt{)}) = \begin{cases} true & \text{for some } j\ t_{iv}(\phi_j) = false \text{ or } t_{iv}(\phi) = true \\ false & \text{otherwise} \end{cases}$$

A reverse implication is just an implication with the consequent and antecedents reversed.

$$t_{iv}(\texttt{(<=}\ \phi\ \phi_1\ \ldots\ \phi_n\texttt{)}) = \begin{cases} true & t_{iv}(\phi) = true \text{ or for some } j\ t_{iv}(\phi_j) = false \\ false & \text{otherwise} \end{cases}$$

The truth value of an equivalence is *true* if and only if the embedded sentences have the same truth value.

$$t_{iv}(\texttt{(<=>}\ \phi_1\ \phi_2\texttt{)}) = \begin{cases} true & t_{iv}(\phi_1) = t_{iv}(\phi_2) \\ false & \text{otherwise} \end{cases}$$

Given an interpretation $i$ and variable assignment $v$, the truth value of an existentially quantified sentence is *true* if and only if the truth value of the second argument is *true* for *some* version $v'$ of variable assignment $v$ with respect to the variables in the first argument.

$$t_{iv}(\texttt{(exists (}\nu_1 \ \ldots \ \nu_k \ \omega\texttt{)} \ \phi\texttt{))} = \begin{cases} \mathit{true} & \exists v' \, t_{iv'}(\phi) = \mathit{true} \\ \mathit{false} & \text{otherwise} \end{cases}$$

Given an interpretation $i$ and variable assignment $v$, the truth value of a universally quantified sentence is *true* if and only if the truth value of the second argument of the sentence is *true* for *every* version $v'$ of $v$ with respect to variables in the first argument.

$$t_{iv}(\texttt{(forall (}\nu_1 \ \ldots \ \nu_k \ \omega\texttt{)} \ \phi\texttt{))} = \begin{cases} \mathit{true} & \forall v' \, t_{iv'}(\phi) = \mathit{true} \\ \mathit{false} & \text{otherwise} \end{cases}$$

## §4.5 Logical Entailment

The definition of truth value relies on both an interpretation for the constants of KIF and an assignment for its variables. In encoding knowledge, we often have in mind a specific interpretation for the constants in our language, but we want our variables to range over the universe of discourse (either existentially or universally). In order to provide a notion of semantics that is independent of the assignment of variables, we turn to the notion of satisfaction.

An interpretation $i$ *logically satisfies* a sentence $\phi$ if and only if the truth value of the sentence is *true* for all variable assignments. Whenever this is the case, we say that $i$ is a *model* of $\phi$. Extending this notion to sets of sentences, we say that an interpretation is a model of a set of sentences if and only if it is a model of every sentence in the set of sentences.

Obviously, a variable assignment has no effect on the truth value of a sentence without free variables (i.e. a ground sentence or one in which all variables are bound). Consequently, if an interpretation satisfies such a sentence for one variable assignment, it satisfies it for every variable assignment.

The occurrence of free variables in a sentence means that the sentence is true for all assignments of the variables. For example, the sentence `(p $x)` means that the relation denoted by `p` is true for all objects in the universe of discourse. In other words, the meaning of a sentence with free variables is the same as the meaning of a universally quantified sentence in which all of the free variables are boundby the universal quantifier. In KIF, we use this fact to sanction the dropping of prefix universal quantifiers that do not occur inside the scope of existential quantifiers. In other words, we are permitted to write `(=> (apple $x) (red $x))` in place of the more cumbersome `(forall ($x) (=> (apple $x) (red $x)))`.

Unfortunately, the notion of satisfaction is disturbing in that it is relative to an interpretation. As a result, different individuals and different programs with different interpretations may disagree on the truth of a sentence.

It is true that, as we add more sentences to a knowledge base, the set of models generally decreases. The goal of knowledge encoding is to write enough sentences so that unwanted interpretations are eliminated. Unfortunately, this is not always possible. In the light of this fact, how are we to interpret the expressions in such situations? The answer is to generalize over interpretations as earlier we generalized over variable assignments.

If $\Delta$ is a set of sentences, we say that $\Delta$ *logically entails* a sentence $\phi$ if and only every model of $\Delta$ is also a model of $\phi$.

With this notion, we can rephrase the goal of knowledge representation as follows. It is to encode enough sentences so that every conclusion we desire is logically entailed by our set of sentences. It is a sad fact that this is not always possible, but it is the ideal toward which we strive.

## §4.6 Indexical Entailment

In the definition of logical entailment, all interpretations are taken into account; there is no constraint. In certain situations, it is desirable to restrict the possible interpretations to those in which certain constants are assigned values having to do with the set of sentences itself. In this case, the constants are said to be *indexical.* An interpretation then is *indexical* if and only if it assigns these indexical constants correctly.

In KIF, there is a single indexical constant, viz. the object constant `knowedge-base`. An indexical interpretation of a knowledge base $\Delta$ is one in which this constant is assigned $\Delta$ as value. This one indexical makes it possible for the user to write sentences that depend on the knowledge base within which the sentences are contained.

Finally, we say that a set of sentences *indexically entails* a conclusion if and only if every indexical interpretation and variable assignment that satisfies the set of sentences also satisfies the conclusion.

## §4.7 Nonmonotonic Entailment

Recall that the truth value of a sentence is defined relative to an interpretation $i$ and a variable assignment $v$. To define the nonmonotonic value of a premise in a rule, we need to select, instead of a single interpretation $i$, a *set* of interpretations – the interpretations that are considered "possible". In the following definition, $I$ is a set of interpretations which all have the same universe of discourse $O$, and $v$ is a variable assignment with this universe. We consider prerequisites and justifications separately.

The nonmonotonic value of a prerequisite is *true* if and only if it is true at every "possible" intepretation.

$$n_{Iv}(\phi) = \begin{cases} true & \forall i \in I \; t_{iv}(\phi) = true \\ false & \text{otherwise} \end{cases}$$

The nonmonotonic value of a justification is *true* if and only if its argument is true for at least one "possible" intepretation.

$$n_{Iv}(\texttt{(consis } \phi\texttt{)}) = \begin{cases} true & \exists i \in I \; t_{iv}(\phi) = true \\ false & \text{otherwise} \end{cases}$$

Let $\Delta$ be a knowledge base with rules. We define when a set $I$ of interpretations is "a set of possible worlds" for $\Delta$, by means of the following fixpoint construction. Consider a universe of discourse $O$; by a *world* we understand an interpretation with the universe $O$. Let $I$ be the set of all worlds that satisfy the sentences in $\Delta$. Consider a maximal set $I'$ of worlds such that, for each rule $\delta \in \Delta$ and each variable assignment $v$ with the universe

$O$, the following condition holds. If the nonmonotonic value of every prerequisite of $\delta$ for $I'$ and $v$ is *true*, and the nonmonotonic value of every justification of $\delta$ for $I$ and $v$ is true, then the nonmonotonic value of the consequent of $\delta$ for $I'$ and $v$ is *true*. (This $I'$ always exists.) If $I'$ is maximal, then we say that $I'$ is *a set of possible worlds* for $\Delta$. Typically, a knowledge base with rules has many sets of possible worlds; it is clear, for instance, that any two interpretations with different universes cannot belong to the same set of possible worlds.

An interpretation $i$ is a *nonmonotonic model* of $\Delta$ if it belongs to some set of possible worlds for $\Delta$. We say that a nonmonotonic knowledge base $\Delta$ *nonmonotonically entails* a sentence $\phi$ if and only every nonmonotonic model of $\Delta$ is also a model of $\phi$.

Note that the definition of a model for nonmonotonic knowledge bases is "nonlocal" – we cannot check whether an interpretation $i$ is a model by looking at each rule in isolation. This feature of the definition is responsible for the nonmonotonic character in this notion of entailment.

## §4.8 Definitions

The definitional operators in KIF allow us to state sentences that are true "by definition" in a way that distinguishes them from sentences that express contingent properties of the world. Definitions have no truth values in the sense described above. They are so because we say that they are so.

On the other hand, definitions have content – sentences that allow us to derive other sentences as conclusions. In KIF, every definition has a corresponding set of sentences, called the *content* of the definition. In general, there are three parts to this content.

First of all, there is information about the category of the constant in the definition. If the constant is a function constant or a relation constant, there is also information about its arity.

Second, there is the *defining axiom* associated with the definition (see below).

Finally, there is a sentence stating that the defining axiom associated with the definition is indeed a defining axiom for the associated concept (named by the constant $\sigma$ used in the definition). The following sentence expresses this fact. Note the use of quotes to capture the fact that this is a relationship between a constant and a sentence.

**(defining-axiom** $'\sigma\ '\phi$)

The rules for determining the defining axioms for a definition are somewhat complicated and are described fully in the chapter on definitions. The following is a brief outline, sufficient to enable the reader to understand the use of definitional constructs in the intervening chapters.

The `defobject` operator is used to define objects. The two simplest forms are shown below, together with their defining axioms. In the first case, the defining axiom is the equation involving the object constant in the definition with the defining term. In the second case, the defining axiom is the conjunction of the constituent sentences.

| Definition | Defining Axiom |
|---|---|
| (defobject $\sigma$ := $\tau$) | (= $\sigma$ $\tau$) |
| (defobject $\sigma$ $\phi_1$ ... $\phi_n$) | (and $\phi_1$ ... $\phi_n$) |

The `deffunction` operator is used to define functions. Again, the two simplest forms are shown below, together with their defining axioms. In the first case, the defining axiom is the equation involving (1) the term formed from the function constant in the definition and the variables in its argument list and (2) the defining term. In the second case, as with object definitions, the defining axiom is the conjunction of the constituent sentences.

| Definition | Defining Axiom |
|---|---|
| (deffunction $\pi$ ($\nu_1$ ...$\nu_n$) := $\tau$) | (= $\pi$ (lambda ($\nu_1$ ...$\nu_n$) $\tau$)) |
| (deffunction $\pi$ $\phi_1$ ... $\phi_n$) | (and $\phi_1$ ... $\phi_n$) |

The `defrelation` operator is used to define relations. The two simplest forms are shown below, together with their defining axioms. In the first case, the defining axiom is the equivalence relating (1) the relational sentence formed from the relation constant in the definition and the variables in its argument list and (2) the defining sentence. In the second case, as with object and function definitions, the defining axiom is the conjunction of the constituent sentences.

| Definition | Defining Axiom |
|---|---|
| (defrelation $\rho$ ($\nu_1$ ...$\nu_n$) := $\phi$) | (= $\rho$ (kappa ($\nu_1$ ...$\nu_n$) $\phi$)) |
| (defrelation $\rho$ $\phi_1$ ... $\phi_n$) | (and $\phi_1$ ... $\phi_n$) |

For most purposes, a definition can be viewed as shorthand for the sentences in the content of the definition.

# Chapter 5
# Numbers

KIF includes the following standard vocabulary for describing properties of numbers. *A formal axiomatization of numbers and of the associated functions and relations is being developed for inclusion in later versions of this manual. Common Lisp is being used as a guide in that development to determine both the types of numbers and the number-related functions and relations to include in the language. The informal descriptions below are provided to indicate the anticipated vocabulary.*

## §5.1 Functions on Numbers

`*` - If $\tau_1$, ..., $\tau_n$ denote numbers, then the term `(*` $\tau_1 \ldots \tau_n$`)` denotes the product of those numbers.

`+` - If $\tau_1$, ..., $\tau_n$ are numerical constants, then the term `(+` $\tau_1 ... \tau_n$`)` denotes the sum $\tau$ of the numbers corresponding to those constants.

`-` - If $\tau$ and $\tau_1$, ..., $\tau_n$ denote numbers, then the term `(-` $\tau$ $\tau_1 ... \tau_n$`)` denotes the difference between the number denoted by $\tau$ and the numbers denoted by $\tau_1$ through $\tau_n$. An exception occurs when $n = 0$, in which case the term denotes the negation of the number denoted by $\tau$.

`/` - If $\tau_1$, ..., $\tau_n$ are numbers, then the term `(/` $\tau_1 ... \tau_n$`)` denotes the result $\tau$ obtained by dividing the number denoted by $\tau_1$ by the numbers denoted by $\tau_2$ through $\tau_n$. An exception occurs when $n = 1$, in which case the term denotes the reciprocal $\tau$ of the number denoted by $\tau_1$.

`1+` - The term `(1+` $\tau$`)` denotes the sum of the object denoted by $\tau$ and 1.

```
(deffunction 1+ (?x) :=  (+ ?x 1))                                    (5.1)
```

`1-` - The term `(1-` $\tau$`)` denotes the difference of the object denoted by $\tau$ and 1.

```
(deffunction 1- (?x) := (- ?x 1))                                     (5.2)
```

`abs` - The term `(abs` $\tau$`)` denotes the absolute value of the object denoted by $\tau$.

```
(deffunction abs (?x) := (if (>= ?x 0) ?x (- ?x)))                    (5.3)
```

`acos` - If $\tau$ denotes a number, then the term `(acos` $\tau$`)` denotes the arc cosine of that number (in radians).

`acosh` - The term `(acosh` $\tau$`)` denotes the arc cosine of the object denoted by $\tau$ (in radians).

`ash` - The term `(ash` $\tau_1$ $\tau_2$`)` denotes the result of arithmetically shifting the object denoted by $\tau_1$ by the number of bits denoted by $\tau_2$ (left or right shifting depending on the sign of $\tau_2$).

`asin` - The term `(asin` $\tau$`)` denotes the arc sine of the object denoted by $\tau$ (in radians).

`asinh` - The term `(asinh` $\tau$`)` denotes the hyperbolic arc sine of the object denoted by $\tau$ (in radians).

`atan` - The term `(atan` $\tau$`)` denotes the arc tangent of the object denoted by $\tau$ (in radians).

`atanh` - The term `(atanh` $\tau$`)` denotes the hyperbolic arc tangent of the object denoted by $\tau$ (in radians).

`boole` - The term `(boole` $\tau$ $\tau_1$ $\tau_2$`)` denotes the result of applying the operation denoted by $\tau$ to the objects denoted by $\tau_1$ and $\tau_2$.

`ceiling` - If $\tau$ denotes a real number, then the term `(ceiling` $\tau$`)` denotes the smallest integer greater than or equal to the number denoted by $\tau$.

`cis` - The term `(cis` $\tau$`)` denotes the complex number denoted by $cos(\tau) + isin(\tau)$. The argument is any non-complex number of radians.

`conjugate` - If $\tau$ denotes a complex number, then the term `(conjugate` $\tau$`)` denotes the complex conjugate of the number denoted by $\tau$.

```
(deffunction conjugate (?c) :=
  (complex-number (realpart ?c) (- (imagpart ?c))))                (5.4)
```

`cos` - The term `(cos` $\tau$`)` denotes the cosine of the object denoted by $\tau$ (in radians).

`cosh` - The term `(cosh` $\tau$`)` denotes the hyperbolic cosine of the object denoted by $\tau$ (in radians).

`decode-float` - The term `(decode-float` $\tau$`)` denotes the mantissa of the object denoted by $\tau$.

`denominator` - The term `(denominator` $\tau$`)` denotes the denominator of the canonical reduced form of the object denoted by $\tau$.

`exp` - The term `(exp` $\tau$`)` denotes $e$ raised to the power the object denoted by $\tau$.

```
(deffunction exp (?x) := (expt e ?x))                              (5.5)
```

`expt` - The term `(expt` $\tau_1$ $\tau_2$`)` denotes the object denoted by $\tau_1$ raised to the power the object denoted by $\tau_2$.

`fceiling` - The term `(fceiling` $\tau$`)` denotes the smallest integer (as a floating point number) greater than the object denoted by $\tau$.

`ffloor` - The term `(ffloor` $\tau$`)` denotes the largest integer (as a floating point number) less than the object denoted by $\tau$.

`float` - The term `(float` $\tau$`)` denotes the floating point number equal to the object denoted by $\tau$.

`float-digits` - The term `(float-digits` $\tau$`)` denotes the number of digits used in the representation of a floating point number denoted by $\tau$.

**float-precision** - The term (**float-precision** $\tau$) denotes the number of significant digits in the floating point number denoted by $\tau$.

**float-radix** - The term (**float-radix** $\tau$) denotes the radix of the floating point number denoted by $\tau$. The most common values are 2 and 16.

**float-sign** - The term (**float-sign** $\tau_1$ $\tau_2$) denotes a floating-point number with the same sign as the object denoted by $\tau_1$ and the same absolute value as the object denoted by $\tau_2$.

**floor** - The term (**floor** $\tau$) denotes the largest integer less than the object denoted by $\tau$.

**fround** - The term (**fround** $\tau$) is equivalent to (**ffloor** (+ 0.5 $\tau$)).

**ftruncate** - The term (**ftruncate** $\tau$) denotes the largest integer (as a floating point number) less than the object denoted by $\tau$.

**gcd** - The term (**gcd** $\tau_1 \ldots \tau_n$) denotes the greatest common divisor of the objects denoted by $\tau_1$ through $\tau_n$.

**imagpart** - The term (**imagpart** $\tau$) denotes the imaginary part of the object denoted by $\tau$.

**integer-decode-float** - The term (**integer-decode-float** $\tau$) denotes the significand of the object denoted by $\tau$.

**integer-length** - The term (**integer-length** $\tau$) denotes the number of bits required to store the absolute magnitude of the object denoted by $\tau$.

**isqrt** - The term (**isqrt** $\tau$) denotes the integer square root of the object denoted by $\tau$.

**lcm** - The term (**lcm** $\tau_1 \ldots \tau_n$) denotes the least common multiple of the objects denoted by $\tau_1, \ldots, \tau_n$.

**log** - The term (**log** $\tau_1$ $\tau_2$) denotes the logarithm of the object denoted by $\tau_1$ in the base denoted by $\tau_2$.

**logand** - The term (**logand** $\tau_1 \ldots \tau_n$) denotes the bit-wise logical and of the objects denoted by $\tau_1$ through $\tau_n$.

**logandc1** - The term (**logandc1** $\tau_1$ $\tau_2$) is equivalent to (**logand** (**lognot** $\tau_1$) $\tau_2$).

**logandc2** - The term (**logandc2** $\tau_1$ $\tau_2$) is equivalent to (**logand** $\tau_1$ (**lognot** $\tau_2$)).

**logcount** - The term (**logcount** $\tau$) denotes the number of *on* bits in the object denoted by $\tau$. If the denotation of $\tau$ is positive, then the one bits are counted; otherwise, the zero bits in the twos-complement representation are counted.

**logeqv** - The term (**logeqv** $\tau_1 \ldots \tau_n$) denotes the logical-exclusive-or of the objects denoted by $\tau_1, \ldots, \tau_n$.

**logior** - The term (**logior** $\tau_1 \ldots \tau_n$) denotes the bit-wise logical inclusive or of the objects denoted by $\tau_1$ through $\tau_n$. It denotes 0 if there are no arguments.

`lognand` - The term (`lognand` $\tau_1$ $\tau_2$) is equivalent to (`lognot` (`logand` $\tau_1$ $\tau_2$)).

`lognor` - The term (`lognor` $\tau_1$ $\tau_2$) is equivalent to (`not` (`logior` $\tau_1$ $\tau_2$)).

`lognot` - The term (`lognot` $\tau$) denotes the bit-wise logical not of the object denoted by $\tau$.

`logorc1` - The term (`logorc1` $\tau_1$ $\tau_2$) is equivalent to (`logior` (`lognot` $\tau_1$) $\tau_2$).

`logorc2` - The term (`logorc2` $\tau_1$ $\tau_2$) is equivalent to (`logior` $\tau_1$ (`lognot` $\tau_2$)).

`logxor` - The term (`logxor` $\tau_1 \ldots \tau_n$) denotes the bit-wise logical exclusive or of the objects denoted by $\tau_1$ through $\tau_n$. It denotes 0 if there are no arguments.

`max` - The term (`max` $\tau_1 \ldots \tau_k$) denotes the largest object denoted by $\tau_1$ through $\tau_n$.

`min` - The term (`min` $\tau_1 \ldots \tau_k$) denotes the smallest object denoted by $\tau_1$ through $\tau_n$.

`mod` - The term (`mod` $\tau_1$ $\tau_2$) denotes the root of the object denoted by $\tau_1$ modulo the object denoted by $\tau_2$. The result will have the same sign as denoted by $\tau_1$.

`numerator` - The term (`numerator` $\tau$) denotes the numerator of the canonical reduced form of the object denoted by $\tau$.

`phase` - The term (`phase` $\tau$) denotes the angle part of the polar representation of the object denoted by $\tau$ (in radians).

`rationalize` - The term (`rationalize` $\tau$) denotes the rational representation of the object denoted by $\tau$.

`realpart` - The term (`realpart` $\tau$) denotes the real part of the object denoted by $\tau$.

`rem` - The term (`rem <number> <divisor>`) denotes the remainder of the object denoted by `<number>` divided by the object denoted by `<divisor>`. The result has the same sign as the object denoted by `<divisor>`.

`round` - The term (`round` $\tau$) denotes the integer nearest to the object denoted by $\tau$. If the object denoted by $\tau$ is halfway between two integers (for example 3.5), it denotes the nearest integer divisible by 2.

`scale-float` - The term (`scale-float` $\tau_1$ $\tau_2$) denotes a floating-point number that is the representational radix of the object denoted by $\tau_1$ raised to the integer denoted by $\tau_2$.

`signum` - The term (`signum` $\tau$) denotes the sign of the object denoted by $\tau$. This is one of -1, 1, or 0 for rational numbers, and one of -1.0, 1.0, or 0.0 for floating point numbers.

`sin` - The term (`sin` $\tau$) denotes the sine of the object denoted by $\tau$ (in radians).

`sinh` - The term (`sinh` $\tau$) denotes the hyperbolic sine of the object denoted by $\tau$ (in radians).

`sqrt` - The term (`sqrt` $\tau$) denotes the principal square root of the object denoted by $\tau$.

`tan` - The term (`tan` $\tau$) denotes the tangent of the object denoted by $\tau$ (in radians).

`tanh` - The term `(tanh` $\tau$`)` denotes the hyperbolic tangent of the object denoted by $\tau$ (in radians).

`truncate` - The term `(truncate` $\tau$`)` denotes the largest integer less than the object denoted by $\tau$.

## §5.2 Relations on Numbers

`integer` - The sentence `(integer` $\tau$`)` means that the object denoted by $\tau$ is an integer.

`real-number` - The sentence `(real-number` $\tau$`)` means that the object denoted by $\tau$ is a real number.

`complex-number` - The sentence `(complex-number` $\tau$`)` means that the object denoted by $\tau$ is a complex number.

```
    (defrelation number (?x) :=
      (or (real-number ?x) (complex-number ?x)))                    (5.6)

    (defrelation natural (?x) := (and (integer ?x) (>= ?x 0)))      (5.7)

    (defrelation rational-number (?x) :=
      (exists (?y) (and (integer ?y) (integer (* ?x ?y)))))         (5.8)
```

`<` - The sentence `(<` $\tau_1$ $\tau_2$`)` is true if and only if the number denoted by $\tau_1$ is less than the number denoted by $\tau_2$.

```
    (defrelation > (?x ?y) := (< ?y ?x))                            (5.9)

    (defrelation =< (?x ?y) := (or (= ?x ?y) (< ?x ?y)))            (5.10)

    (defrelation >= (?x ?y) := (or (> ?x ?y) (= ?x ?y)))            (5.11)

    (defrelation positive (?x) := (> ?x 0))                         (5.12)

    (defrelation negative (?x) := (< ?x 0))                         (5.13)

    (defrelation zero (?x) := (= ?x 0))                             (5.14)

    (defrelation odd-integer (?x) := (integer (/ (+ ?x 1) 2))       (5.15)

    (defrelation even-integer (?x) := (integer (/ ?x 2))            (5.16)
```

`logbit` - The sentence `(logbit` $\tau_1$ $\tau_2$`)` is true if bit $\tau_2$ of $\tau_1$ is 1.

`logtest` - The sentence `(logtest` $\tau_1$ $\tau_2$`)` is true if the logical *and* of the two's-complement representation of the integers $\tau_1$ and $\tau_2$ is not zero.

# Chapter 6
# Lists

A *list* is a finite sequence of objects. The objects in a list need not be KIF expressions, though they may be. In other words, it is just as acceptable to talk about a list of two people as it is to talk about a list of two symbols.

In KIF, we use the term (`listof` $\tau_1$ ... $\tau_k$) to denote the list of objects denoted by $\tau_1$, ..., $\tau_k$. For example, the following expression denotes the list of an object named `mary`, a list of objects named `tom`, `dick`, and `harry`, and an object named `sally`.

```
(listof mary (listof tom dick harry) sally)
```

The relation `list` is the type predicate for lists. An object is a list if and only if there is a corresponding expression involving the `listof` operator.

```
(defrelation list (?x) :=
  (exists (@l) (= ?x (listof @l)))                              (6.1)
```

The object constant `nil` denotes the empty list. `null` tests whether or not an object is the empty list. The relation constants `single`, `double`, and `triple` allow us to assert the length of lists containing one, two, and three elements, respectively.

```
(defobject nil := (listof))                                     (6.2)

(defrelation null (?l) := (= ?l (listof)))                      (6.3)

(defrelation single (?l) := (exists ?x (= ?l (listof ?x))))     (6.4)

(defrelation double (?l) :=
  (exists (?x ?y) (= ?l (listof ?x ?y))))                       (6.5)

(defrelation triple (?l) :=
  (exists (?x ?y ?z) (= ?l (listof ?x ?y ?z))))                 (6.6)
```

The functions `first`, `rest`, `last`, and `butlast` each take a single list as argument and select individual items or sublists from those lists.

```
(deffunction first (?l) := (if (= (listof ?x @items) ?l) ?x)    (6.7)

(deffunction rest (?l) :=
  (cond ((null ?l) ?l)
        ((= ?l (listof ?x @items)) (listof @items))))           (6.8)

(deffunction last (?l) :=
  (cond ((null ?l) bottom)
        ((null (rest ?l)) (first ?l))
        (true (last (rest ?l)))))                               (6.9)
```

```
(deffunction butlast (?l) :=
  (cond ((null ?l) bottom)
        ((null (rest ?l)) nil)
        (true (cons (first ?l) (butlast (rest ?l))))))               (6.10)
```

The sentence (item $\tau_1$ $\tau_2$) is true if and only if the object denoted by $\tau_2$ is a non-empty list and the object denoted by $\tau_1$ is either the first item of that list or an item in the rest of the list.

```
(defrelation item (?x ?l) :=
  (and (list ?l)
       (not (null ?l))
       (or (= ?x (first ?l)) (item ?x (rest ?l)))))                 (6.11)
```

The sentence (sublist $\tau_1$ $\tau_2$) is true if and only if the object denoted by $\tau_1$ is a final segment of the list denoted by $\tau_2$.

```
(defrelation sublist (?l1 ?l2) :=
  (and (list ?l1)
       (list ?l2)
       (or (= ?l1 ?l2)
           (sublist ?l1 (rest ?l2)))))                              (6.12)
```

The function cons adds the object specified as its first argument to the front of the list specified as its second argument.

```
(deffunction cons (?x ?l) :=
  (if (= ?l (listof @l)) (listof ?x @l)))                           (6.13)
```

The function append adds the items in the list specified as its first argument to the list specified as its second argument. The function revappend is simiar, except that it adds the items in reverse order.

```
(deffunction append (?l1 ?l2) :=
  (if (null ?l1) (if (list ?l2) ?l2)
      (cons (first ?l1) (append (rest ?l1) ?l2))))                  (6.14)

(deffunction revappend (?l1 ?l2) :=
  (if (null ?l1) (if (list ?l2) ?l2)
      (revappend (rest ?l1) (cons (first ?l1) ?l2))))               (6.15)
```

The function reverse produces a list in which the order of items is the reverse of that in the list supplied as its single argument.

```
(deffunction reverse (?l) := (revappend ?l (listof)))               (6.16)
```

The functions adjoin and remove construct lists by adding or removing objects from the lists specified as their arguments.

```
(deffunction adjoin (?x ?l) := (if (item ?x ?l) ?l (cons ?x ?l))) (6.17)
```

```
(deffunction remove (?x ?l) :=
  (cond ((null ?l) nil)
        ((and (= ?x (first ?l)) (listp ?l))
         (remove ?x (rest ?l)))
        ((list ?l) (cons ?x (remove ?x (rest ?l))))))              (6.18)
```

The value of **subst** is the object or list obtained by substituting the object supplied as first argument for all occurrences of the object supplied as second argument in the object or list supplied as third argument.

```
(deffunction subst (?x ?y ?z) :=
  (cond ((= ?y ?z) ?x)
        ((null ?z) nil)
        ((list ?z) (cons (subst ?x ?y (first ?z))
                         (subst ?x ?y (rest ?z))))
        (true ?z)))                                                (6.19)
```

The function constant **length** gives the number of items in a list. **nth** returns the item in the list specified as its first argument in the position specified as its second argument. **nthrest** returns the list specified as its first argument minus the first $n$ items, where $n$ is the number specified as its second argument.

```
(deffunction length (?l) :=
   (cond ((null ?l) 0)
         ((list ?l) (1+ (length (rest ?l))))))                     (6.20)

(deffunction nth (?l ?n) :=
   (cond ((= ?n 1) (first ?l))
         ((positive ?n) (nth (rest ?l) (1- ?n)))))                 (6.21)

(deffunction nthrest (?l ?n) :=
  (cond ((= ?n 0) (if (listp ?l) ?l))
        ((positive ?n)) (nthrest (rest ?l) (1- ?n)))))             (6.22)
```

# Chapter 7
# Sets

In many applications, it is helpful to talk about sets of objects as objects in their own right, e.g. to specify their cardinality, to talk about subset relationships, and so forth.

The formalization of sets of simple objects is a simple matter; but, when we begin to talk about sets of sets, the job becomes difficult due to the threat of paradoxes (like Russell's hypothesized set of all sets that do not contain themselves).

Fortunately, there is no shortage of mathematical theories for our use in KIF – various higher order logics, Zermelo-Fraenkel set theory, von Neuman-Bernays-Gödel set theory, Quine's variants on the previous two approaches, the more recently elaborated proposals by Feferman and Aczel, and so forth. In KIF, we have adopted a version of the von Neumann-Bernays-Gödel set theory.

In our presentation here, we first discuss the basic concepts of this theory – the notions of set and membership. Next, we look at some terminology for describing the properties of sets. We then present the standard axioms of set theory. Finally, we discuss the threat of paradox and indicate how our use of the von Neumann-Bernays-Gödel set theory avoids this problem.

An important word of warning for mathematicians. In KIF, certain words are used nontraditionally. Specifically, the standard notion of *class* is here called a *set*; the standard notion of *set* is replaced by the notion of *bounded* set; and the standard notion of *proper class* is replaced by *unbounded* set.

## §7.1 Basic Concepts

In KIF, a fundamental distinction is drawn between *individuals* and *sets.* A set is a collection of objects. An individual is any object that is not a set.

A distinction is also drawn between objects that are *bounded* and those that are *unbounded.* This distinction is orthogonal to the distinction between individuals and sets. There are bounded individuals and unbounded individuals. There are bounded sets and unbounded sets.

The fundamental relationship among these various types of entities is that of membership. Sets can have members, but individuals cannot. Bounded objects can *be* members of sets, but unbounded objects cannot. (It is this condition that allows us to avoid the traditional paradoxes of set theory.)

In KIF, we use the unary relation constants `individual` and `set`, `bounded` and `unbounded` to make these distinctions; and we use the binary relation constant `member` to talk about membership.

The sentence `(individual` $\tau$`)` is true if and only if the object denoted by $\tau$ is an individual. The sentence `(set` $\tau$`)` is true if and only if the object denoted by $\tau$ is a set. As just described, individuals and sets are exhaustive and mutually disjoint.

```
(or (set ?x) (individual ?x))                         (7.1)

(or (not (set ?x)) (not (individual ?x)))             (7.2)
```

The sentence (bounded $\tau$) is true if and only if the object denoted by $\tau$ is bounded. The sentence (unbounded $\tau$) is true if and only if the object denoted by $\tau$ is unbounded. Boundedness and unboundedness are exhaustive and mutually disjoint.

```
(or (bounded ?x) (unbounded ?x))                                        (7.3)

(or (not (bounded ?x)) (not (unbounded ?x)))                            (7.4)
```

The sentence (member $\tau_1$ $\tau_2$) is true if and only if the object denoted by $\tau_1$ is contained in the set denoted by $\tau_2$. As mentioned above, an object can be a member of another object if and only if the former is bounded and the latter is a set.

```
(=> (member ?x ?s)
    (bounded ?x))                                                       (7.5)

(=> (member ?x ?s)
    (set ?x))                                                           (7.6)
```

An important property shared by all sets is the extensionality property. Two sets are identical if and only if they have the same members.

```
(=> (and (set ?s1) (set ?s2))
    (<=> (forall (?x) (<=> (member ?x ?s1) (member ?x ?s2)))
         (= ?s1 ?s2)))                                                  (7.7)
```

## §7.2 Sets

To allow us to name specific sets, KIF provides the operators setof and setofall.

The term (setof $\tau_1$ ... $\tau_k$) denotes the set consisting of the objects denoted by $\tau_1$, ..., $\tau_k$ that are bounded.

```
(=> (item ?x (listof @items))
    (bounded ?x)
    (member ?x (setof @items)))                                         (7.8)

(=> (member ?x (setof @items))
    (item ?x (listof @items)))                                          (7.9)
```

Note that the cardinality of the set denoted by (setof $\tau_1$ ... $\tau_k$) can be less than $k$. By definition, an object can appear in a set only once. Consequently, if $\tau_i$ and $\tau_j$ (for different $i$ and $j$) denote the same object, the resulting set must contain fewer than $k$ members.

The operator setofall allows us to define sets in terms of their properties. The term (setofall $\tau$ $\phi$) denotes the set of all bounded objects denoted by $\tau$ for any assignment of the free variables in $\tau$ that satisfies $\phi$.

(<=> (member $\tau$ (setofall $\nu$ $\phi$))
  (and (bounded $\tau$) $\phi_{\nu/\tau}$))  (7.10)

Note that the first argument to setofall must be a term, not a list of variables as with forall and exists. The term can be a single variable, a functional expression, or even a quantified term. If the term contains no free variables, then the set consists of either zero members or one member, depending on the truth value of the embedded sentence.

The `empty` relation is true of the empty set but false of all other objects.

```
(defrelation empty (?x) := (= ?x (setof)))                    (7.11)
```

In KIF, the functions `union`, `intersection`, `difference`, and `complement` are defined as follows.

```
(deffunction union (@sets) :=
  (if (forall (?s) (=> (item ?s (listof @sets)) (set ?s)))
      (setofall ?x (exists (?s) (and (item ?s (listof @sets))
                                     (member ?x ?s))))))      (7.12)

(deffunction intersection (@sets) :=
  (if (forall (?s) (=> (item ?s (listof @sets)) (set ?s)))
      (setofall ?x (forall (?s) (=> (item ?s (listof @sets))
                                    (member ?x ?s))))))       (7.13)

(deffunction difference (?set @sets) :=
  (if (and (set ?set)
           (forall (?s) (=> (item ?s (listof @sets)) (set ?s))))
      (setofall ?x
                (and (member ?x ?set)
                     (forall (?s) (=> (item ?s (listof @sets))
                                      (not (member ?x ?s)))))))) (7.14)

(deffunction complement (?s) :=
  (if (set ?s)
      (setofall ?x (not (member ?x ?s)))))                    (7.15)
```

The functions `generalized-union` and `generalized-intersection` allow us to talk about the union and intersection of the sets in a set of sets.

```
(deffunction generalized-union (?set) :=
  (if (and (set ?set)
           (forall (?s) (=> (member ?s ?set) (set ?s))))
      (setofall ?x (exists (?s) (and (member ?s ?set)
                                     (member ?x ?s))))))      (7.16)

(deffunction generalized-intersection (?set) :=
  (if (and (set ?set)
           (forall (?s) (=> (member ?s ?set) (set ?s))))
      (setofall ?x (exists (?s) (=> (member ?s ?set)
                                    (member ?x ?s))))))       (7.17)
```

The sentence `(subset` $\tau_1$ $\tau_2$`)` is true if and only if $\tau_1$ and $\tau_2$ are sets and the objects in the set denoted by $\tau_1$ are contained in the set denoted by $\tau_2$.

```
(defrelation subset (?s1 ?s2) :=
  (and (set ?s1) (set ?s2)
       (forall ?x (=> (member ?x ?s1) (member ?x ?s2)))))           (7.18)
```

The sentence (**proper-subset** $\tau_1$ $\tau_2$) is true if the set denoted by $\tau_1$ is a subset of the set denoted by $\tau_2$ but not vice-versa.

```
(defrelation proper-subset (?s1 ?s2) :=
  (and (subset ?s1 ?s2)
       (not (subset ?s2 ?s1))))                                     (7.19)
```

Two sets are disjoint if and only if there is no object that is a member of both sets. Sets are pairwise-disjoint if and only if every set is disjoint from every other set. Sets are mutually-disjoint if and only if there is no object that is a member of all of the sets. Note that mutually-disjoint sets need not be pairwise disjoint; in fact, every pair of sets might be overlapping. For example, the sets $\{a, b\}$ and $\{b, c\}$ and $\{a, c\}$ are mutually disjoint but not pairwise disjoint.

```
(defrelation disjoint (?s1 ?s2) :=
  (empty (intersection ?s1 ?s2)))                                   (7.20)

(defrelation pairwise-disjoint (@sets) :=
  (forall (?s1 ?s2) (=> (item ?s1 (listof @sets))
                        (item ?s2 (listof @sets))
                        (or (= ?s1 ?s2) (disjoint ?s1 ?s2)))))      (7.21)

(defrelation mutually-disjoint (@sets) :=
  (= (intersection @sets) (set)))                                   (7.22)

(defrelation set-partition (?s @sets) :=
  (and (= ?s (union @sets))
       (pairwise-disjoint @sets)))                                  (7.23)

(defrelation set-cover (?s @set) :=
  (subset ?s (union @sets)))
                                                                    (7.24)
```

We close this section with two axioms that allow us to conclude that sets of various sorts do, in fact, exist. The first is the *axiom of regularity* – every non-empty set has an element with which it has no members in common.

```
(forall (?s)
  (=> (not (empty ?s))
      (exists (?u) (and (member ?u ?s) (disjoint ?u ?s)))))         (7.25)
```

This axiom is not absolutely essential for set theory. However, it makes many proofs a lot easier, and so it is commonly included among the axioms of set theory.

The second axiom is the *axiom of choice.* It asserts that there is a set that associates every bounded set with a distinguished element of that set. In effect, it *chooses* an element from every bounded set.

```
(exists (?s)
  (and (set ?s)
       (forall (?x) (=> (member ?x ?s) (double ?x)))
       (forall (?x ?y ?z) (=> (and (member (listof ?x ?y) ?s)
                                   (member (listof ?x ?z) ?s))
                              (= ?y ?z)))
       (forall (?u)
         (=> (and (bounded ?u) (not (empty ?u)))
             (exists (?v) (and (member ?v ?u)
                               (member (listof ?u ?v) ?s)))))))   (7.26)
```

Again, this axiom is not essential. In some versions of set theory, the axiom of choice is omitted. However, it is a highly desirable property and is included here for that reason.

## §7.3 Boundedness

As mentioned earlier, the key difference between bounded and unbounded objects is that the former can be members of other sets while the latter cannot. This fact establishes a necessary and sufficient test for boundedness - an object is bounded just in case it is a member of a set. However, this is not very helpful, since we often need to determine whether or not an object is bounded based on other properties, not the sets of which it is a member. In this section, we look at some axioms that help us make this determination for sets.

First of all, we have the *finite set axiom.* Any finite set of bounded sets is itself a bounded set.

```
(bounded (setof @l))                                                    (7.27)
```

The *subset axiom* assures that the set of all of subsets of a bounded set is also a bounded set.

```
(=> (bounded ?v) (bounded (setofall ?u (subset ?u ?v))))               (7.28)
```

The *union axiom* tells us that the generalized union of any bounded set of bounded sets is also a bounded set. Since every finite set is bounded, this allows us to conclude, as a special case, that the union of any finite number of bounded sets is a bounded set.

```
(=> (and (bounded ?u) (forall (?x) (=> (member ?x ?u) (bounded ?x))))
    (bounded (generalized-union ?u)))                                  (7.29)
```

The *intersection axiom* tells us that the intersection of a bounded set and any other set is a bounded set. So long as one of the sets defining the intersection is bounded, the resulting set is bounded.

```
(=> (and (bounded ?u) (set ?s))
    (bounded (intersection ?u ?s)))                                    (7.30)
```

Finally, we have the *axiom of infinity.* There is a bounded set containing a set, a set that properly contains that set, a third set that properly contains the second set, and so forth. In short, there is at least one bounded set of infinite cardinality.

```
(exists (?u)
  (and (bounded ?u)
       (not (empty ?u))
       (forall (?x)
         (=> (member ?x ?u)
             (exists (?y) (and (member ?y ?u)
                               (proper-subset ?x ?y)))))))        (7.31)
```

## §7.4 Paradoxes

The presence of sets in our universe of discourse does not in itself lead to paradoxes. The paradoxes appear only when we try to define set primitives that are too powerful. We have defined the sentence (member $\tau$ $\sigma$) to be true in exactly those cases when the object denoted by $\tau$ is a member of the set denoted by $\sigma$, and we might consider defining the term (setofall $\tau$ $\phi$) to mean simply the set of all objects denoted by $\tau$ for any assignment of the free variables of $\tau$ that satisfies $\phi$. Unfortunately, these two definitions quickly lead to paradoxes.

Let $\phi_{\nu/\tau}$ be the result of substituting term $\tau$ for all free occurrences of $\nu$ in sentence $\phi$. Provided that $\tau$ is a term not containing any free variables captured in $\phi_{\nu\tau}$, then the following schema follows from our informal definition. This schema is called the *principle of unrestricted set abstraction.*

(<=> (member $\tau$ (setofall $\nu$ $\phi$)) $\phi_{\nu/\tau}$)

Now, let us substitute the variable ?x for $\nu$, the sentence (not (member ?x ?x)) for $\phi$, and the term (setofall ?x (not (member ?x ?x))) for $\tau$. The resulting instance of the principle of unrestricted set abstraction follows.

```
(<=> (member (setofall ?x (not (member ?x ?x)))
             (setofall ?x (not (member ?x ?x))))
     (not (member (setofall ?x (not (member ?x ?x)))
                  (setofall ?x (not (member ?x ?x))))))
```

This sentence has the form (<=> $\phi$ (not $\phi$)), which cannot be true in any interpretation. This is Russell's paradox, only one of a family of familiar absurdities following from the principle of unrestricted set abstraction.

It is crucial that the paradoxes of set theory be avoided. One of the goals in the design of KIF is that it have a clearly specified model-theoretic semantics in terms of which the concepts of entailment, equivalence, consistency, soundness and completeness can be defined. If the paradoxes are allowed to persist in principle, even if they are easy to avoid in practice, the consequence would be that no KIF theory would be true in any model. Definitions couched in terms of models would be trivialized, becoming useless. All sentences would be entailed by any theory, any two theories would be equivalent, no theory would be consistent, every possible inference rule would be sound, and so on.

In the von-Neuman-Gödel-Bernays version of set theory, these paradoxes are avoided by replacing the principle of unrestricted set abstraction with the ***principle of restricted set abstraction*** given above.

(<=> (member $\tau$ (setofall $\nu$ $\phi$))
    (and (bounded $\tau$) $\phi_{\nu/\tau}$))

With this principle, there are two reasons why something may be excluded from a set (setofall $\nu$ $\phi$). It may fail to be a member because it does not satisfy the defining condition $\phi$, or it may be excluded because it is an unbounded object. Conditioning the membership of objects in this set on their boundedness effectively eliminates the paradoxes.

# Chapter 8
# Functions and Relations

In KIF, we can describe specific functions and relations by naming them with function constants and relation constants and then writing sentences in which those names occur in functional or relational position. For most purposes, this is adequate; but in some cases it is also useful to describe functions and relations more generally – to name their properties (such as associativity and transitivity) and to write axioms relating these properties (possibly quantifying over the functions and relations possessing these properties). In order to do this, we need to treat functions and relations as objects in our universe of discourse.

By definition, functions and relations are sets of lists of objects from our universe of discourse. The immediately preceding chapters offer a vocabulary for describing lists and sets in general. However, functions and relations have enough special properties to warrant additional vocabulary.

In what follows, we begin by presenting the KIF vocabulary for abstraction and application of functions and relations. We then talk about the use of functions and relation constants in argument position of terms. Finally, we present some supporting vocabulary.

Note that the introduction of functions and relations into our universe of discourse comes with the threat of paradox, as with sets in general. In KIF, we sidestep such paradoxes by defining the sets comprising our functions and relations in terms of the set concepts introduced in the preceding chapter.

## §8.1 Basic Vocabulary

As described in chapter 3, a relation is an arbitrary set of lists. A collection of objects satisfies a relation if and only if the list of those objects is a member of this set.

```
(defrelation relation (?r) :=
  (and (set ?r)
       (forall (?x) (=> (member ?x ?r) (list ?x)))))              (8.1)
```

Since KIF allows for n-ary relations, the lists in the set need not be of the same length. For example, the $<$ relation is defined on 2-lists, 3-lists, 4-lists, and so forth.

A function is a set of lists in which the items in every list except for the last determine the last item, i.e. there cannot be two lists that agree on all but the last item and disagree on the last item.

```
(defrelation function (?f) :=
  (and (relation ?f)
       (forall (?l ?m)
         (=> (member ?l ?f)
             (member ?m ?f)
             (= (butlast ?l) (butlast ?m))
             (= (last ?l) (last ?m))))))                          (8.2)
```

As with relations in general, the lists of a function need not be of the same length, to allow for functions of variable arity. For example, associative functions like $+$ and $*$ functions can be applied to arbitrary numbers of arguments.

An important difference between our treatment of functions and the traditional treatment is that functions need not contain lists for every possible combination of arguments. If a function is undefined for a particular combination of objects (i.e. if its value is $\perp$), then we omit that list from the set. Thus, even though our universe of discourse is infinite, it is possible for a function to have a finite number of lists.

## §8.2 Function and Relation Constants

Since function constants and relation constants denote functions and relations and since functions and relations are objects in our universe of discourse, it is natural to allow function and relation constants to appear as as arguments in terms and sentences.

Unfortunately, in order to avoid paradoxes, it is sometimes essential for there to be a difference between the interpretation of a function or relation constant and its semantic value. We can sidestep these potential difficulties by writing axioms that define function and relation constants, used in argument position, in terms of the `setof` operator.

As described in chapter 4, the semantic value of a function constant $\pi$ is the set of lists of objects corresponding to the function denoted by $\pi$. The following axiom schema expresses this property.

(= $\pi$ (setofall (listof $\nu_1$ ... $\nu_k$ $\nu$) (= ($\pi$ $\nu_1$ ... $\nu_k$) $\nu$)) (8.3)

Similarly, the semantic value of a relation constant $\rho$ is the set of lists of objects that satisfy the relation denoted by $\rho$. Again, we have an axiom schema corresponding to this property.

(= $\rho$ (setofall (listof $\nu_1$ ... $\nu_k$) ($\rho$ $\nu_1$ ... $\nu_k$))) (8.4)

The use of function and relation constants in argument position weakens the distinction between object constants on the one hand and function and relation constants on the other.

The distinction between function and relation constants can also be weakened, since functions are a special class of relations. Any position that requires a relation constant can also be filled by a function constant. When this happens, the function denoted by the function constant is treated as a relation (which it is). For instance, in the following sentence, the first occurrence of + plays the role of a relation constant, while in the second occurrence, it plays the role of a function constant. (In both cases, + denotes the same entity.)

```
(and (+ 2 3 5)
     (= (+ 2 3 5) 10))
```

In KIF, all function constants are treated as relation constants, and all relation constants (and hence all function constants) are treated as object constants. An object constant is still prohibited from occurring as the first item of a term or a sentence, and a relation constant that is not a function constant cannot occupy the first position in a term.

The convenience afforded by the ability to use function and relation constants as arguments and to use function constants in relational position often causes concern over grammatical ambiguity. The expression `(+ 5 2 3)` is both a term and a sentence. Fortunately, this ambiguity is always resolved when such expressions occur within well-formed databases. Any expression that occurs at top level cannot be a term. An expression embedded in a non-operator expression must be a term. An expression embedded in an operator expression can be either a term or a sentence, but in either case the type of the expression is known from the operator's syntax.

## §8.3 Concretion

If $\tau$ denotes a relation, then the sentence (holds $\tau$ $\tau_1$ ... $\tau_k$) is true if and only if the list of objects denoted by $\tau_1$,...,$\tau_k$ is a member of that relation.

```
(defrelation holds (?r @args) :=
  (and (relation ?r) (member (listof @args) ?r)))                (8.5)
```

If $\tau$ denotes a function with a value for the objects denoted by $\tau_1$,..., $\tau_k$, then the term (value $\tau$ $\tau_1$ ... $\tau_k$) denotes the value of applying that function to the objects denoted by $\tau_1$,...,$\tau_k$. Otherwise, the value is undefined.

```
(deffunction value (?f @args) :=
  (if (and (function ?f)
           (member ?l ?f)
           (= (butlast ?l) (listof @args)))
      (last ?l)))                                                (8.6)

(deffunction apply (?f ?l) :=
  (if (and (function ?f) (= ?l (listof @args)))
      (value ?f @args)))                                         (8.7)

(deffunction map (?f ?l) :=
  (if (null ?l) (list)
      (cons (value ?f (first ?l)) (map ?f (rest ?l)))))          (8.8)
```

## §8.4 Abstraction

As described in chapter 4, the semantic value of the term (lambda ($\nu_1$ ... $\nu_k$ [$\omega$]) $\tau$) is the set of lists associated with the function that maps every list of objects "matching" the variable list to the value of $\tau$ when the variables in the variable list are assigned to the objects in the list. We can capture this meaning with the following axiom schema.

(= (lambda ($\nu_1$ ... $\nu_k$ [$\omega$]) $\tau$)
  (setofall (listof $\nu_1$ ... $\nu_k$ [$\omega$] $\nu$) (= $\nu$ $\tau$)))   (8.9)

The semantic value of the term (kappa ($\nu_1$ ... $\nu_k$ [$\omega$]) $\phi$) is the set of lists associated with the relation that holds of every list of objects "matching" the variable list for which the sentence $\phi$ is satisfied. We can capture this meaning with the following axiom schema.

```
(= (kappa (ν₁ ...  νₖ [ω]) φ)
   (setofall (listof ν₁ ...  νₖ [ω]) φ)))                    (8.10)
```

## §8.5 Additional Concepts

The universe of a relation is the set of all objects occurring in some list contained in that relation.

```
(deffunction universe (?r) :=
  (if (relation ?r)
      (setofall ?x (exists (?l) (and (member ?l ?r)
                                      (item ?x ?l))))))      (8.11)
```

A unary relation is a non-empty relation in which all lists have exactly one item.

```
(defrelation unary-relation (?r) :=
  (and (not (empty ?r))
       (forall (?l) (=> (member ?l ?r) (single ?l)))))       (8.12)
```

A binary relation is a non-empty relation in which all lists have exactly two items. The inverse of a binary relation is a binary relation with all tuples reversed. The composition of a binary relation $r_1$ and a binary relation $r_2$ is a binary relation in which an object $x$ is related to an object $z$ if and only if there is an object $y$ such that $x$ is related to $y$ by $r_1$ and $y$ is related to $z$ by $r_2$.

```
(defrelation binary-relation (?r) :=
  (and (not (empty ?r))
       (forall (?l) (=> (member ?l ?r) (double ?l)))))       (8.13)

(deffunction inverse (?r) :=
  (if (binary-relation ?r)
      (setofall (listof ?y ?x) (holds ?r ?x ?y))))           (8.14)

(deffunction composition (?r1 ?r2) :=
  (if (and (binary-relation ?r1)
           (binary-relation ?r2)
      (setofall (listof ?x ?z)
        (exists (?y)
          (and (holds ?r1 ?x ?y)
               (holds ?r2 ?y ?z))))))                        (8.15)

(defrelation one-one (?r) :=
  (and (binary-relation ?r)
       (function ?r)
       (function (inverse ?r))))                             (8.16)
```

```
(defrelation many-one (?r) :=
  (and (binary-relation ?r)
       (function ?r)))                                    (8.17)
```

```
(defrelation one-many (?r) :=
  (and (binary-relation ?r)
       (function (inverse ?r))))                          (8.18)
```

```
(defrelation many-many (?r) :=
  (and (binary-relation ?r)
       (not (function ?r))
       (not (function (inverse ?r)))))                    (8.19)
```

A unary function is a function with a single argument and a single value. Hence, it is also a binary relation.

```
(defrelation unary-function (?f) :=
  (and (function ?f)
       (binary-relation ?f)))                             (8.20)
```

A binary function is a function with two arguments and one value. Hence, it is a relation with three arguments.

```
(defrelation binary-function (?f) :=
  (and (function ?f)
       (not (empty ?f))
       (forall (?l) (=> (member ?l ?f) (triple ?l)))))    (8.21)
```

# Chapter 9
# Metaknowledge

## §9.1 Naming Expressions

In formalizing knowledge about knowledge, we use a conceptualization in which expressions are treated as objects in the universe of discourse and in which there are functions and relations appropriate to these objects. In our conceptualization, we treat atoms as primitive objects (i.e. having no subparts). We conceptualize complex expressions (i.e. non-atoms) as lists of subexpressions (either atoms or other complex expressions). In particular, every complex expression is viewed as a list of its immediate subexpressions.

For example, we conceptualize the sentence `(not (p (+ a b c) d))` as a list consisting of the operator `not` and the sentence `(p (+ a b c) d)`. This sentence is treated as a list consisting of the relation constant `p` and the terms `(+ a b c)` and `d`. The first of these terms is a list consisting of the function constant `+` and the object constants `a`, `b`, and `c`.

For Lisp programmers, this conceptualization is relatively obvious, but it departs from the usual conceptualization of formal languages taken in the mathematical theory of logic. It has the disadvantage that we cannot describe certain details of syntax such as parenthesization and spacing (unless we augment the conceptualization to include string representations of expressions as well). However, it is far more convenient for expressing properties of knowledge and inference than string-based conceptualizations.

In order to assert properties of expressions in the language, we need a way of referring to those expressions. There are two ways of doing this in KIF.

One way is to use the `quote` operator in front of an expression. From the section on semantics, we know that a quotation denotes the expression embedded within the term. Therefore, to refer to the symbol `john`, we use the term `'john` or, equivalently, `(quote john)`. To refer to the expression `(p a b)`, we use the term `'(p a b)` or, equivalently, `(quote (p a b))`.

With a way of referring to expressions, we can assert their properties. For example, the following sentence ascribes to the individual named `john` the belief that the moon is made of a particular kind of blue cheese.

```
(believes john '(material moon stilton))
```

Note that, by nesting quotes within quotes, we can talk about quoted expressions. In fact, we can write towers of sentences of arbitrary heights, in which the sentences at each level talk about the sentences at the lower levels.

Since expressions are first-order objects, we can quantify over them, thereby asserting properties of whole classes of sentences. For example, we could say that Mary believes everything that John believes. This fact together with the preceding fact allows us to conclude that Mary also believes the moon to be made of blue cheese.

```
(=> (believes john ?p) (believes mary ?p))
```

The second way of referring to expressions is KIF is to use the `listof` function. For example, we can denote a complex expression like `(p a b)` by a term of the form `(listof 'p 'a 'b)`, as well as `'(p a b)`.

The advantage of the `listof` representation over the `quote` representation is that it allows us to quantify over parts of expressions. For example, let us say that Lisa is more skeptical than Mary. She agrees with John, but only on the composition of things. The first sentence below asserts this fact without specifically mentioning `moon` or `stilton`. Thus, if we were to later discover that John thought the sun to be made of chili peppers, then Lisa would be constrained to believe this as well.

```
(=> (believes john (listof 'material ?x ?y))
    (believes lisa (listof 'material ?x ?y)))
```

While the use of `listof` allows us to describe the structure of expressions in arbitrary detail, it is somewhat awkward. For example, the term `(listof 'material ?x ?y)` is somewhat awkward. Fortunately, we can eliminate this difficulty using backquote and comma. Rather than using the `listof` function constant as described above, we write the expression preceded by the backquote character ` and add a comma character , in front of any subexpression that is not to be taken literally. For example, we would rewrite the preceding sentence as follows.

```
(=> (believes john `(material ,?x ,?y))
    (believes lisa `(material ,?x ,?y)))
```

This approach is particularly nice in that it parallels the treatment of quoting and unquoting in Common Lisp. However, a warning is in order. All Common Lisps translate quoted expressions into lists with `quote` as the first element, e.g. `'(f a)` translates into `(quote (f a))`. However, not all Common Lisps are consistent in the handling of backquote. Some Lisps translate backquoted expressions into internal forms involving `listof`, e.g. `` `(f ,?x) `` translates into `(listof 'f ?x)`. Some use `cons`, e.g. `(cons 'f (cons ?x nil))`. Some use neither or a mixture. This does not prohibit our using the approach in KIF, but it means that we cannot rely on all Lisp readers to produce the internal form we want.

## §9.2 Formalizing Syntax

In order to facilitate the encoding of knowledge about KIF, the language includes type relations for the various syntactic categories defined in chapter 2.

For every individual variable $\nu$, there is an axiom asserting that it is indeed an individual variable. Each such axiom is a defining axiom for the `indvar` relation.

`(indvar (quote` $\nu$`))` (9.1)

For every sequence variable $\omega$, there is an axiom asserting that it is a sequence variable. Each such axiom is a defining axiom for the `seqvar` relation.

`(seqvar (quote` $\omega$`))` (9.2)

```
(defrelation termop (?x) :=
  (member ?x (setof 'quote 'if 'cond 'the 'setof
                    'kappa 'lambda)))                                (9.3)
```

```
(defrelation sentop (?x) :=
  (member ?x (setof 'not 'and 'or '=> '<= '<=> 'forall 'exists)))  (9.4)
```

```
(defrelation ruleop (?x) := (member ?x (setof  '=>> '<<=)))         (9.5)
```

```
(defrelation defop (?x) :=
  (member ?x (setof 'defobject 'deffunction 'defrelation ':=
                    ':=> ':axiom ':conservative-axiom)))            (9.6)
```

For every constant $\sigma$, there is an axiom asserting that it is a constant. Each such axiom is a defining axiom for the **constant** relation. The category of each constant is determined from its definition and/or the uses of the constant in a knowledge base.

```
(defrelation constant (constant (quote $\sigma$)))                   (9.7)
```

From these basic vocabulary items, we define variables, operators, words, and expressions.

```
(defrelation variable (?x) := (or (indvar ?x) (seqvar ?x)))         (9.8)
```

```
(defrelation operator (?x) :=
  (or (termop ?x) (sentop ?x) (ruleop ?x) (defop ?x)))              (9.9)
```

```
(defrelation word (?x) :=
  (or (variable ?x) (operator ?x) (constant ?x)))                   (9.10)
```

```
(defrelation expression (?x) :=
  (or (word ?x)
      (and (list ?x)
           (forall (?y) (=> (item ?y ?x) (expression ?y))))))       (9.11)
```

The sentence (**term** $\tau$) is true if and only if the object denoted by $\tau$ is a term, i.e. it is either a constant, a variable, functional term, a list term, a set term, a quoted term, a logical term, or a quantified term.

```
(defrelation term (?x) :=
  (or (indvar ?x) (objconst ?x) (funconst ?x) (relconst ?x)
      (funterm ?x) (listterm ?x) (setterm ?x) (quoterm ?x)
      (logterm ?x) (quanterm ?x)))                                  (9.12)
```

```
(defrelation funterm (?x) :=
  (exists (?f ?tlist)
    (and (funconst ?f)
         (list ?tlist)
         (=> (item ?t  ?tlist) (term ?t))
```

```
        (= ?x (cons ?f ?tlist)))))                                      (9.13)

(defrelation listterm (?x) :=
  (exists ?tlist
    (and (list ?tlist)
         (=> (item ?t  ?tlist) (term ?t))
         (= ?x (cons 'listof ?tlist)))))                                (9.14)

(defrelation setterm (?x) :=
  (exists ?tlist
    (and (list ?tlist)
         (=> (item ?t  ?tlist) (term ?t))
         (= ?x (cons 'setof ?tlist)))))                                 (9.15)

(defrelation (?x) :=
  (exists (?e)
    (and (expression ?e)
         (= ?x `(quote ,?e)))))                                         (9.16)

(defrelation logterm (?x) :=
  (or (exists (?p1 ?t1)
        (and (sentence ?p1) (term ?t1) (= ?x `(if ,?p1 ,?t1))))
      (exists (?p1 ?t1 ?t2)
        (and (sentence ?p1)
             (term ?t1)
             (term ?t2)
             (= ?x `(if ,?p1 ,?t1 ,?t2))))
      (exists ?clist
        (and (list ?clist)
             (=> (item ?c ?clist)
                 (exists (?p ?t)
                   (and (sentence ?p) (term ?t)
                        (= ?c (listof ?p ?t)))))
             (= ?x (cons 'cond ?clist))))))                             (9.17)

(defrelation quanterm (?x) :=
  (or (exists (?t ?p)
        (and (term ?t) (sentence ?p)
             (= ?x (listof 'the ?t ?p))))
      (exists (?t ?p)
        (and (term ?t) (sentence ?p)
             (= ?x (listof 'setof ?t ?p))))
      (exists (?vlist ?p)
        (and (list ?vlist) (sentence ?p)
             (>= (length ?vlist) 1)
```

```
              (=> (item ?v ?vlistp) (indvar ?v))
              (= ?x (listof 'kappa ?vlistp ?p))))
        (exists (?vlist ?sv ?p)
          (and (list ?vlist) (seqvar ?sv) (sentence ?p)
              (=> (item ?v ?vlist) (indvar ?v))
              (= ?x (listof 'kappa (append ?vlist (listof ?sv)) ?p))))
        (exists (?vlist ?t)
          (and (list ?vlist) (term ?t)
              (>= (length ?vlist) 1)
              (=> (item ?v ?vlistp) (indvar ?v))
              (= ?x (listof 'lambda ?vlistp ?t))))
        (exists (?vlist ?sv ?t)
          (and (list ?vlist) (seqvar ?sv) (sentence ?t)
              (=> (item ?v ?vlist) (indvar ?v))
              (= ?x (listof 'lambda
                            (append ?vlist (listof ?sv))
                            ?t))))))                                  (9.18)
```

The sentence (sentence $\tau$) is true if and only if the object denoted by $\tau$ is a sentence, i.e. it is either a logical constant, a relational sentence, a logical sentence, or a quantified sentence.

```
(defrelation sentence (?x) :=
  (or (logconst ?x) (relsent ?x) (equation ?x)
      (inequality ?x) (logsent ?x) (quantsent ?x)))                   (9.19)

(defrelation relsent (?x) :=
  (exists (?r ?tlist)
    (and (or (relconst ?r) (funconst ?r)) (list ?tlist)
         (>= (length ?tlist) 1)
         (=> (item ?t ?tlist) (term ?t))
         (= ?x (cons ?r ?tlist)))))                                   (9.20)

(defrelation equation (?x) :=
  (exists (?t1 ?t2)
    (and (term ?t1) (term ?t2)
         (= ?x `(= ,?t1 ,?t2)))))                                     (9.21)

(defrelation inequality (?x) :=
  (exists (?t1 ?t2)
    (and (term ?t1) (term ?t2)
         (= ?x `(/= ,?t1 ,?t2)))))                                    (9.22)

(defrelation logsent (?x) :=
  (or (negation ?x) (conjunction ?x) (disjunction ?x)
      (implication ?x) (reverse-implication ?x)
```

```
     (equivalence ?x)))                                                   (9.23)

(defrelation negation (?x) :=
  (exists (?p)
    (and (sentence ?p)
         (= ?x `(not ,?p)))))                                             (9.24)

(defrelation conjunction (?x) :=
  (exists ?plist
    (and (list ?plist)
         (>= (length ?plist) 1)
         (=> (item ?p ?plist) (sentence ?p))
         (= ?x (cons 'and ?plist)))))                                     (9.25)

(defrelation disjunction (?x) :=
  (exists ?plist
    (and (list ?plist)
         (>= (length ?plist) 1)
         (=> (item ?p ?plist) (sentence ?p))
         (= ?x (cons 'or ?plist)))))                                      (9.26)

(defrelation implication (?x) :=
  (exists (?plist)
    (and (list ?plist)
         (>= (length ?plist) 2)
         (=> (item ?p ?plist) (sentence ?p))
         (= ?x (cons '=> ?plist)))))                                      (9.27)

(defrelation reverse-implication (?x) :=
  (exists (?plist)
    (and (list ?plist)
         (>= (length ?plist) 2)
         (=> (item ?p ?plist) (sentence ?p))
         (= ?x (cons '<= ?plist)))))                                      (9.28)

(defrelation equivalence (?x) :=
  (exists (?p1 ?p2)
    (and (sentence ?p1)
         (sentence ?p2)
         (= ?x `(<=> ,?p1 ,?p2)))))                                       (9.29)

(defrelation quantsent (?x) :=
  (or (exists (?v ?p)
        (and (indvar ?v) (sentence ?p)
             (or (= ?x (listof 'forall ?v ?p))
              (= ?x (listof 'exists ?v ?p)))))
    (exists (?vlist ?p)
      (and (list ?vlist) (sentence ?p)
           (>= (length ?vlist) 1)
           (=> (item ?v ?vlist) (indvar ?v))
           (or (= ?x (listof 'forall ?vlist ?p))
               (= ?x (listof 'exists ?vlist ?p)))))))          (9.30)
```

## §9.3 Changing Levels of Denotation

The vocabulary introduced in the preceding subsection allows us to encode properties of expressions in and of themselves. In this section, we add some vocabulary that allows us to change levels of denotation, i.e. to relate expressions about expressions with the expressions they denote.

The term `(denotation` $\tau$`)` denotes the object denoted by the object denoted by $\tau$. A quotation denotes the quoted expression; the denotation of any other object is $\perp$.

The term `(name` $\tau$`)` denotes the standard name for the object denoted by the term $\tau$. The standard name for an expression $\tau$ is `(quote` $\tau$`)`; the standard name for a non-expression is at the discretion of the user. (Note that there are only a countable number of terms in KIF, but there can be models with uncountable cardinality; consequently, it is not always possible for every object in the universe of discourse to have a unique name.)

The final level-crossing vocabulary item is the relation constant `true`. For example, we can say that a sentence of the form `(=> (p ?x) (q ?x))` is true by writing the following sentence.

```
(true '(=> (p ?x) (q ?x)))
```

This may seem of limited utility, since we can just write the sentence denoted by the argument as a sentence in its own right. The advantage of the metanotation becomes clear when we need to quantify over sentences, as in the encoding of axiom schemas. For example, we can say that every sentence of the form `(=>` $\phi$ $\phi$`)` is true with the following sentence.

```
(=> (sentence ?p) (true `(=> ,?p ,?p)))
```

Semantically, we would like to say that a sentence of the form `(true '`$\phi$`)` is true if and only if the sentence $\phi$ is true. In other words, for any interpretation and variable assignment, the truth value $t_{iv}$(`(true '`$\phi$`)`) is the same as the truth value $t_{iv}(\phi)$. In other words, for every truth function $t_{iv}$, `true` is our language's name for $t_{iv}$.

Unfortunately, this causes serious problems. Equating a truth function with the meaning it ascribes to `true` quickly leads to paradoxes. The English sentence "This sentence is false." illustrates the paradox. We can write this sentence in KIF as shown below. The sentence, in effect, asserts its own negation.

```
(true (subst (name `(subst (name x) `x `(true ,x)))
             `x
             `(not (true (subst (name x) `x `(not (true ,x)))))))
```

For any truth function $t_{iv}$ that maps `true` to itself, we get a contradiction. If $t_{iv}$ of this sentence is *true*, then by the rules for assignment of the logical operators contained in the sentence, we see that $t_{iv}$ must make the sentence *false*. If $t_{iv}$ assigns the value *false*, then, again by the rules for assignment of the logical operators, we see that it must assign it the value *true*. In either case, we get a contradication.

Fortunately, we can circumvent such paradoxes by slightly modifying the definition of `true`. The treatment here follows that of Kripke, Gilmore, and Perlis. Although the approach is a little complicated, it is nice in that the intuitive interpretation of `true` is in all important cases exactly what we would guess, yet paradoxes are completely avoided.

`(<=> (true` $\phi$`)` $\phi^*$`)` (9.31)

Given a sentence $\phi$, we define $\phi^*$ to be the sentence obtained from $\phi$ as follows. If the sentence is logical, then all occurrences of `not` are pushed inside other operators. If the sentence is `(not (true` $\tau$`))`, the $\phi^*$ is `(true (listof 'not` $\tau$`))`.

Since the truth of a sentence `(true` $\phi$`)` is determined by the truth value of $\phi^*$, not $\phi$, the potential for paradoxes is eliminated. For most sentences, $\phi^*$ and $\phi$ are the same. For apparently paradoxical sentences, the two differ and so no contradiction arises. (See Perlis for the description of a model for databases containing this axiom schema.)

# Chapter 10
# Nonmonotonicity

Many knowlege representation and reasoning systems are capable of drawing conclusions based on the absence of knowedge from a database. This is nonmonotonic reasoning. The addition of new sentences to the database may be cause for the system to retract earlier conclusions.

In some systems, the exact policy for deriving nonmonotonic conclusions is built into the system. In other systems, the policy can be modified by its user, though rarely within the system's knowledge representation language (e.g. by selecting which predicates to circumscribe). Since KIF is a knowledge representation language and not a system, it is necessary to provide means for its user to express his nomonotonic reasoning policy within the language itself.

We use default rules for this purpose. For instance, the following default rule expresses that an object can be assumed to fly if this object is known to be a bird and it is consistent to assume that it flies.

```
(<<= (flies ?x) (bird ?x) (consis (flies ?x)))
```

The use of `consis` is the only source of nonmonotonicity in KIF. Accordingly, a rule without justifications will be called *monotonic*. This particularly simple case will be discussed first.

## §10.1 Monotonic Rules

A *monotonic rule* is an expression of the following form or its reverse (using `=>>`, where $\phi$, $\phi_1, \ldots, \phi_n$ are sentences.

$$(\texttt{<<=}\ \phi\ \phi_1\ \ldots\ \phi_n),$$

Such an expression should be distinguished from an implication like the following.

$$(\texttt{<=}\ \phi\ \phi_1\ \ldots\ \phi_n)$$

Athough sentences can be monotonic rules, monotonic rules are not sentences; they are similar to *inference rules*, familiar from elementary logic. If, for instance, $\Delta$ consists of some sentences $\Delta_0$ and one rule $(\texttt{<<=}\ \psi\ \phi)$, where $\phi$ and $\psi$ are sentences without free variables, then the set of sentences entailed by $\Delta$ is the smallest set of sentences which (i) contains $\Delta_0$, (ii) is closed under logical entailment, and (iii) contains $\psi$ provided that it contains $\phi$. It is not generally true that this set contains the implication $(\texttt{<=}\ \psi\ \phi)$.

The rationale for using monotonic rules in knowledge representation, instead of implications, is twofold. On the one hand, the "directed" character of rules can simplify the task of developing efficient inference procedures. On the other hand, in some cases, replacing `<<=` by `<=` would be semantically unacceptable. For instance, the rules

```
(<<= (status-known ?x) (citizen ?x))
```

```
(<<= (status-known ?x) (not (citizen ?x)))
```

allow us to infer `(status-known Joe)` only if one of the sentences

```
(citizen Joe),  (not (citizen Joe))
```

can be inferred. Replacing the rules by implications would make `(status-known ?x)` identically true.

## §10.2 Logic Programs

A pure Prolog rule

$$\phi\texttt{:-}\phi_1,\ldots,\phi_n$$

where $\phi, \phi_1, \ldots, \phi_n$ are atoms, can be viewed as a syntactic variant of the monotonic rule

$$\texttt{(<<=}\ \phi\ \phi_1\ \ldots\ \phi_n\texttt{)}$$

except for two important details. First, the declarative semantics of Prolog applies the *unique names assumption* to its ground terms. If, for example, the program contains no function constants, then this assumption can be expressed by the sentences

$$\texttt{(not (=}\ \sigma_1\ \sigma_2\texttt{))}$$

for all distinct object constants $\sigma_1$, $\sigma_2$ in the language of the program. Second, this semantics applies the *closed world assumption* to each relation. For a relation constant $\sigma$, this assumption can be expressed by the following rule.

$$\texttt{(<<= (not (}\sigma\ \texttt{@l)) (consis (not (}\sigma\ \texttt{@l))))}$$

A pure Prolog program can be translated into KIF by appending to it (i) the sentences expressing the unique names assumption, and (ii) the default rules expressing the closed world assumption.

This method can be easily extended to programs with negation as failure. A negative subgoal not $\phi$ is represented in KIF by the premise (consis (not $\phi$)). (Adding consis is necessary because, in KIF, not represents classical negation, rather than negation as failure.)

## §10.3 Circumscribing Abnormality

Extending a set of sentences by the closed world assumption for some relation constant $\sigma$, expressed by a default rule as shown above, has the same effect as circumscribing $\sigma$ (with all object, function and relation constants varied). In particular, circumscribing abnormality can be expressed by the default rule

```
(<<= (not (ab ?aspect ?x)) (consis (not (ab ?aspect ?x))))
```

Consider, for instance, the nonmonotonic database that contains, in addition to this standard default, two facts.

```
(bird tweety)
```

```
(<= (flies ?x) (bird ?x) (not (ab aspect1 ?x)))
```

Birds fly unless they are abnormal in aspect1). This database nonmonotonically entails the conclusion that everything is *not* abnormal, including tweety:

```
(not (ab ?x))
```

From this, we can conclude that tweety flies.

Suppose, on the other hand, that our database includes also the fact that tweety is abnormal in aspect1:

```
(ab aspect1 Tweety)
```

In this case, we can no longer infer that tweety is not abnormal, and, therefore, we cannot conclude that tweety is a flier. Note, however, that we have *not* asserted that tweety cannot fly; we have only prevented the default rule from taking effect in this case.

# Chapter 11
# Definitions

KIF includes a set of definition operators for declaring the category and defining axioms (e.g. "Triangles have 3 sides.") for constants. Such ***analytic definitions*** are intended for use in specifying representation and domain ontologies, and are in contrast to metalinguistic ***substitutional definitions*** that specify new object level syntax in a macro-like fashion.* KIF definitions can be ***complete*** in that they specify an expression that is equivalent to the constant, or ***partial*** in that they specify a defining axiom that restricts the possible denotations of the constant. Partial definitions can be either ***unrestricted*** or ***conservative*** extensions to the language. Conservative definitions are restricted in that adding the defining axioms they specify to any given collection of sentences not containing the constant being defined does not logically entail any additional sentences not containing the constant being defined. [Enderton 72].

An analytic definition associates with the constant being defined a *defining axiom.* Intuitively, the meaning of a definition is that its defining axiom is true and that its defining axiom is an *analytic truth.* Analytic truths are considered to be those sentences that are logically entailed from defining axioms. For example, term subsumption in the KL-ONE family of representation languages is an analytic truth in that it is determined solely on the basis of term definitions. The notions of defining axiom and analytic truth are formally defined as follows.

Given a knowledge base $\Delta$, the sentence `(defining-axiom '`$\sigma$` '`$\phi$`)` means that there is in $\Delta$ an analytic definition of constant $\sigma$ which specifies sentence $\phi$ as a defining axiom of constant $\sigma$. Moreover, defining axioms are true. That is, the following axiom schema holds:

```
(=> (defining-axiom 'σ 'φ) φ)
```

Given a knowledge base $\Delta$, the sentence `(analytic-truth '`$\phi$`)` means that the sentence $\phi$ is logically entailed by the defining axioms of the definitions in knowledge base $\Delta$.

## §11.1 Complete Definitions

Complete definitions specify an equivalent term or sentence for the constant being defined as described below. If a constant has a complete definition in a knowledge base, then no other definition for that constant may occur in the knowledge base. Complete definitions are guaranteed to be conservative extensions of the language.

The following table shows the defining axiom specified by each form of complete definition:

* KIF 3.0 does not provide facilities for substitutional definitions. Consideration is being given to including them in later versions of the language.

| Definition | Defining Axiom |
|---|---|
| `(defobject` $\sigma$ `:=` $\tau$`)` | `(=` $\sigma$ $\tau$`)` |
| `(deffunction` $\pi$ `(`$\nu_1$ ... $\nu_n$ `[`$\omega$`])` `:=` $\tau$`)` | `(=` $\pi$ `(lambda (`$\nu_1$ ... $\nu_n$ `[`$\omega$`])` $\tau$`))` |
| `(defrelation` $\rho$ `(`$\nu_1$ ... $\nu_n$ `[`$\omega$`])` `:=` $\phi$`)` | `(=` $\rho$ `(kappa (`$\nu_1$ ... $\nu_n$ `[`$\omega$`])` $\phi$`))` |

Object constants are defined using the `defobject` operator. In the complete definition of an object constant, the first argument, $\sigma$, is the constant being defined, and the argument, $\tau$, following the := keyword, is a term. For example, the following definition defines the constant `origin` to be the list (0,0,0).

```
(defobject origin := (listof 0 0 0))
```

The defining axiom specified by this definition of `origin` is:

```
(= origin (listof 0 0 0))
```

Function constants are defined using the `deffunction` operator. In the complete definition of a function constant, the first argument, $\pi$, is the constant being defined, the second argument is a list of individual variables with an optional final sequence variable specifying the arguments of the function, and the argument, $\tau$, following the := keyword is a term. For example, the following definition defines the function `paternal-grandfather` in terms of the `father` function.

```
(deffunction paternal-grandfather (?x) := (father (father ?x)))
```

The defining axiom specified by this definition of `paternal-grandfather` is:

```
(= paternal-grandfather (lambda (?x) (father (father ?x))))
```

Relation constants are defined using the `defrelation` operator. In the complete definition of a relation constant, the first argument, $\rho$, is the constant being defined, the second argument is a list of individual variables with an optional final sequence variable specifying the arguments of the relation; and the argument, $\phi$, following the := keyword, is a sentence. For example, the following sentence defines a `bachelor` to be an unmarried man.

```
(defrelation bachelor (?x) := (and (man ?x) (not (married ?x))))
```

The defining axiom specified by this definition of `bachelor` is:

```
(= bachelor (kappa (?x) (and (man ?x) (not (married ?x)))))
```

## §11.2 Partial Definitions

A constant can have multiple partial definitions, each of which restricts the possible denotations of the constant. All the definitions of a constant must declare the constant to be the same category; i.e., they must all use the same operator – `defobject`, `deffunction`, or `defrelation`. The defining axioms specified by partial definitions can be either unrestricted or optionally required to be conservative extensions to the language.

## Unrestricted Partial Definitions

Unrestricted partial definitions can specify any sentence as a defining axiom, as described below. The following table shows the defining axiom specified by each form of unrestricted partial definition:

| Definition | Defining Axiom |
|---|---|
| `(defobject` $\sigma$ $\phi_1$ ... $\phi_n$`)` | `(and` $\phi_1$ ... $\phi_n$`)` |
| `(deffunction` $\pi$ $\phi_1$ ... $\phi_n$`)` | `(and` $\phi_1$ ... $\phi_n$`)` |
| `(defrelation` $\rho$ $\phi_1$ ... $\phi_n$`)` | `(and` $\phi_1$ ... $\phi_n$`)` |
| `(defrelation` $\rho$ `(`$\nu_1$ ... $\nu_n$`)` `:=>` $\phi_1$ `:axiom` $\phi_2$`)` | `(and (=> (member ?x` $\rho$`) (= (length ?x) n))` `(=> (`$\rho$ $\nu_1$ ... $\nu_n$`)` $\phi_1$`)` $\phi_2$`)` |
| `(defrelation` $\rho$ `(`$\nu_1$ ... $\nu_n$ $\omega$`)` `:=>` $\phi_1$ `:axiom` $\phi_2$`)` | `(and (=> (member ?x` $\rho$`) (>= (length ?x) n))` `(=> (`$\rho$ $\nu_1$ ... $\nu_n$ $\omega$`)` $\phi_1$`)` $\phi_2$`)` |

In an unrestricted partial definition of an object constant, the first argument, $\sigma$, is the constant being defined, and the remaining arguments, $\phi_1$ ... $\phi_n$, are sentences. For example, the following definition defines the constant `id` to be a left and right identity for the binary function `f`.

```
(defobject id (= (f ?x id) ?x) (= (f id ?x) ?x))
```

The defining axiom specified by this definition of `id` (which is just the
conjunction of the second and third arguments in the definition) is unrestricted in that it may contradict other partial definitions of `id` and `f` may not have a left and right identity.

In an unrestricted partial definition of a function constant, the first argument, $\pi$, is the constant being defined and the remaining arguments, $\phi_1$ ... $\phi_n$, are sentences. For example, the following definition defines `f` to be a function which has a value that is greater than 1 for all numbers.

```
(deffunction f (=> (number ?y) (> (f ?y) 1)))
```

The defining axiom specified by this definition of `f` is just the implication that is the second argument in the definition.

There are two basic forms of unrestricted partial definitions for relations. Both forms allow inclusion of an arbitrary sentence to be a defining axiom for the constant being defined. The second form additionally provides for the specification of necessary conditions for the relation to hold. The second form has two variants, depending on whether a sequence variable is included in the function's argument list.

In the first form of unrestricted partial definition of a relation constant, the first argument, $\rho$, is the constant being defined and the remaining arguments, $\phi_1$ ... $\phi_n$, are sentences. For example, the following definition defines R to be a relation that holds for all single arguments that are positive numbers.

```
(defrelation R (=> (> ?z 0) (R ?z)))
```

The defining axiom specified by this definition of R is just the implication that is the second argument in the definition.

In the second form of unrestricted partial definition of a relation constant, the first argument, $\rho$, is the constant being defined, the second argument is a list of individual variables specifying the arguments of the relation, and the arguments, $\phi_1$ and $\phi_2$, following the `:=>` and `:axiom` keywords, are sentences. The form has two variants, depending on whether the argument list includes a sequence variable. The following is an example of this form of definition in which above is defined to be a binary transitive relation that holds only for "located objects".

```
(defrelation above (?b1 ?b2)
     :=> (and (located-object ?b1) (located-object ?b2))
     :axiom (transitive above))
```

The defining axiom specified by this definition of above is:

```
(and (=> (member ?x above) (= (length ?x) 2))
     (=> (above ?b1 ?b2)
         (and (located-object ?b1) (located-object ?b2)))
     (transitive above))
```

## Conservative Partial Definitions

Conservative partial definitions specify defining axioms that are conservative extensions of the language. A defining axiom is a conservative extension if adding it to any given collection of sentences not containing the constant being defined does not logically entail any additional sentences not containing the constant being defined. The defining axioms specified by complete definitions and the defining axioms produced directly from some forms of partial definitions are necessarily conservative extensions. However, the arbitrary sentences that can be included in partial definitions are not in general conservative extensions of the language and therefore must be transformed into a conditional form of defining axiom that is guaranteed to be conservative. If a knowledge base contains conservative partial definitions containing arbitrary sentences for a given constant, then those definitions specify a single *conditional defining axiom* for that constant as described below.

The following table shows the defining axiom(s) specified by each form of conservative partial definition:

| Definition | Defining Axiom |
|---|---|
| (defobject $\sigma$) | (objconst (quote $\sigma$)) |
| (defobject $\sigma$ :conservative-axiom $\phi$) | *The conditional defining axiom for* $\sigma$ |
| (deffunction $\pi$) | (funconst (quote $\pi$)) |
| (deffunction $\pi$ :conservative-axiom $\phi$) | *The conditional defining axiom for* $\pi$ |
| (defrelation $\rho$) | (relconst (quote $\rho$)) |
| (defrelation $\rho$ :conservative-axiom $\phi$) | *The conditional defining axiom for* $\rho$ |
| (defrelation $\rho$ ($\nu_1$ ... $\nu_n$) :=> $\phi_1$) | (and (=> (member ?x $\rho$) (= (length ?x) n)) (=> ($\rho$ $\nu_1$ ... $\nu_n$) $\phi_1$)) |
| (defrelation $\rho$ ($\nu_1$ ... $\nu_n$ $\omega$) :=> $\phi_1$) | (and (=> (member ?x $\rho$) (>= (length ?x) n)) (=> ($\rho$ $\nu_1$ ... $\nu_n$ $\omega$) $\phi_1$)) |
| (defrelation $\rho$ ($\nu_1$ ... $\nu_n$) :=> $\phi_1$ :conservative-axiom $\phi_2$) | (and (=> (member ?x $\rho$) (= (length ?x) n)) (=> ($\rho$ $\nu_1$ ... $\nu_n$) $\phi_1$)) *The conditional defining axiom for* $\rho$ |
| (defrelation $\rho$ ($\nu_1$ ... $\nu_n$ $\omega$) :=> $\phi_1$ :conservative-axiom $\phi_2$) | (and (=> (member ?x $\rho$) (>= (length ?x) n)) (=> ($\rho$ $\nu_1$ ... $\nu_n$ $\omega$) $\phi_1$)) *The conditional defining axiom for* $\rho$ |

There are two forms of conservative partial definitions for objects. In the first form, the argument, $\sigma$, is the constant being defined, and the definition simply declares that the constant denotes an object. In the second form, the first argument, $\sigma$, is the constant being defined, and the argument, $\phi$, following the `:conservative-axiom` keyword is a sentence. The second form of definition provides a sentence to be included in the conditional defining axiom for $\sigma$, as described below.

There are two forms of conservative partial definitions for functions. In the first form, the argument, $\pi$, is the constant being defined, and the definition simply declares that the constant denotes a function. In the second form, the first argument, $\pi$, is the constant being defined and the argument, $\phi$, following the `:conservative-axiom` keyword is a sentence. The second form of definition provides a sentence to be included in the conditional defining axiom for $\pi$, as described below.

There are three basic forms of conservative partial definitions for relations. In the first form, the argument, $\rho$, is the constant being defined, and the definition simply declares that the constant denotes a relation. In the second form, the first argument, $\rho$, is the constant being defined and the argument, $\phi$, following the `:conservative-axiom` keyword is a sentence. The second form of definition provides a sentence to be included in the conditional defining axiom for $\rho$ as described below.

The third form of conservative partial definition of a relation constant provides for the specification of necessary conditions for the relation to hold and optionally provides an arbitrary sentence to be included in the constant's conditional defining axiom. The third form has four variants, depending on whether the optional sentence is included and whether a sequence variable is included in the function's argument list.

In the third form of conservative partial definition of a relation constant, the first argument, $\rho$, is the constant being defined; the second argument is a list of individual variables with an optional final sequence variable, $\omega$, specifying the arguments of the relation; $\phi_1$, in the keyword-argument pair, `:=>` $\phi_1$, is a sentence; and $\phi_2$, in the optional final keyword-argument pair, `:conservative-axiom` $\phi_2$, is a sentence. For example, the following definition defines a person to necessarily be a mammal.

```
(defrelation person (?x) :=> (mammal ?x))
```

The defining axiom produced by this definition of `person` is:

```
(and (=> (member ?x person) (= (length ?x) 1))
     (=> (person ?x) (mammal ?x)))
```

The sentences following the keyword `:conservative-axiom` in all of the partial definitions for a given constant are used to form a single conservative defining axiom for that constant. The defining axiom essentially states that if an entity exists in the domain of discourse having all the properties ascribed to the constant by its definitions, then the constant denotes such an entity and the sentences in the constant's definitions following the keyword `:conservative-axiom` are true. That defining axiom is formed as follows.

For a given knowledge base $\Delta$ and a given constant $\sigma$, let $\phi_1$,...,$\phi_i$ be the sentences following the keyword `:conservative-axiom` in partial definitions of $\sigma$ in $\Delta$, and $\phi_{i+1}$,...,$\phi_n$ be the defining axioms otherwise specified in partial definitions of $\sigma$ in $\Delta$. The sentences $\phi_1$,...,$\phi_i$ specify the following conservative defining axiom:

(=> (exists ?x $\phi_{1(\sigma\rightarrow ?x)}$ ... $\phi_{n(\sigma\rightarrow ?x)}$)
    (and $\phi_1$ ... $\phi_i$)),

where ?x is an individual variable that does not occur in any $\phi_j$, and for each j = 1,...,n, $\phi_{j(\sigma\rightarrow ?x)}$ is $\phi_j$ with the following substitutions:

- Each occurrence of $\sigma$ as a term is replaced by `?x`.
- Each occurrence of ($\sigma$ `<args>`) as a function term is replaced by `(value ?x <args>)`.
- Each occurrence of ($\sigma$ `<args>`) as a relational sentence is replaced by `(holds ?x <args>)`.

This form of defining axiom cannot introduce an inconsistency into a knowledge base since any inconsistency will occur in the antecedent of the implication, thus making the antecedent false and blocking the entailment of the consequent. Also, this form of defining axiom cannot introduce a new domain fact about other constants (e.g., `(color Clyde` grey)), since such a domain fact will occur in the antecedent of the defining axiom and will therefore block the implication of the consequent if it is not already true.

Note that, in general, a constant can have infinitely many partial definitions (metalinguistically specified by a definition schema). However, if any of the partial definitions of a constant contain a sentence following the keyword :conservative-axiom, then the constant must have only a finite number of definitions. Otherwise, the conditional defining axiom for that constant would be an infinitely long sentence, which is not allowed in KIF.

As an example of conservative partial definitions containing arbitrary sentences, consider the following conservative version of the definition given above of id, a left and right identity for f.

```
(defobject id :conservative-axiom (= (f ?x id) ?x))

(defobject id :conservative-axiom (= (f id ?x) ?x))
```

Assuming there are no other definitions of id in the knowledge base, these two partial definitions produce a single defining axiom for id as follows:

```
(=> (exists ?y (and (= (f ?x ?y) ?x) (= (f ?y ?x) ?x)))
    (and (= (f ?x id) ?x) (= (f id ?x) ?x)))
```

This axiom states that if there exists a left and right identity for f, then id is that identity.

The following table summarizes all the forms of KIF definitions and the defining axioms specified by each.

| Definition | Defining Axiom |
|---|---|
| `(defobject` $\sigma$ `:=` $\tau$`)` | `(=` $\sigma$ $\tau$`)` |
| `(defobject` $\sigma$ $\phi_1$ `...` $\phi_n$`)` | `(and` $\phi_1$ `...` $\phi_n$`)` |
| `(defobject` $\sigma$`)` | `(objconst (quote` $\sigma$`))` |
| `(defobject` $\sigma$ `:conservative-axiom` $\phi$`)` | *The conditional defining axiom for* $\sigma$ |
| `(deffunction` $\pi$ `(`$\nu_1$ `...` $\nu_n$ `[`$\omega$`]) :=` $\tau$`)` | `(=` $\pi$ `(lambda (`$\nu_1$ `...` $\nu_n$ `[`$\omega$`])` $\tau$`))` |
| `(deffunction` $\pi$ $\phi_1$ `...` $\phi_n$`)` | `(and` $\phi_1$ `...` $\phi_n$`)` |
| `(deffunction` $\pi$`)` | `(funconst (quote` $\pi$`))` |
| `(deffunction` $\pi$ `:conservative-axiom` $\phi$`)` | *The conditional defining axiom for* $\pi$ |
| `(defrelation` $\rho$ `(`$\nu_1$ `...` $\nu_n$ `[`$\omega$`]) :=` $\phi$`)` | `(=` $\rho$ `(kappa (`$\nu_1$ `...` $\nu_n$ `[`$\omega$`])` $\phi$`))` |
| `(defrelation` $\rho$ $\phi_1$ `...` $\phi_n$`)` | `(and` $\phi_1$ `...` $\phi_n$`)` |
| `(defrelation` $\rho$`)` | `(relconst (quote` $\rho$`))` |
| `(defrelation` $\rho$ `:conservative-axiom` $\phi$`)` | *The conditional defining axiom for* $\rho$ |
| `(defrelation` $\rho$ `(`$\nu_1$ `...` $\nu_n$`)` `:=>` $\phi_1$ `[:axiom` $\phi_2$`])` | `(and (=> (member ?x` $\rho$`) (= (length ?x) n))` `(=> (`$\rho$ $\nu_1$ `...` $\nu_n$`)` $\phi_1$`))` `[`$\phi_2$`])` |
| `(defrelation` $\rho$ `(`$\nu_1$ `...` $\nu_n$ $\omega$`)` `:=>` $\phi_1$ `[:axiom` $\phi_2$`])` | `(and (=> (member ?x` $\rho$`) (>= (length ?x) n))` `(=> (`$\rho$ $\nu_1$ `...` $\nu_n$ $\omega$`)` $\phi_1$`))` `[`$\phi_2$`])` |
| `(defrelation` $\rho$ `(`$\nu_1$ `...` $\nu_n$`)` `:=>` $\phi_1$ `[:conservative-axiom` $\phi_2$`])` | `(and (=> (member ?x` $\rho$`) (= (length ?x) n))` `(=> (`$\rho$ $\nu_1$ `...` $\nu_n$`)` $\phi_1$`))` [*The conditional defining axiom for* $\rho$] |
| `(defrelation` $\rho$ `(`$\nu_1$ `...` $\nu_n$ $\omega$`)` `:=>` $\phi_1$ `[:conservative-axiom` $\phi_2$`])` | `(and (=> (member ?x` $\rho$`) (>= (length ?x) n))` `(=> (`$\rho$ $\nu_1$ `...` $\nu_n$ $\omega$`)` $\phi_1$`))` [*The conditional defining axiom for* $\rho$] |

# Chapter A
# Abstract Algebra

This appendix contains an ontology for the basic concepts in abstract algebra. The first section gives properties of binary functions. The second section does the same for binary relations. In the third section, these properties are used in defining the a variety of common algebraic structures.

## §A.1 Binary Operations

```
(defrelation binop (?f ?s) :=
  (and (binary-function ?f)
       (subset (universe ?f) ?s)))                                (A.1)
```

```
(defrelation associative (?f ?s) :=
  (forall (?x ?y ?z)
    (=> (member ?x ?s) (member ?y ?s) (member ?z ?s)
        (= (value ?f ?x (value ?f ?y ?z))
           (value ?f (value ?f ?x ?y) ?z)))))                     (A.2)
```

```
(defrelation commutative (?f ?s) :=
  (forall (?x ?y)
    (=> (member ?x ?s) (member ?y ?s)
        (= (value ?f ?x ?y) (value ?f ?y ?x)))))                  (A.3)
```

```
(defrelation invertible (?f ?o ?s) :=
  (forall (?x)
    (=> (memberp ?x ?s)
        (exists (?y)
          (and (member ?y ?s)
               (= (value ?x ?y) ?o) (= (value ?y ?x) ?o))))))     (A.4)
```

```
(defrelation distributes (?f ?g ?s) :=
  (and (binop ?f ?s) (binop ?g ?s)
       (forall (?x ?y ?z)
         (=> (member ?x ?s) (member ?y ?s) (member ?z ?s)
             (= (value ?f (value ?g ?x ?y) ?z)
                (value ?g (value ?f ?x ?z)
                          (value ?f ?y ?z)))))))                  (A.5)
```

## §A.2 Binary Relations

```
(defrelation binrel (?r ?s) :=
  (and (binary-relation ?r)
        (subset (universe ?r) ?s)))                                (A.6)

    (defrelation reflexive (?r ?s) :=
      (and (binrel ?r ?s)
           (forall ?x (=> (member ?x ?s)
                          (holds ?r ?x ?x)))))                     (A.7)

    (defrelation irreflexive (?r ?s) :=
      (and (binrel ?r ?s)
           (forall (?x)
                   (=> (member ?x ?s) (not (holds ?r ?x ?x))))))   (A.8)

    (defrelation symmetric (?r ?s) :=
      (and (binrel ?r ?s)
           (forall (?x ?y) (=> (holds ?r ?x ?y) (holds ?r ?y ?x)))))  (A.9)

    (defrelation asymmetric (?r ?s) :=
      (and (binrel ?r ?s)
           (forall (?x ?y) (=> (holds ?r ?x ?y))
                               (not (holds ?r ?y ?x))))))          (A.10)

    (defrelation antisymmetric (?r ?s) :=
      (and (binrel ?r ?s)
           (forall (?x ?y)
             (=> (holds ?r ?x ?y) (holds ?r ?y ?x) (= ?x ?y)))))   (A.11)

    (defrelation trichotomizes (?r ?s) :=
      (and (binrel ?r ?s)
           (forall (?x ?y)
             (=> (member ?x ?s) (member ?y ?s)
                 (or (holds ?r ?x ?y)
                     (= ?x ?y)
                     (holds ?r ?y ?x))))))                         (A.12)

    (defrelation transitive (?r ?s) :=
      (and (binrel ?r ?s)
           (forall (?x ?y ?z)
             (=> (holds ?r ?x ?y) (holds ?r ?y ?z)
                 (holds ?r ?x ?z)))))                              (A.13)
```

## §A.3 Algebraic Structures

```
    (defrelation semigroup (?s ?f ?o) :=
      (and (binop ?f ?s)
       (associative ?f ?s)
       (identity ?o ?f ?s)))                                    (A.14)

(defrelation abelian-semigroup (?s ?f ?o) :=
  (and (semigroup ?s ?f ?o)
       (commutative ?f ?s)))                                    (A.15)

(defrelation group (?s ?f ?o) :=
  (and (binop ?f ?s)
       (associative ?f ?s)
       (identity ?o ?f ?s)
       (invertible ?f ?o ?s)))                                  (A.16)

(defrelation abelian-group (?s ?f ?o) :=
  (and (group ?s ?f ?o)
       (commutative ?f ?s)))                                    (A.17)

(defrelation ring (?s ?f ?o ?g ?i) :=
  (and (abelian-group ?s ?f ?o)
       (semigroup ?s ?g ?i)
       (distributes ?g ?f ?s)))                                 (A.18)

(defrelation commutative-ring (?s ?f ?o ?g ?i) :=
  (and (abelian-group ?s ?f ?o)
       (abelian-semigroup ?s ?g ?i)
       (distributes ?g ?f ?s)))                                 (A.19)

(defrelation integral-domain (?s ?f ?o ?g ?i) :=
  (and (commutative-ring ?s ?f ?o ?g ?i)
       (operation ?g (difference ?s (setof ?o)))))              (A.20)

(defrelation division-ring (?s ?f ?o ?g ?i) :=
  (and (ring ?s ?f ?o ?g ?i)
       (binop ?g (difference ?s (setof ?o)))
       (invertible ?g (difference ?s (setof ?o)))))             (A.21)

(defrelation field (?s ?f ?o ?g ?i) :=
  (and (division-ring ?s ?f ?o ?g ?i)
       (commutative ?f ?s)))                                    (A.22)

(defrelation partial-order (?s ?r) :=
  (and (irreflexive ?r ?s)
       (asymmetric ?r ?s)
       (transitive ?r ?s)))                                     (A.23)
```

```
(defrelation linear-order (?s ?r) :=
  (and (irreflexive ?r ?s)
       (asymmetric ?r ?s)
       (transitive ?r ?s)
       (trichotomizes ?r ?s)))                    (A.24)
```

# APPENDIX D

## Management of Knowledge Representation Standards Activities NASA-Ames Contract NCC 2-719

## Final Report covering the period of 6/1/1991 - 5/31/1993

## Principal Investigator

Ramesh S. Patil
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
Phone: (310) 822-1511
ramesh@isi.edu

# Working Draft
# Description-Logic Knowledge Representation System Specification from the KRSS Group of the ARPA Knoweledge Sharing Effort

Peter F. Patel-Schneider
AT&T Bell Laboratories
Murry Hill, New Jersey

William Swartout.
Information Sciences Institute
University of Southern California

# Working Version (Draft): Description-Logic Knowledge Representation System Specification from the KRSS Group of the ARPA Knowledge Sharing Effort

Peter F. Patel-Schneider, co-chair
Bill Swartout, co-chair

1 November 1993

## 1 Overview

This is the KRSS group specification for description-logic-based KR systems. It describes the required behavior for compliant KR systems. This report is not an overview of description logics, nor is it a rationale for using description logics in knowledge representation.

The description logic in this specification is based closely on the description logic defined by researchers at DFKI [?]. However, it includes several other features, notably role closures and rules.

A knowledge base in this specification is a sequence of statements. The semantics of non-rule, non-closure statements is similar to that in the DFKI proposal. The semantics of role closures is determined by replacing them, in sequence, with the best derivable role maximum or the derivable set of fillers. Rules are treated as epistemic statements, in line with their treatment by Donini *et al* [?].

Compliant implementations are required to parse the entire language, but may replace constructs that they cannot reason about with the closest approximation that they can handle. Compliant implementations are required to be complete on a subset of the logic, selected for its easy, but non-trivial, inferences.

## 2 Syntax

Major parts of the syntax of knowledge bases are taken from [?]. The top-level syntactic categories in the specification are descriptions, statements, knowledge bases, and inquiries.

Throughout the specification, C, R, A, I, CI, and S, possibly subscripted, are concepts, roles (including attributes), attributes, individuals, concrete individuals, and assertions, respectively; N, CN, RN, AN, IN, AIN, DIN, XN, and GN, possibly subscripted, are names of any sort, concept names, role names (including attribute names), attribute names, individual names, anonymous individual names, distinct individual names, rule names, and group names, respectively; QQ and RR are queries and retrievals, respectively. The syntax of names, numbers, integers, and strings are the same as in LISP.

The three kinds of descriptions in the specification are concepts, roles (including attributes), and individuals. Concepts are either concept names or are formed using the operators in Table ??.[1] Roles (attributes) are either role (attribute) names or are formed using the operators in Table ?? (??). Individuals are either individual names or concrete individuals. Concrete individuals are either numbers or strings. Individual names are either distinct individual names or anonymous individual names.

Statements are formed according to Tables ?? and ??.

A knowledge base is a sequence of statements in which there is exactly one definition of every occurring concept, role, attribute, individual, and rule name. Concept, role, attribute, and individual names must be defined before their first use, so no cyclic definitions are allowed. The name spaces of concepts, roles, individuals, and rules are distinct (i.e., there may be a concept and a role with the same name). The name space of attributes is a sub-space of the name space of roles.

Comment lines, starting with a ';', are allowed in knowledge bases.

[1]These tables include the abstract form syntax from [?], so that the logic here can be compared with the many published papers using this abstract form.

| Syntax | | Extension |
|---|---|---|
| **Input** | **Abstract** | |
| TOP | $\top$ | $\Delta^{\mathcal{I}}$ |
| BOTTOM | $\bot$ | $\emptyset$ |
| NUMBER | | the numbers |
| INTEGER | | the integers |
| STRING | | the strings |
| (and $C_1$ ... $C_n$) | $C_1 \sqcap \cdots \sqcap C_n$ | $C_1{}^{\mathcal{I}} \cap \cdots \cap C_n{}^{\mathcal{I}}$ |
| (or $C_1$ ... $C_n$) | $C_1 \sqcup \cdots \sqcup C_n$ | $C_1{}^{\mathcal{I}} \cup \cdots \cup C_n{}^{\mathcal{I}}$ |
| (not C) | $\neg C$ | $\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$ |
| (all R C) | $\forall R{:}C$ | $\{d \in \Delta_a^{\mathcal{I}} \mid R^{\mathcal{I}}(d) \subseteq C^{\mathcal{I}}\}$ |
| (some R) | $\exists R$ | $\{d \in \Delta_a^{\mathcal{I}} \mid R^{\mathcal{I}}(d) \neq \emptyset\}$ |
| (none R) | $\uparrow R$ | $\{d \in \Delta_a^{\mathcal{I}} \mid R^{\mathcal{I}}(d) = \emptyset\}$ |
| (at-least n R) | $\geq n\, R$ | $\{d \in \Delta_a^{\mathcal{I}} \mid \lvert R^{\mathcal{I}}(d)\rvert \geq n\}$ |
| (at-most n R) | $\leq n\, R$ | $\{d \in \Delta_a^{\mathcal{I}} \mid \lvert R^{\mathcal{I}}(d)\rvert \leq n\}$ |
| (exactly n R) | $= n\, R$ | $\{d \in \Delta_a^{\mathcal{I}} \mid \lvert R^{\mathcal{I}}(d)\rvert = n\}$ |
| (some R C) | $\exists R.C$ | $\{d \in \Delta_a^{\mathcal{I}} \mid R^{\mathcal{I}}(d) \cap C^{\mathcal{I}} \neq \emptyset\}$ |
| (at-least n R C) | $\geq n\, R{:}C$ | $\{d \in \Delta_a^{\mathcal{I}} \mid \lvert R^{\mathcal{I}}(d) \cap C^{\mathcal{I}}\rvert \geq n\}$ |
| (at-most n R C) | $\leq n\, R{:}C$ | $\{d \in \Delta_a^{\mathcal{I}} \mid \lvert R^{\mathcal{I}}(d) \cap C^{\mathcal{I}}\rvert \leq n\}$ |
| (exactly n R C) | $= n\, R{:}C$ | $\{d \in \Delta_a^{\mathcal{I}} \mid \lvert R^{\mathcal{I}}(d) \cap C^{\mathcal{I}}\rvert = n\}$ |
| (equal $R_1$ $R_2$) | $R_1 = R_2 \sqcap \exists R_1$ | $\{d \in \Delta_a^{\mathcal{I}} \mid R_1{}^{\mathcal{I}}(d) = R_2{}^{\mathcal{I}}(d) \wedge R_1{}^{\mathcal{I}}(d) \neq \emptyset\}$ |
| (not-equal $R_1$ $R_2$) | $R_1 \neq R_2$ | $\{d \in \Delta_a^{\mathcal{I}} \mid R_1{}^{\mathcal{I}}(d) \neq R_2{}^{\mathcal{I}}(d)\}$ |
| (subset $R_1$ $R_2$) | $R_1 \subseteq R_2$ | $\{d \in \Delta_a^{\mathcal{I}} \mid R_1{}^{\mathcal{I}}(d) \subseteq R_2{}^{\mathcal{I}}(d)\}$ |
| (fillers R $I_1$ ... $I_n$) | $R{:}I_1 \sqcap \cdots \sqcap R{:}I_n$ | $\{d \in \Delta_a^{\mathcal{I}} \mid R^{\mathcal{I}}(d) \supseteq \{I_1{}^{\mathcal{I}}, \ldots, I_n{}^{\mathcal{I}}\}\}$ |
| (only-fillers R $I_1$ ... $I_n$) | | $\{d \in \Delta_a^{\mathcal{I}} \mid R^{\mathcal{I}}(d) = \{I_1{}^{\mathcal{I}}, \ldots, I_n{}^{\mathcal{I}}\}\}$ |
| (in A C) | $A{:}C$ | $\{d \in \Delta_a^{\mathcal{I}} \mid A^{\mathcal{I}}(d) \in C^{\mathcal{I}}\}$ |
| (is A I) | $A{:}I$ | $\{d \in \Delta_a^{\mathcal{I}} \mid A^{\mathcal{I}}(d) = I^{\mathcal{I}}\}$ |
| (set $I_1$ ... $I_n$) | $\{I_1, \ldots, I_n\}$ | $\{I_1{}^{\mathcal{I}}, \ldots, I_n{}^{\mathcal{I}}\}$ |
| (minimum CI) | | $\{d \in \Delta_c^{\mathcal{I}} :\mid d \geq CI^{\mathcal{I}}\}$ |
| (maximum CI) | | $\{d \in \Delta_c^{\mathcal{I}} :\mid d \leq CI^{\mathcal{I}}\}$ |
| (satisfies...) | | see text |

Table 1: Concept Syntax and Semantics

| Syntax | | Extension |
|---|---|---|
| Input | Abstract | |
| top | $\top$ | $\Delta_a^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ |
| bottom | $\perp$ | $\emptyset$ |
| identity | $id$ | $\{(d,d) \mid d \in \Delta_a^{\mathcal{I}}\}$ |
| (and R1 ... Rn) | $\mathsf{R}_1 \sqcap \cdots \sqcap \mathsf{R}_n$ | $\mathsf{R}_1{}^{\mathcal{I}} \cap \cdots \cap \mathsf{R}_n{}^{\mathcal{I}}$ |
| (or R1 ... Rn) | $\mathsf{R}_1 \sqcup \cdots \sqcup \mathsf{R}_n$ | $\mathsf{R}_1{}^{\mathcal{I}} \cup \cdots \cup \mathsf{R}_n{}^{\mathcal{I}}$ |
| (not R) | $\neg\mathsf{R}$ | $(\Delta_a^{\mathcal{I}} \times \Delta^{\mathcal{I}}) \setminus \mathsf{R}^{\mathcal{I}}$ |
| (inverse R) | $\mathsf{R}^{-1}$ | $(\mathsf{R}^{\mathcal{I}})^{-1} \cap (\Delta_a^{\mathcal{I}} \times \Delta^{\mathcal{I}})$ |
| (restrict R C) | $\mathsf{R} \mid \mathsf{C}$ | $\mathsf{R}^{\mathcal{I}} \cap (\Delta_a^{\mathcal{I}} \times \mathsf{C}^{\mathcal{I}})$ |
| (compose R1 ... Rn) | $\mathsf{R}_1 \circ \cdots \circ \mathsf{R}_n$ | $\mathsf{R}_1{}^{\mathcal{I}} \circ \cdots \circ \mathsf{R}_n{}^{\mathcal{I}}$ |
| (range C) | | $\Delta_a^{\mathcal{I}} \times \mathsf{C}^{\mathcal{I}}$ |
| (domain C) | | $(\mathsf{C}^{\mathcal{I}} \cap \Delta_a^{\mathcal{I}}) \times \Delta^{\mathcal{I}}$ |
| (domain-range C1 C2) | $\mathsf{C}_1 \times \mathsf{C}_2$ | $(\mathsf{C}_1{}^{\mathcal{I}} \cap \Delta_a^{\mathcal{I}}) \times \mathsf{C}_2{}^{\mathcal{I}}$ |
| (transitive-closure R) | $\mathsf{R}^+$ | $\bigcup_{n \geq 1} (\mathsf{R}^{\mathcal{I}})^n$ |
| (transitive-reflexive-closure R) | $\mathsf{R}^*$ | $\{(d,d) \mid d \in \Delta_a^{\mathcal{I}}\} \cup \bigcup_{n \geq 1} (\mathsf{R}^{\mathcal{I}})^n$ |
| (satisfies...) | | see text |

Table 2: Role Syntax and Semantics

| Syntax | | Extension |
|---|---|---|
| Input | Abstract | |
| bottom | $\perp$ | $\emptyset$ |
| identity | $id$ | $\{(d,d) \mid d \in \Delta_a^{\mathcal{I}}\}$ |
| (and A1 R2 ... Rn) | $\mathsf{A}_1 \sqcap \mathsf{R}_2 \sqcap \cdots \sqcap \mathsf{R}_n$ | $\mathsf{A}_1{}^{\mathcal{I}} \cap \mathsf{R}_2{}^{\mathcal{I}} \cap \cdots \cap \mathsf{R}_n{}^{\mathcal{I}}$ |
| (restrict A C) | $\mathsf{A} \mid \mathsf{C}$ | $\mathsf{A}^{\mathcal{I}} \cap (\Delta_a^{\mathcal{I}} \times \mathsf{C}^{\mathcal{I}})$ |
| (compose A1 ... An) | $\mathsf{A}_1 \circ \cdots \circ \mathsf{A}_n$ | $\mathsf{A}_1{}^{\mathcal{I}} \circ \cdots \circ \mathsf{A}_n{}^{\mathcal{I}}$ |

Table 3: Attribute Syntax and Semantics

| Syntax | | Semantics |
|---|---|---|
| **Input** | **Abstract** | |
| (define-concept CN C) | $\mathsf{CN} \doteq \mathsf{C}$ | $\mathsf{CN}^{\mathcal{I}} = \mathsf{C}^{\mathcal{I}}$ |
| (define-primitive-concept CN C) | $\mathsf{CN} \sqsubseteq \mathsf{C}$ | $\mathsf{CN}^{\mathcal{I}} \subseteq \mathsf{C}^{\mathcal{I}}$ |
| (define-disjoint-primitive-concept CN ($\mathsf{GN}_1$ ... $\mathsf{GN}_n$) C) | | see text |
| (define-role RN R) | $\mathsf{RN} \doteq \mathsf{R}$ | $\mathsf{RN}^{\mathcal{I}} = \mathsf{R}^{\mathcal{I}}$ |
| (define-primitive-role RN R) | $\mathsf{RN} \sqsubseteq \mathsf{R}$ | $\mathsf{RN}^{\mathcal{I}} \subseteq \mathsf{R}^{\mathcal{I}}$ |
| (define-attribute AN A) | $\mathsf{AN} \doteq \mathsf{A}$ | $\mathsf{AN}^{\mathcal{I}} = \mathsf{A}^{\mathcal{I}}$ |
| (define-primitive-attribute AN R) | $\mathsf{AN} \sqsubseteq \mathsf{R}$ | $\mathsf{AN}^{\mathcal{I}} \subseteq \mathsf{R}^{\mathcal{I}}$ |
| (define-distinct-individual DIN) | | see text |
| (define-anonymous-individual AIN) | | see text |
| (define-rule XN CN C) | | see text |
| (state S) | | $\mathsf{S}^{\mathcal{I}}$ |
| (close-role IN R) | | see text |
| (close-role-fillers IN R) | | see text |

Table 4: Statement Syntax and Semantics

| Syntax | | Semantics |
|---|---|---|
| **Input** | **Abstract** | |
| (and $\mathsf{S}_1$ ... $\mathsf{S}_n$) | | $\mathsf{S}_1{}^{\mathcal{I}} \wedge \cdots \wedge \mathsf{S}_n{}^{\mathcal{I}}$ |
| (or $\mathsf{S}_1$ ... $\mathsf{S}_n$) | | $\mathsf{S}_1{}^{\mathcal{I}} \vee \cdots \vee \mathsf{S}_n{}^{\mathcal{I}}$ |
| (not S) | | $\neg \mathsf{S}^{\mathcal{I}}$ |
| (instance IN C) | $\mathsf{IN} \in \mathsf{C}$ | $\mathsf{IN}^{\mathcal{I}} \in \mathsf{C}^{\mathcal{I}}$ |
| (related IN I R) | $\langle \mathsf{IN}, \mathsf{I} \rangle \in \mathsf{R}$ | $\langle \mathsf{IN}^{\mathcal{I}}, \mathsf{I}^{\mathcal{I}} \rangle \in \mathsf{R}^{\mathcal{I}}$ |
| (equal $\mathsf{IN}_1$ $\mathsf{IN}_2$) | | $\mathsf{IN}_1{}^{\mathcal{I}} = \mathsf{IN}_2{}^{\mathcal{I}}$ |

Table 5: Assertion Syntax and Semantics

## 3 Semantics

The semantics of the description logic is defined in terms of interpretations and model-sets. The interpretation part of the semantics is mostly taken from [?]. The idea of (epistemic) model-sets is taken from [?].

Semantics are defined directly only for simple knowledge bases. A simple knowledge base is a knowledge base without any role or role fillers closure statements or disjoint primitive definitions. Simple knowledge bases can also contain disjointness statements of the form (disjoint $CN_1$ $CN_2$) where the concept names have been primitively defined.

A non-simple knowledge base is transformed into a simple knowledge base by modifying, in order, each role or role filler closure or disjoint primitive definition as follows:

1. A role closure, (close-role IN R), is replaced by (instance IN (at-most n R)), where n is the largest integer such that (instance IN (at-least n R)) follows from the simplified version of the portion of the knowledge base before the role closure, provided that there is such an n. Otherwise the role closure is ignored.

2. A role fillers closure, (close-role-fillers IN R), is replaced by (instance IN (only-fillers R $I_1$ ... $I_n$)), where the $I_i$ are the individuals such that (instance IN (fillers R $I_i$)) follows from the simplified version of the portion of the knowledge base before the role fillers closure.

3. A disjoint primitive definition, (define-disjoint-primitive-concept CN ($GN_1$ ... $GN_n$) C), is changed to (define-primitive-concept CN C), (disjoint CN $CN_1$), ..., (disjoint CN $CN_m$), where the $CN_i$ are the disjoint primitive concepts in the portion of the knowledge base before this definition that have any of the $GN_j$ in their definition.

An interpretation, $\mathcal{I}$, consists of a domain, $\Delta^{\mathcal{I}}$, and a mapping, $\cdot^{\mathcal{I}}$, from concept names, role names, attribute names, and individual names to their extensions in the interpretation. The interpretation function is extended to all concepts, roles, attributes, individuals, and assertions as defined below.

All domains contain the rationals and strings over some alphabet of size at least 2, this is called the concrete part of the domain, $\Delta_c^{\mathcal{I}}$. The rest of the domain, $\Delta_a^{\mathcal{I}}$, is called the abstract part of the domain.

The extension of concepts are subsets of $\Delta^{\mathcal{I}}$. The extension of roles are set-valued functions from $\Delta_a^{\mathcal{I}}$ to $\Delta^{\mathcal{I}}$. The extensions of attributes are single-valued, partial functions from $\Delta_a^{\mathcal{I}}$ to $\Delta^{\mathcal{I}}$. (The extension of roles and attributes will sometimes also be treated as the equivalent subset of $\Delta_a^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The extension of attributes are also sometimes treated as set-valued functions.) The extensions of individual names are elements of the abstract part of the domain. The extension of a distinct individual name is different from the extension of all other distinct individual names. The extension of a number or a string is the appropriate rational or string. The extension of assertions is either true or false.

The extension of the concept-, role-, and attribute-forming operators is as in the DFKI proposal [?], extended in the obvious way. See Tables ??, ??, and ?? for details. The extension of the **satisfies** constructs is unconstrained (within $\Delta^{\mathcal{I}}$ or $\Delta_a^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, of course).

A non-empty set of interpretations is a model-set for a simple knowledge base if each statement in the simple knowledge base is is true in the set. A simple knowledge base (any knowledge base) is inconsistent if it (its simple version) has no model-sets.

A non-rule, non-closure statement is true in a non-empty set of interpretations if it is true in each interpretation in the set. Conditions for the truth of several types of statements are given in Table ??. The statement (**disjoint** $\mathsf{CN}_1$ $\mathsf{CN}_2$) is true in an interpretation if the extensions of $\mathsf{CN}_1$ and $\mathsf{CN}_2$ are disjoint. Definitions of individuals only serve to distinguish anonymous and distinct individual names.

A non-empty set of interpretations makes (**define-rule** $\mathsf{N}$ $\mathsf{C}_1$ $\mathsf{C}_2$) true for a simple knowledge base if for each individual name, $\mathsf{IN}$, in the simple knowledge base, if $\mathsf{IN}^{\mathcal{I}} \in \mathsf{C}_1{}^{\mathcal{I}}$ in each interpretation, $\mathcal{I}$, in the set, then $\mathsf{IN}^{\mathcal{I}} \in \mathsf{C}_2{}^{\mathcal{I}}$ in each interpretation in the set.

A subsumption relationship, $\mathsf{C}_1 \Longrightarrow \mathsf{C}_2$ ($\mathsf{R}_1 \Longrightarrow \mathsf{R}_2$), follows from a knowledge base if $\mathsf{C}_1{}^{\mathcal{I}} \subseteq \mathsf{C}_2{}^{\mathcal{I}}$ ($\mathsf{R}_1{}^{\mathcal{I}} \subseteq \mathsf{R}_2{}^{\mathcal{I}}$) in each interpretation of each model-set of the simple knowledge base version of the concept, role, attribute, disjointness, and individual definitions (i.e., no rules or assertions or closures) in the original knowledge base. An instance relationship, $\mathsf{IN} \in \mathsf{C}$, follows from a knowledge base if $\mathsf{IN}^{\mathcal{I}} \in \mathsf{C}^{\mathcal{I}}$; a role relationship, $\langle \mathsf{IN}, \mathsf{I} \rangle \in \mathsf{R}$, follows if $\langle \mathsf{IN}^{\mathcal{I}}, \mathsf{I}^{\mathcal{I}} \rangle \in \mathsf{R}^{\mathcal{I}}$; and an individual equality, $\mathsf{IN}_1 = \mathsf{IN}_2$, follows if $\mathsf{IN}_1{}^{\mathcal{I}} = \mathsf{IN}_2{}^{\mathcal{I}}$; all in each interpretation of each model-set of the simple knowledge base version of the original knowledge base.

## 4 Queries, Retrievals, and Validations

The input language of the specification also contains inquiries about the knowledge base that is being constructed. The input language is thus a sequence of statements, queries (see Table ??), retrievals (see Table ??), and validations (see Table ??).

A query follows from a knowledge base, and returns something other than the symbol NIL, if its meaning in Table ?? follows from the knowledge base before the query. Otherwise, the query is false, and returns the symbol NIL. However, if a concept or role in a subsumption query is incoherent[2] then the result of the query is unspecified. Note that determining if a query follows from a knowledge base is not decidable, nor even recursively enumerable. Therefore, no implementation can possibly be complete.

Retrievals return sets (as lists) of concept names, role names, individual names, and concrete individuals.

The retrieval (**concept-descendants** C) ((**concept-ancestors** C)) returns the set of concept names, CN, that are defined in the KB, and for which $\mathsf{CN} \Longrightarrow \mathsf{C}$ ($\mathsf{C} \Longrightarrow \mathsf{CN}$) follows from the KB but $\mathsf{C} \Longrightarrow \mathsf{CN}$ ($\mathsf{CN} \Longrightarrow \mathsf{C}$) does not. The retrieval (**concept-offspring** C) ((**concept-parents** C)) returns the set of maximal (minimal), under the subsumption relationship in the KB, elements of the result of (**concept-descendants** C) ((**concept-ancestors** C)). The retrieval (**concept-instances** C) returns the set of individual names, IN, that are defined in the KB, and for which $\mathsf{IN} \in \mathsf{C}$ follows from the KB. The retrieval (**concept-direct-instances** C) returns the subset of the result of (**concept-instances** C) that are not instances of any member of the result of (**concept-descendants** C).

Role retrievals are defined analogously.

The retrieval (**individual-types** IN) returns the set of concept names, CN, that are defined in the KB and for which $\mathsf{IN} \in \mathsf{CN}$ follows from the KB. The retrieval (**individual-direct-types** IN) returns the set of minimal, under the subsumption relationship in the KB, elements of the result of (**individual-types** IN).

[2]An incoherent concept or role has empty extension in all model sets.

| **Query** | **Meaning** |
|---|---|
| (concept-subsumes? $C_1$ $C_2$) | $C_1 \Longrightarrow C_2$ |
| (role-subsumes? $R_1$ $R_2$) | $R_1 \Longrightarrow R_2$ |
| (individual-instance? IN C) | $IN \in C$ |
| (individual-related? IN I R) | $\langle IN, I \rangle \in R$ |
| (individual-equal? $IN_1$ $IN_2$) | $IN_1 = IN_2$ |
| (individual-not-equal? $IN_1$ $IN_2$) | $\neg(IN_1 = IN_2)$ |

Table 6: Query Syntax and Semantics

(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: Retrieval Syntax

(validate-true QQ)
(validate-not-true QQ)
(validate-set RR $N_1$ ... $N_n$)

Table 8: Validation Syntax

The retrieval (individual-fillers IN R) returns the set of individual names, $\mathsf{IN}_1$, that are defined in the KB and for which $\langle \mathsf{IN}, \mathsf{IN}_1 \rangle \in \mathsf{R}$ follows from the KB, unioned with the set of concrete individuals, $\mathsf{I}_c$, for which $\langle \mathsf{IN}, \mathsf{I}_c \rangle \in \mathsf{R}$ follows from the KB, unless this latter set is infinite, in which case the retrieval is undefined.

Validations simply check to see if the query or retrieval returned the expected result. If so, the validation is true; otherwise, the validation is false, and may print a message.

## 5 Compliance

Conforming implementations must parse the entire syntax. If a conforming implementation cannot internally represent a particular statement, it must replace it some syntactically-close representable statement and issue a warning.

For simple knowledge bases, conforming implementations must be sound for queries. Retrievals must be correct with respect to a definition that replaces semantic entailment by the (incomplete) subsumption query in the conforming implementation.

Core knowledge bases are defined as knowledge bases containing only the following sorts of statements:

```
(define-concept CN cC)
(define-primitive-concept CN cC)
(define-disjoint-primitive-concept CN (GN1 ... GNn) cC)
(define-primitive-attribute AN top)
(define-distinct-individual DIN)
(define-rule XN CN cC)
(state (instance IN cC))
(state (related IN I cR))
```

Here cC is a core concept, which is either a concept name, TOP, BOTTOM, NUMBER, INTEGER, or STRING or formed as follows:

```
(and cC1 ... cCn)
(all cR cC)
(some cR)
```

```
(none cR)
(eq (compose AN_11 ... AN_1n) (compose AN_21 ... AN_2m))
(minimum CI)
(maximum CI)
```

Also, cR is either a role name or an attribute name. (In the core cR has to be an attribute, but the number restriction extension allows multi-valued roles also.)

The core syntax was selected to be easy to perform inferences on. Subsumption inferences are polynomial. Other inferences are similarly easy. The inference for rules is that in the presence of the rule (define-rule XN CN C), if (instance IN CN) can be derived, then (instance IN C) can also.

Conforming implementations must accept all consistent core knowledge bases that also have no incoherent concept definitions. The actions of conforming implementations on inconsistent knowledge bases or knowledge bases with incoherent concept definitions are unspecified. It is recommended that an error be signaled or the knowledge base be reverted to a consistent state (or both) as soon as an inconsistency is detected. If an incoherent concept (or role) definition is detected, an error may be signalled and the knowledge base be reverted, or a warning produced.

Conforming implementations must perform complete reasoning on core knowledge bases. Note that because subsumption is unspecified for incoherent concepts or roles conformining implementations do not have to perform complete subsumption on such concepts or roles.

## 5.1 Extensions

If an implementation is complete with respect to subsumption of incoherent concepts then it satisfies the "incoherent-subsumption" extension.

If an implementation is complete on the core plus the statements:

```
(define-primitive-role RN top)
(close-role IN cR)
(close-role-fillers IN cR)
```

with concepts extended to include (at-least n RN), (at-most n RN), and (exactly n RN) then it satisfies the "number-restriction" extension.

## References

[1] Franz Baader, Hans-Jürgen Bürckert, Jochen Heinsohn, Bernhard Hollunder, Jürgen Müller, Bernhard Nebel, Werner Nutt, and Hans-Jürgen Profitlich. Terminological knowledge representation: A proposal for a terminological logic. A DFKI note, June 1991.

[2] Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, Andrea Schaerf, and Werner Nutt. Adding epistemic operators to concept languages. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann, October 1992.

## A Test Suite

This is a simple test suite for compliance with the specification.

```
;; The following macro calls test the parsing.
(define-concept c1 top)
(define-concept c c1)
(define-role relative top)
(define-primitive-role sibling (and top relative))
(define-primitive-role brother (or (and sibling)))
(define-primitive-role sister (and (or sibling)))
(define-attribute age top)
(define-primitive-attribute name (or relative))
;; defaults to sister as parent:
(define-primitive-role sister2 (and sister brother))
;; Defaults to simple role:
(define-primitive-role relative3 (or sister brother))
(define-primitive-role r1 top)
(define-primitive-role r2 top)
(define-primitive-role r3 top)
(define-primitive-role r top)
(define-primitive-attribute a1 top)
(define-primitive-attribute a2 top)
(define-primitive-attribute a top)
```

```
(define-disjoint-primitive-concept animal type top)
(define-disjoint-primitive-concept animal3 type top)
(define-primitive-concept animal2 top)
(define-concept c2a bottom)
(define-concept c2 (all (and relative) c1))
(define-concept c3 (all age (or number)))
(define-concept c4 (all age (and integer)))
(define-concept c5 (all name string))
(define-concept c6 c5)

(define-concept c7 (and (or c1 c2)))
(define-concept c8 (or c1))
(define-concept c9 (or c1 c2))
(define-concept c10 (or (and (some r1) (some r2))))
(define-concept c11 (not top))
(define-concept c12 (not c1))
(define-concept c13 (all r top))
(define-concept c14 (all (not r) top))
(define-concept c15 (some r1))
(define-concept c16 (some (not r1)))
(define-concept c17 (some r top))
(define-concept c18 (some (not r) top))
(define-concept c19 (at-least 2 r1))
(define-concept c20 (at-least 2 (not r1)))
(define-concept c21 (at-least 2 r1 C1))
(define-concept c22 (at-least 2 (not r1) C1))
(define-concept c23 (at-most 2 r1))
(define-concept c24 (at-most 2 (not r1)))
(define-concept c25 (at-most 2 r1 C1))
(define-concept c26 (at-most 2 (not r1) C1))
(define-concept c27 (exactly 2 r1))
(define-concept c28 (exactly 2 (not r1)))
(define-concept c29 (exactly 2 r1 C1))
(define-concept c30 (exactly 2 (not r1) C1))
(define-concept c31 (defined r))
(define-concept c32 (defined (not r)))
```

```
(define-concept c33 (undefined r))
(define-concept c34 (undefined (not r)))

(define-concept c35 (equal (compose a1) (compose a2 a1)))
(define-concept c36 (equal (compose a1) (compose a2 r3)))
(define-concept c37 (equal a1 a2))
(define-concept c38 (equal (range C) a2))
(define-concept c39 (not-equal a1 a2))
(define-concept c40 (subset a1 a2))
(define-concept c41 (fills r i1 i2))
(define-concept c42 (fills r 5 6))
(define-concept c43 (fills (and (or r)) 5 6))
(define-concept c44 (fills (range C) i1 i2))
(define-concept c45 (fills-only r i1 i2))
(define-concept c46 (fills-only r 5 6))

(define-concept c47 (fills-only (range C) i1 i2))
(define-concept c48 (in a1 C1))
(define-concept c49 (in (and A) C))
(define-concept c50 (in (not A) C))
(define-concept c51 (in A (some r)))
(define-concept c52 (is A I))
(define-concept c53 (is (and A) I))
(define-concept c54 (is (not A) I))
(define-concept c55 (set i1 i2 i3))
(define-concept c55a (set 4 5))
(define-concept c56 (minimum 5))
(define-concept c57 (maximum 5))
(define-concept c58 (satisfies integerp))

(define-distinct-individual mary)
(define-anonymous-individual unnamed-ind)
(define-rule rule1 top (at-least 1 r1))
(close-role-fillers mary (and r2))
(close-role-fillers mary r)
(close-role-fillers mary (or r2 r))
(define-distinct-individual fred)
```

```
(close-role fred r2)
(define-primitive-concept athlete top)
(state (instance mary athlete))
(state (not (instance mary athlete)))
(state (equal mary joe))
(state (or (instance mary athlete)))
(state (or (instance mary athlete) (equal mary joe)))
(define-distinct-individual joe)
(state (related mary joe brother))
(state (related mary 35 age))
(state (and (instance mary athlete)))
(state (and (instance mary athlete) (related mary 35 age)))
(state (and (instance mary athlete) (equal mary joe)))

;; role syntax
(define-concept c59 (all top top))
(define-concept c60 (all bottom top))
(define-concept c61 (all identity top))
(define-concept c62 (all (inverse parent) top))
(define-concept c63 (all (restrict r c) top))
(define-concept c64 (all (compose r1 r2) top))
(define-concept c65 (all (range c1) top))
(define-concept c66 (all (domain c1) top))
(define-concept c67 (all (domain-range c1 c2) top))
(define-concept c68 (all (transitive-closure r1) top))
(define-concept c69 (all (transitive-reflexive-closure r1) top))
(define-concept c70 (all (satisfies foo r1) top))

;; Test the queries
(define-primitive-role r1 top)
(define-primitive-role r2 top)
(define-primitive-role r2-child r2)
(define-primitive-role r2-child2 r2)
(define-primitive-role r2-descendant r2-child)
(define-primitive-role sibling top)
(define-primitive-attribute age top)
(define-primitive-concept athlete top)
```

```
(define-primitive-concept healthy top)
(define-concept healthy-athlete (and athlete healthy))
(define-primitive-concept very-healthy-athlete healthy-athlete)
(define-distinct-individual mary)
(state (and (related mary 35 age)
            (related mary herbie sibling)
            (related mary joe sibling)
            (related mary martin sibling)))

(define-distinct-individual joe)
(state (and (instance joe athlete)
            (related mary joe r1)))
(close-role-fillers mary r1)
(define-distinct-individual joe-healthy)
(state (and (instance joe-healthy healthy-athlete)))

(concept-subsumes? athlete healthy-athlete)
(concept-subsumes? healthy-athlete athlete)
(concept-subsumes?
 (and (at-least 1 r2) (all r1 athlete))
 (and (at-least 2 r2-child) (all r1 healthy-athlete)))
;; one containing an error:
(concept-subsumes?
 (and (at-least 1 top) (all r1 athlete))
 (and (at-least 2 r2-child) (all r1 healthy-athlete)))

(role-subsumes? r1 r2-child) ;nil
(role-subsumes? r2 r2-child)
(role-subsumes? r2 r2)
(role-subsumes? top r2)

(individual-instance? mary (all r1 athlete))
(individual-instance? joe athlete)
(individual-instance? joe (all r1 athlete)) ;nil
(individual-instance? joel athlete) ;error

(individual-related? mary joe r1)
```

```
(individual-related? mary joe r2)
(individual-related? mary joel r2)

(individual-equal? mary mary)
(individual-equal? mary joe)
(individual-equal? mary mary2)
(individual-equal? mary mary5)

(individual-not-equal? mary mary)
(individual-not-equal? mary joe)
(individual-not-equal? mary mary2)
(individual-not-equal? mary mary5)

;; Retrieval and Validation Macros

(concept-descendants athlete)
(concept-descendants bottom)
(concept-descendants (and athlete healthy))
(concept-descendants (defined r1))

(concept-offspring athlete)
(concept-offspring bottom)
(concept-offspring (and athlete healthy))
(concept-offspring (defined r1))

(concept-ancestors athlete)
(concept-ancestors bottom)
(concept-ancestors (and athlete healthy))
(concept-ancestors (defined r1))

(concept-parents athlete)
(concept-parents bottom)
(concept-parents (and athlete healthy))
(concept-parents (defined r1))

(concept-instances athlete)
(concept-instances bottom)
```

```
(concept-instances healthy-athlete)
(concept-instances (and athlete healthy))
(concept-instances (defined r1))

(concept-direct-instances athlete)
(concept-direct-instances bottom)
(concept-direct-instances healthy-athlete)
(concept-direct-instances (and athlete healthy))
(concept-direct-instances (defined r1))

(role-descendants r2)
(role-descendants r2-child)
(role-descendants r2-descendant)
(role-descendants (and r2-descendant r1))

(role-offspring r2)
(role-offspring r2-child)
(role-offspring r2-descendant)
(role-offspring (and r2-descendant r1))

(role-ancestors r2)
(role-ancestors r2-child)
(role-ancestors r2-descendant)
(role-ancestors (and r2-descendant r1))

(role-parents r2)
(role-parents r2-child)
(role-parents r2-descendant)
(role-parents (and r2-descendant r1))

(individual-types joe)
(individual-types joe-healthy)
(individual-types mary)
(individual-types joel)

(individual-direct-types joe)
(individual-direct-types joe-healthy)
```

```
(individual-direct-types mary)
(individual-direct-types joel)

(individual-fillers mary r1)
(individual-fillers mary age)
(individual-fillers mary r2)
(individual-fillers joel r2)
(individual-fillers mary r21)
(individual-fillers mary (not r21))

(validate-true (role-subsumes? r1 r2-child)) ;nil
(validate-true (role-subsumes? r2 r2-child))

(validate-not-true (role-subsumes? r1 r2-child)) ;nil
(validate-not-true (role-subsumes? r2 r2-child))

(validate-set (individual-fillers mary sibling) joe martin herbie)
(validate-set (individual-fillers mary sibling) martin herbie)
```

# APPENDIX E

## Management of Knowledge Representation Standards Activities NASA-Ames Contract NCC 2-719

## Final Report covering the period of 6/1/1991 - 5/31/1993

## Principal Investigator

Ramesh S. Patil
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
Phone: (310) 822-1511
ramesh@isi.edu

# Draft Specification of the KQML Agent-Communication Language

Tim Finin
University of Maryland
Baltimore, Maryland

Jay Weber
Enterprise Inegration Technologies
Palo Alto, California

Gio Wiederhold, Michael Geneserath
Stanford University
Palo Alto, California

Richard Fritzson, Donald McKay
Paramax Systems Corp.
Paoli, PA

James McGuire, Richard Pelavin
Lockheed AI Cener
Palo Alto, California

Stuart Shapiro
State University of New York
Buffalo, New York

Chris Beck
University of Toranto
Toranto, Canada

# DRAFT
# Specification of the KQML Agent-Communication Language

*plus example agent policies and architectures*

The DARPA Knowledge Sharing Initiative
External Interfaces Working Group

Tim Finin (co-chair)
University of Maryland

Jay Weber (co-chair)
Enterprise Integration Technologies

Gio Wiederhold (former co-chair)
Stanford University

Michael Genesereth
Stanford University

Richard Fritzson
Donald McKay
Paramax Systems

James McGuire
Richard Pelavin
Lockheed AI Center

Stuart Shapiro
SUNY Buffalo

Chris Beck
University of Toronto

February 3, 1993

**Abstract**

This document is a **draft** of an initial specification for the KQML agent communication language being developed by the external interfaces working group of the DARPA Knowledge Sharing Effort. KQML is intendewd to be a high-level language to be used by knowledge-based system to share knowledge at run time.

Send general comments on this document by email to the mailing list KQML-USERS@ISI.EDU. The current working group co-chairs can be reached as follows: Tim Finin, Computer Science, University of Maryland Baltimore County, Baltimore MD 21228. phone:410-455-3522, email: finin@cs.umbc.edu. Jaw Weber, Enterprise Integration Technologies Corporation, 459 Hamilton Avenue, Suite 100, Palo Alto, CA 94301 (415)617-8002. weber@eitech.com.

# Contents

## Preface

The External Interfaces Working Group is a collection of artificial intelligence and distributed systems researchers interested in software systems of communicating agents.

The group was formed in 1990 as a part of the DARPA Knowledge Sharing Effort, with the charter to develop protocols for the exchange of represented knowledge among autonomous information systems. The principal result of this effort is KQML, the Knowledge Query and Manipulation Language. Other working groups include the Knowledge Interchange Format (KIF) working group, the Ontologies working group, and the Knowledge Representations Systems Standards (KRSS) working group.

The Knowledge Sharing Effort has received some direct funding from DARPA, the NSF and AFOSR for organization and coordination. In addition, many of the members of the External Interfaces Working Group are funded through research contracts from these and other agencies.

The development of KQML has been influenced, in particular, by two prototypical agent-based systems. The first is part of the DARPA/Rome Planning Initiative, and involves wide-area communication among planning, scheduling, resource control, and temporal reasoning programs [reference?]. These programs were written in combinations of LISP, Prolog, and C++, they run on a variety of workstation platforms, and communicate over TCP/IP connections.

The second is the Palo Alto Collaborative Testbed (PACT), which demonstrated collaborative distributed design, validation, and prototyping of an electromechanical device. PACT involves metropolitan-area communication among software, circuit, power, sensor, and mechanical CAD systems [PACT-ref]. These systems were written in combinations of LISP and C/C++, they run on a variety of workstation and personal computer platforms, and communicate using either TCP/IP or SMTP (electronic mail) connections (cf. Appendix ??).

# 1 Introduction

Modern computing systems often involve multiple intergenerating computations/nodes. Distinct, and often autonomous nodes can be viewed as agents performing within the overall system, in response to messages from other nodes. There are several levels at which agent-based systems must agree, at least in their interfaces, in order to successfully interoperate:

**Transport:** how agents send and receive messages;

**Language:** what the individual messages mean;

**Policy:** how agents structure conversations;

**Architecture:** how to connect systems in accordance with constituent protocols.

This document is mostly about the language level. This document specifies the syntactic and semantic fundamentals of the Knowledge Query and Manipulation Language (KQML).

KQML is complementary to work on representation languages for domain content, including the DARPA Knowledge Sharing Initiative's Knowledge Interchange Format (KIF). KQML has also been used to transmit object-oriented data, and a wide range of information can be accumulated. KQML is a language for programs to use to communicate attitudes about information, such as querying, stating, believing, requiring, achieving, subscribing, and offering. KQML is indifferent to the format of the information itself, thus KQML expressions will often contain subexpressions in other so-called "content languages."

KQML is most useful for communication among agent-based programs, in the sense that the programs are autonomous and asynchronous. Autonomy entails that agents may have different and even conflicting agendas; thus the meaning of a KQML message is defined in terms of constraints on the message sender rather than the message receiver. This allows the message receiver to choose a course of action that is compatible with other aspects of its function. Of course, most useful agent architectures strive for maximal cooperation among agents, but just as with human organizations, complete cooperation is not always possible.

KQML is complementary to new approaches to distributed computing, which focus on the transport level. For example, the new and popular Object Request Broker [OMG ORB] specification defines distributed services for interprocess and interplatform messaging, data type translation, and name registration. It does not specify a rich set of message types and their meanings, as does KQML.

A KQML message is called a *performative*, in that the message is intended to perform some action by virtue of being sent. (The term is from speech act theory.) This document defines a substantial number of performatives in terms of what they connote about the sender's knowledge.

However, we recognize that the performatives defined herein are neither necessary nor sufficient for all agent-based applications. Therefore, agents need not support the entire set of defined performatives (indeed, we expect that agents will usually support a small subset), and agents may use performatives that do not appear in this specification. New performatives should be defined precisely, and in the style of this specification.

The performative names in this specification are **reserved**; an application is not KQML-compliant if it uses these performatives in ways that are inconsistent with the definitions in this specification. We encourage implementors to use these reserved performatives when possible, to increase overall interoperability.

The primary dimension of KQML extension is through the definition of new performatives. The definitions of new performatives must explicitly describe all permissible parameters, and when applicable, default values for parameters that do not appear in particular messages. A performative definition may coin new parameter names; however, we encourage the use of the parameter names in this specification when they apply.

Besides KQML, at the language level of interoperation, this document touches on issues at the other three levels. Appendix A describes work-in-progress on KQML APIs that provide a definition of and code for the transport level, Appendix B defines some useful terms for describing messaging policies (e.g., timely responses, pertinent communications), and Appendix C describes the architectures of some existing agent-based systems. Our intent is for these terms and examples to make other agent-based systems easier to characterize and compare for the task of achieving high-level interoperation.

This specification is written in the style of an Internet RFC. That is, the main thread of this document is a dry description of what KQML *is*; comments regarding motivation for particular aspects are relegated to inset NOTEs. Also, the latter portion of this document describes several example systems that use (or in one case, could use) KQML messages.

## 1.1 KQML Transport Assumptions

It is not the intent of this document to standardize a programming interface, much less a system infrastructure, for message transport. Such issues are usually dominated by implementation considerations, including programming language choice, network services and security.

Nevertheless, this document does intend to make prescriptions regarding an agent communication language, and this requires a model of message transport. So for these purposes, we define the following abstraction of the transport level:

- Agents are connected by unidirectional communication links that carry discrete messages;
- these links may have a non-zero message transport delay associated with them;
- when an agent receives a message, it knows from which incoming link the message arrived;
- when an agent sends a message it may direct to which outgoing link the message goes;
- messages to a single destination arrive in the order they were sent;
- message delivery is reliable.

NOTE: The latter property is less useful than it may appear, unless there is a guarantee of *agent reliability* as well. Such a guarantee is a policy issue, and may vary among systems.

This abstraction may be implemented in many ways. For example, the links could be TCP/IP connections over the Internet, which may only actually exist during the transmission of a single message or groups of messages. The links could be email paths used by mail-enabled programs [ServiceMail]. The links could be UNIX IPC connections among processes running on an ether-networked LAN. Or, the links could be high-speed switches in a multiprocessor machine like the Hypercube, accessed via Object Request Broker software [OMG ORB]. Regardless of how communication is actually carried out, KQML assumes that at the level of agents, the communication appears to be point-to-point message passing.

Conversely, higher levels can implement a variety of different communication abstractions. For example, a star architecture (cf. Section ??) where the hub handles `broadcast` (cf. Section ??) messages provides a virtual broadcast communication abstraction. A hierarchical architecture may provide a virtual content-based multicast abstraction (cf. Section ??). The use of the `pipe` message produces a virtual connection-oriented approach to message transport.

The point of this point-to-point message transport abstraction is to provide a simple, uniform model of communication for the outer layers of agent-based programs. This should make agent-based programs and APIs easier to design and build.

## 2 KQML String Syntax

A KQML message is also called a *performative*. A performative is expressed as an ASCII string using the syntax defined by this section. This syntax is a restriction on the ASCII representation of Common Lisp Polish-prefix notation.

> NOTE: We chose the ASCII-string LISP list notation because it is readable by humans, simple for programs to parse (particularly for many knowledge-based programs), and transportable by many inter-application messaging platforms. However, no choice of message syntax will be both convenient and efficient for all messaging APIs; Appendix ?? describes some alternate syntaxes for particular applications.

Unlike Lisp function invocations, parameters in performatives are indexed by keywords and therefore order independent. These keywords, called *parameter names*, must begin with a colon (:) and must precede the corresponding *parameter value*.

> NOTE: Performative parameters are identified by keywords rather than by their position due to a large number of optional parameters to performatives.

Several examples of the syntax appear in Section 5 of this document.

### 2.1 KQML string syntax in BNF

The BNF given in Figure ?? assumes definitions for `<ascii>`, `<alphabetic>`, `<numeric>`, `<double-quote>`, `<backslash>`, and `<whitespace>`. "*" means any number of occurrences, and "-" indicates set difference. Note that `<performative>` is a specialization of `<expression>`.

> NOTE: In length-delimited strings, e.g. "#3"abc", the whole number before the double-quote specifies the length of the string after the double-quote.

```
<performative> ::= (<word> {<whitespace> :<word> <whitespace> <expression>}*)

<expression> ::= <word> | <quotation> | <string> |
                 (<word> {<whitespace> <expression>}*)

<word> ::= <character><character>*

<character> ::= <alphabetic> | <numeric> | <special>

<special> ::= < | > | = | + | - | * | / | & | ^ | ~ | _ |
              @ | $ | % | : | . | ! | ?

<quotation> ::= '<expression> | `<comma-expression>

<comma-expression> ::= <word> | <quotation> | <string> | ,<comma-expression>
                       (<word> {<whitespace> <comma-expression>}*)

<string> ::= "<stringchar>*" | #<digit><digit>*"<ascii>*

<stringchar> ::= \<ascii> | <ascii>-\-<double-quote>
```

Figure 1: KQML string syntax in BNF

## 3 KQML Semantics

The semantic model underlying KQML is a simple, uniform context for agents to view each others' capabilities. Each agent appears, on the outside, as if it manages a knowledge base (KB). That is, communication with the agent is with regard to this KB base, e.g., questions about what a KB contains, statements about what a KB contains, requests to add or delete statements from the KB, or requests to use knowledge in the KB to route messages to appropriate other agents.

The implementation of an agent is not necessarily structured as a knowledge base. The implementation may use a simpler database system, or a program using a special datastructure, as long as wrapper code translates that representation into a knowledge-based abstraction for the benefit of other agents. Thus we say that each agent manages a *virtual* knowledge base (VKB).

When defining performatives, it is useful to classify the statements in a VKB into two categories: beliefs and goals. An agent's beliefs encode information it has about itself and its external environment, including the VKBs of other agents. An agent's goals encode states of its external environment that the agent will act to achieve. Performative definitions make reference to either or both of an agent's goals and beliefs, e.g., that the agent wants another agent to send it a certain class of information. The English-prose performatives in this document make reference to these terms, but this view of the VKB is especially important in the formal semantics of KQML [SEMANTICS].

Agents talk about the contents of theirs and other's VKBs using KQML, but the encoding of statements in VKBs can use a variety of representation languages. That is, the KQML performative `tell` is used to specify that a particular string is contained in an agent's belief store, but the encoding of that string can be a representation language other than KQML.

The only restrictions on such a representation is that it be sentential, so that expressions using that representation can be viewed as entries in a VKB, and that sentences have an encoding as an

| *Keyword* | *Meaning* |
|---|---|
| `:content` | the information about which the performative expresses an attitude |
| `:force` | whether the sender will ever deny the meaning of the performative |
| `:in-reply-to` | the expected label in a reply |
| `:language` | the name of represenation language of the `:content` parameter |
| `:ontology` | the name of the ontology (e.g., set of term definitions) used in the `:content` parameter |
| `:receiver` | the actual receiver of the performative |
| `:reply-with` | whether the sender expects a reply, and if so, a label for the reply |
| `:sender` | the actual sender of the performative |

Table 1: Summary of reserved parameter keywords and their meanings.

ascii string, so that sentences can be embedded in KQML messages. Fortunately, these restrictions appear to hold for the representations of interest to KQML users, including AI languages, database languages, object-oriented representations, and many CAD formats.

## 4 Reserved Performative Parameters

As described in Section 2, performatives take parameters identified by keywords. This section defines the meaning of some common performative parameters, by coining their keywords and describing the meaning of the accompanying values. This will facilitate brevity in the performative definitions of Section 5, since the following parameters are used heavily.

The following parameters are *reserved* in the sense that any performative's use of parameters with those keywords must be consistent with the definitions below. These keywords and information parameter meanings are summarized in Table 1.

> NOTE: The specification of reserved parameter keywords is useful in at least two ways: 1) to mandate some degree of uniformity on the semantics of common parameters, and thereby reduce programmer confusion, and 2) to support some level of understanding, by programs, of performatives with unknown names but with known parameter keywords.

```
:sender <word>
:receiver <word>
```

These parameters convey the actual sender and receiver of a performative, as opposed to the virtual sender and receiver in the `:from` and `:to` parameters of networking performatives (cf. Section 5.8).

```
:reply-with <expression>
:in-reply-to <expression>
```

If the `<expression>` is the word `nil` or this parameter is absent from the performative, then the sender does not expect a reply. If the `<expression>` is the word `t` then the sender expects a reply. Otherwise, the sender expects a reply containing a `:in-reply-to` parameter with a value identical to `<expression>`.

```
:content <expression>
:language <word>
:ontology <word>
```

The `:content` parameter indicates the "direct object" (in the linguistic sense) of the performative. For example, if the performative name is `tell` then the `:content` will be sentence being told. The `<expression>` in the `:content` parameter must be a valid expression in the representation language specified by the `:language` parameter, or KQML if the `:language` parameter does not appear. Furthermore, the constants used in the `expression` must be a subset of those defined by the ontology named by the `:ontology` parameter, or the standard ontology for the representation language if the `:ontology` parameter does not appear.

> NOTE: Both `:language` and `:ontology` are restricted to only take `<word>`s as values, and therefore complex terms, e.g., denoting unions of ontologies, are not allowed. We do believe that it will be important to support a calculus of ontologies and languages, but we feel that its proper place is in performatives that define new KQML names. This way, only those agents that can process extensional performatives are expected to understand such a calculus.

`:force <word>`

If the value of this parameter is the word `permanent`, then the sender guarantees that it will never deny the meaning of the performative. Any other value indicates that the sender may deny the meaning in the future. (This parameter exists to help agents avoid unnecessary truth-maintenance overhead.) The default value is `tentative`.

| *Name* | *Section* | *Meaning* |
|---|---|---|
| `achieve` | 5.4 | S wants R to do make something true of their environment |
| `advertise` | 5.6 | S is particulary-suited to processing a performative |
| `ask-about` | 5.2 | S wants all relevant sentences in R's VKB |
| `ask-all` | 5.2 | S wants all of R's answers to a question |
| `ask-if` | 5.2 | S wants to know if the sentence is in R's VKB |
| `ask-one` | 5.2 | S wants one of R's answers to a question |
| `break` | 5.8 | S wants R to break an established pipe |
| `broadcast` | 5.8 | S wants R to send a performative over all connections |
| `broker-all` | 5.9 | S wants R to collect all responses to a performative |
| `broker-one` | 5.9 | S wants R to get help in responding to a performative |
| `deny` | 5.1 | the embedded performative does not apply to S (anymore) |
| `discard` | 5.5 | S will not want R's remaining responses to a previously-mentioned performative |
| `evaluate` | 5.2 | S wants R to simplify the sentence |
| `forward` | 5.8 | S wants R to route a performative |
| `generator` | 5.5 | same as a `standby` of a `stream-all` |
| `monitor` | 5.7 | S wants updates to R's response to a `stream-all` |
| `next` | 5.5 | S wants R's next response to a previously-mentioned performative |
| `pipe` | 5.8 | S wants R to route all further performatives to a another agent |
| `ready` | 5.5 | S is ready to respond to R's previously-mentioned performative |
| `recommend-all` | 5.9 | S wants all names of agents who can respond to a performative |
| `recommend-one` | 5.9 | S wants the name of an agent who can respond to a performative |
| `recruit-all` | 5.9 | S wants R to get all suitable agents to respond to a performative |
| `recruit-one` | 5.9 | S wants R to get another agent to respond to a performative |
| `register` | 5.8 | S can deliver performatives to some named agent |
| `reply` | 5.2 | communicates an expected reply |
| `rest` | 5.5 | S wants R's remaining responses to a previously-mentioned performative |
| `sorry` | 5.2 | S cannot provide a more informative reply |
| `standby` | 5.5 | S wants R to be ready to respond to a performative |
| `stream-about` | 5.3 | multiple-response version of `ask-about` |
| `stream-all` | 5.3 | multiple-response version of `ask-all` |
| `subscribe` | 5.7 | S wants updates to R's response to a performative |
| `tell` | 5.1 | the sentence in S's VKB |
| `unregister` | 5.8 | a `deny` of a `register` |
| `untell` | 5.1 | the sentence is not in S's VKB |

Table 2: Summary of reserved performatives, for sender S and recipient R.

## 5 Reserved Performative Names

In this section, we define several **reserved** performatives. That is, they are reserved in the sense that if an implementation uses any of the following performative names in a way that is inconsistent with the following performative definitions, then that implementation is not compliant with KQML. The reserved performatives and their meanings are summarized in Table 2.

In this section, we describe performative semantics in English prose. Since English prose is often ambiguous and sometimes self-contradictory, we have developed a framework for formal definition of performatives. A full description appears in a separate paper [Genesereth et al.].

Definitions of new performatives should follow the style of the definitions in this section. That is, a definition should convey the following:

- the performative name;
- all parameters keywords that the performative may contain;
- syntactic categories and semantics for all values of parameters with non-reserved keywords;
- any additional syntactic and semantic constraints for values of parameters with reserved keywords;
- the default values of all absent parameters;
- the semantics, in terms of a statement the sender is making of itself, of the performative name applied to the parameters.

## 5.1 Basic informative performatives

```
tell
     :content <expression>
     :language <word>
     :ontology <word>
     :in-reply-to <expression>
     :force <word>
     :sender <word>
     :receiver <word>
```

Performatives of this type indicate that the `:content` sentence is in the sender's virtual knowledge base (VKB) (cf. Section 3).

```
deny
     :content <performative>
     :language KQML
     :ontology <word>
     :in-reply-to <expression>
     :sender <word>
     :receiver <word>
```

Performatives of this type indicate that the meaning of the embedded `<performative>` is *not* true of the sender. A `deny` of a `deny` cancels out.

```
untell
     :content <expression>
     :language <word>
     :ontology <word>
     :in-reply-to <expression>
     :force <word>
     :sender <word>
     :receiver <word>
```

A performative of this type is equivalent to a `deny` of a `tell`.

NOTE: `untell` weaker than telling the negation of the sentence; the sender may not have the negation in its VKB either.

NOTE: Inclusion of `untell` performative is obviously redundant; in this document, perspecuity takes precedence over minimality.

## 5.2 Basic query performatives

```
evaluate
     :content <expression>
     :language <word>
     :ontology <word>
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

Performatives of this type indicate that the sender would like the recipient to simplify the expression in the `:content` parameter, and reply with the result. (Simplification is a language specific concept, but it should subsume "believed equal".)

```
reply
     :content <expression>
     :language <word>
     :ontology <word>
     :in-reply-to <expression>
     :force <word>
     :sender <word>
     :receiver <word>
```

Performatives of this type indicate that the sender believes that `:content` is an appropriate reply to the query in the `:in-reply-to` message.

```
ask-if
     :content <expression>
     :language <word>
     :ontology <word>
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

A performative of this type is the same as **evaluate**, except that the `:content` must be a *sentence schema* in the `:language`. In other words, the sender wishes to know if the `:content` matches any sentence in the recipient's VKB.

```
ask-about
     :content <expression>
```

```
    :language <word>
    :ontology <word>
    :reply-with <expression>
    :sender <word>
    :receiver <word>
```

A performative of this type is like **ask-if**, except that the reply should be the collection of all sentences in the recipient's VKB that contain a sentence or term that matches the sentence or term schema in the `:content`. Note that the reply `:language` and `:ontology` must include a "collection" construct (e.g., sets, lists, bags, etc.).

```
ask-one
    :content <expression>
    :aspect <expression>
    :language <word>
    :ontology <word>
    :reply-with <expression>
    :sender <word>
    :receiver <word>
```

A performative of this type is like an **ask-if**, except that the `:aspect` parameter describes the form of the desired reply; for some match of the `:content` in the recipient's VKB, the reply will be the `:aspect` with all of its schema variables replaced by the values bound to the corresponding schema variables in `:content`. The value of the `:aspect` parameter defaults to the value of the `:content` parameter. Note that performatives of this type make most sense with languages that define schema variables.

```
ask-all
    :content <expression>
    :aspect <expression>
    :language <word>
    :ontology <word>
    :reply-with <expression>
    :sender <word>
    :receiver <word>
```

A performative of this type is like **ask-one**, except that the reply should be a collection of instantiated aspects corresponding to all matches of the `:content` sentences on the recipient's VKB.

```
sorry
    :in-reply-to <expression>
    :sender <word>
    :receiver <word>
```

A performative of this type indicates that the sender understands, but is not able to provide any (more) response(s) to the message referenced by the `:in-reply-to` parameter. A performative of this type may be used in response to an **evaluate** or **ask-one** query, when no other reply is appropriate. It may also be used as the last response to a multi-response query performative (e.g., the performatives in the next section).

Agent A sends the following performative to agent B:

```
(evaluate :language KIF :ontology motors :reply-with q1
      :content (val (torque motor1) (sim-time 5)))
```

and agent B replies with:

```
(reply :language KIF :ontology motors :in-reply-to q1
      :content (scalar 12 kgf))
```

Figure 2: In this example of basic query performatives, agent A asks agent B a simple query and receives a response via a tell.

## 5.3 Multi-response query performatives

```
stream-about
     :content <expression>
     :language <word>
     :ontology <word>
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

This type is like **ask-about**, except that rather than replying with the collection of matches, the responder should send a series of performatives that when taken together identify the members of that collection.

```
stream-all
     :content <expression>
     :aspect <expression>
     :language <word>
     :ontology <word>
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

This type is like **ask-all**, except that rather than replying with the collection of instantiated aspects, the responder should send a series of performatives that when taken together identify the members of that collection.

## 5.4 Basic effector performatives

```
achieve
     :content <expression>
     :language <word>
     :ontology <word>
     :force <word>
     :sender <word>
     :receiver <word>
```

Agent A sends the following performative to agent B:

```
(stream-about :language KIF :ontology motors :reply-with q1
              :content motor1)
```

and agent B replies with a series of performatives:

```
(tell :language KIF :ontology motors :in-reply-to q1
      :content (= (val (torque motor1) (sim-time 5)) (scalar 12 kgf))
(tell :language KIF :ontology structures :in-reply-to q1
      :content (fastens frame12 motor1))
(sorry :in-reply-to q1)
```

Figure 3: Agent A asks B to tell all it knows about motor1. B replys with a sequenct of tells terminated with a sorry.

Performatives of this type are requests that the recipient try to make the sentence in `:content` true of the system (technically, that the sender wants the recipient to want to make the sentence true of the system).

```
unachieve
     :content <expression>
     :language <word>
     :ontology <word>
     :sender <word>
     :receiver <word>
```

A performative of this type is the same as a `deny` of an `achieve`.

## 5.5 Generator performatives

The following performatives comprise a *generator* mechanism for the delivery of responses to a KQML performative. That is, this mechanism allows an agent to explicitly retrieve responses in a series; this is especially useful when there are a large number of responses, and/or the agent is not able to efficiently buffer incoming responses.

`standby`

Agent A sends the following performative to agent B:

```
(achieve :language KIF :ontology motors :reply-with q1
         :content (= (val (torque motor1) (sim-time 5))
                     (scalar 2 kgf))
```

and after achieving the requested motor torque, agent B might send the following (though it is not expected):

```
(tell :language KIF :ontology motors
      :content (= (val (torque motor1) (sim-time 5))
                  (scalar 2 kgf))
```

Figure 4: Agent A tells B to achieve a state in which the the torque of motor1 is a particular value.

```
    :content <performative>
    :language KQML
    :ontology <word>
    :reply-with <expression>
    :sender <word>
    :receiver <word>
```

This type indicates that the sender wants the recipient to take the would-be response(s) from the performative in `:content`, and announce its readiness to accept requests for the responses.

```
ready
    :reply-with <expression>
    :in-reply-to <expression>
    :sender <word>
    :receiver <word>
```

This type indicates that the sender will answer requests for the responses to the performative contained in some performative with the `:in-reply-to` label. The `:reply-with` parameter is, in function, the returned generator.

```
next
    :in-reply-to <expression>
    :sender <word>
    :receiver <word>
```

This type indicates that the sender wishes to receive the next response from those promised by the performative identified by the `:in-reply-to` parameter.

> NOTE: The **next** performative does not have a `:reply-with` parameter because the `:in-reply-to` parameter of the next response should match the `:reply-with` parameter of the performative embedded in the original **standby** message.

```
rest
    :in-reply-to <expression>
    :sender <word>
    :receiver <word>
```

This type indicates that the sender wishes to receive the remaining responses, in a stream, from those promised by the `ready` performative identified by the `:in-reply-to` parameter.

```
discard
    :in-reply-to <expression>
    :sender <word>
    :receiver <word>
```

This type indicates that the sender will issue no more replies to the `ready` performative identified by the `:in-reply-to` parameter. (This is a courtesy to the owner of the generator, so it can reclaim resources needed to maintain the generator.)

Agent A sends the following performative to agent B:

```
(standby :language KQML :ontology K10 :reply-with g1
         :content (stream-about :language KIF
                                :ontology motors
                                :reply-with q3
                                :content motor1))
```

and agent B replies with:

```
(ready :reply-with 2F0B :in-reply-to g1)
```

then agent A follows with:

```
   (next :in-reply-to 2F0B)
```

to which B replies with:

```
(tell :language KIF :ontology motors :in-reply-to q3
      :content (= (val (torque motor1) (sim-time 5))
                  (scalar 12 kgf))
```

and so on, until A sends:

```
(discard :in-reply-to 2F0B)
```

Figure 5: In this example, agent A asks B to prepare to generate a stream of all of the information it knows about motor1. Agen B replys that it is ready and returns an identifier for A to use in requesting the individual facts. Agent A asks for a number of facts and finally indicates that no more are required.

```
generator
     :content <expression>
     :aspect <expression>
     :language <word>
     :ontology <word>
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

This type is the same as:

```
   (standby
        :content (stream-all :content <expression>
             :aspect <expression>
             :language <word>
             :ontology <word>
             :sender <word>
             :receiver <word>)
        :language KQML
        :reply-with <expression>)
```

## 5.6 Capability-definition performatives

`advertise`

```
      :content <performative>
      :language KQML
      :ontology <word>
      :force <word>
      :sender <word>
      :receiver <word>
```

This type indicates that the sender is particularly suited to process the class of KQML performatives described by the `:content` parameter. If the embedded performative is missing any parameters (defined for the embedded performative), then those parameters may take any otherwise legal values.

## 5.7 Notification performatives

```
subscribe
      :content <performative>
      :ontology <word>
      :language KQML
      :reply-with <expression>
      :force <word>
      :sender <word>
      :receiver <word>
```

This type indicates that the sender wishes the recipient to tell it about future changes to what would be the response(s) to the KQML performative in the `:content` parameter.

```
monitor
      :content <expression>
      :ontology <word>
      :language <word>
      :reply-with <expression>
      :force <word>
      :sender <word>
      :receiver <word>
```

This type is the same as:

```
   (subscribe :content (stream-all :content <expression>
         :reply-with <expression>
         :language <word>
         :ontology <word>
         :sender <word>
         :receiver <word>
         :force <word>)
```

Agent B sends the following performative to agent A:

```
(advertis :language KQML :ontology K10
          :content (subscribe :language KQML
                              :ontology K10
          :content (stream-about :language KIF
                                 :ontology motors
                                 :content motor1)))
```

to which agent B responds with:

```
(subscribe :reply-with s1
          :language KQML :ontology K10
          :content (stream-about :language KIF
                                 :ontology motors
                                 :content motor1))
```

then agent A follows with this stream of performatives over time:

```
(tell :language KIF :ontology motors :in-reply-to s1
      :content (= (val (torque motor1) (sim-time 5))
                  (scalar 12 kgf))
(tell :language KIF :ontology structures :in-reply-to s1
      :content (fastens frame12 motor1))
(untell :language KIF :ontology motors :in-reply-to s1
        :content (= (val (torque motor1) (sim-time 5))
                    (scalar 12 kgf))
(tell :language KIF :ontology motors :in-reply-to s1
      :content (= (val (torque motor1) (sim-time 5))
                  (scalar 13 kgf))
...
```

Figure 6: In this example, agent A announces that it is willing to accept subscriptions from other agents who would like to find out about motor1. Agent B tells A that it would indeed like to receive a stream of information about motor1. A then supplies the stream.

## 5.8 Networking performatives

```
register
    :name <word>
    :sender <word>
    :receiver <word>
```

This type indicates that the sender can deliver performatives to the agent named by the `:name` parameter (this subsumes the case when the sender calls itself by this name).

```
unregister
    :name <word>
    :sender <word>
    :receiver <word>
```

This type is the same as a `deny` of a `register`.

```
forward
    :to <word>
    :from <word>
    :content <performative>
    :language KQML
    :ontology <word>
    :sender <word>
    :receiver <word>
```

This type indicates that the sender wants the `:to` agent to process the performative in the `:content` parameter as if it came from the `:from` agent directly. It is important that the `:to` agent receive the package, not just the performative, or it will think that the performative is from the next-to-last step in the path.

> NOTE: This will normally entail that the response(s) are also wrapped in forward(s), since the responder will want to deliver the response(s) to the requesting agent, and achieving this may involve the use of package or other networking performatives. However, it is possible that agent `A` must use a package to send a performative to `B`, but `B` can send a performative to `A` directly.

> NOTE: Previous versions of KQML defined three levels of KQML syntax – the communication (package) layer, the message layer, and the content layer. The current approach is a proper generalization, since the layers arise from the embedding of performatives.

```
broadcast
    :from <word>
    :content <expression>
    :ontology <word>
    :language <word>
    :sender <word>
    :receiver <word>
```

This type indicates that the sender would like the recipient to route the `broadcast` performative to each of its outgoing connections, unless the recipient has already received a `broadcast` performative with this `:reply-with` (for cycle detection).

```
pipe
     :to <word>
     :from <word>
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

This type indicates that future traffic on this channel should be routed to the `:to` agent, as if `:to` and `:from` were directly connected. Furthermore, the recipient is expected to pass the `pipe` performative toward the `:to` agent. Like `forward`, it is important that the destination receive the `pipe` performative, so that it knows that performatives from the next-to-last agent on the path come from the `:from` agent.

```
break
     :in-reply-to <expression>
     :sender <word>
     :receiver <word>
```

A performative of this type breaks a pipe. The `:in-reply-to` parameter value must match the `:reply-with` value of a previous `pipe` performative. Not only is the recipient of a `break` expected to cease piped routing, but it is also expected to pass the `break` up the pipe. This will have the effect of dismantling the pipe in the opposite direction in which it was built.

## 5.9 Facilitation performatives

```
broker-one
     :content <expression>
     :ontology <word>
     :language KQML
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender wants the recipient to process the embedded performative through the help of a single agent that is particularly suited to processing the embedded performative. (Presumably, such suitability was established using `:advertise` performatives.)

```
broker-all
     :content <expression>
     :ontology <word>
     :language KQML
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

This type is similar to **broker-one** except that the sender wants the recipient to enlist the help of *all* agents particularly suited to processing the embedded performative. The recipient of the **broker-all** replies with a list of all responses.

```
recommend-one
     :content <expression>
     :ontology <word>
     :language KQML
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender wants the recipient to reply with the name of a single agent that is particularly suited to processing the embedded performative.

```
recommend-all
     :content <expression>
     :language KQML
     :ontology <word>
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender wants the recipient to reply with a list of names of agents that are particularly suited to processing the embedded performative.

```
recruit-one
     :from <word>
     :content <expression>
     :language KQML
     :ontology <word>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender wants the recipient to forward the embedded performative to a single agent that is particularly suited to processing the embedded performative. This differs from **broker-one** because the recruited agent will forward its response directly to the original sender.

```
recruit-all
     :from <word>
     :content <expression>
     :ontology <word>
     :language KQML
     :sender <word>
     :receiver <word>
```

This type is similar to **recruit-one** except that the sender wants the recipient to forward the embedded performative to all agents particularly suited to processing the embedded performative. The recruited agents individually forward their responses to the original sender.

# 6 Proposed Performatives

This section documents some proposed performatives which are currently being discussed and/or reviewed. As the group reaches a consensus on these proposed performatives, they will be included in other section or deleted. They are included in this document to give the reader an accurate picture of the evolving specification and to encourage discussion of these proposals.

## 6.1 Database performatives

These proposed performatives, INSERT, DELETE, etc. provide an ability for one agent to request another agent to insert or delete sentences in its VKB.

```
insert
     :content <expression>
     :language <word>
     :ontology <word>
     :reply-with <expression>
     :in-reply-to <expression>
     :force <word>
     :sender <word>
     :receiver <word>
```

The sender requests the receiver to add the `:content` sentence to its VKB. The performative can either fail or succeed. Possible errors and warning conditions are:

- Content duplicates sentence already in VKB.
- Content contradicts sentence already in VKB.
- Sender is not authorized to INSERT content.
- ...

```
delete
     :content <performative>
     :language KQML
     :ontology <word>
     :reply-with <expression>
     :in-reply-to <expression>
     :sender <word>
     :receiver <word>
```

The sender requests the receiver to delete the `:content` sentence from its VKB. The sentence must be ground. The performative can either fail or succeed. Possible errors and warning conditions are:

- Content not ground.
- Content not in VKB.
- Content necessarily true in VKB.

- Sender is not authorized to DELETE content.
- ...

```
delete-one
     :content <performative>
     :aspect <expression>
     :order { first | last | undefined }
     :language KQML
     :ontology <word>
     :reply-with <expression>
     :in-reply-to <expression>
     :sender <word>
     :receiver <word>
```

The sender requests the receiver to delete one sentence from its VKB which matches `:content`. Note that performatives of this type make most sense with languages that define schema variables.

The `:aspect` parameter describes the form of the desired reply; for the match of the deleted `:content` in the recipient's VKB, the reply will be the `:aspect` with all of its schema variables replaced by the values bound to the corresponding schema variables in deleted sentence. The value of the `:aspect` parameter defaults to the value of the `:content` parameter. if the `:aspect` is NIL, then no response will be given for a successful deletion.

The optional `:order` parameter specifies whether the sentence to be deleted should be the first or last one found in the VKB (this will only make sense to some agents (e.g. Prolog based ones)). The default value for the `:order` parameter is `undefined`.

The performative can either fail or succeed. Possible errors and warning conditions are:

- No sentence matching content in VKB.
- Content necessarily true in VKB.
- Sender is not authorized to DELETE content.
- ...

```
delete-all
     :content <performative>
     :aspect <expression>
     :language KQML
     :ontology <word>
     :reply-with <expression>
     :in-reply-to <expression>
     :sender <word>
     :receiver <word>
```

This performative is like `DELETE-ONE`, except that the reply should be a collection of instantiated aspects corresponding to all deleted sentences matching the `:content`.

The performative can either fail or succeed. Possible errors and warning conditions are:

- No sentence matching content in VKB.
- All Content necessarily true in VKB.
- Sender is not authorized to DELETE content.
- ...

## 6.2 Modifications to Sorry, Eos, Error

This proposal redefines SORRY, spliting off an EOS performative to indicate the end of a stream of replys. It also suggests an ERROR performative by which one agent can indicate that another agent's message was in some way mal-formed.

```
sorry :in-reply-to <expression>
      :sender <word>
      :receiver <word>
      :comment <string>
```

A performative of this type indicates that the sender understands, but is not able to provide any (more) response(s) to the message referenced by the `:in-reply-to` parameter. A performative of this type may be used in response to an `evaluate` or `ask-one` query, when no other reply is appropriate. The optional :COMMENT parameter can be used to pass a string which describes the specifics of situation leading to refusal to provide a response or additional responses.

```
eos   :in-reply-to <expression>
      :sender <word>
      :receiver <word>
```

The "End Of Stream" performative indicates that the sequence of responses to an earlier multi-response message (e.g., stream-all) :IN-REPLY-TO has terminated successfully. No more responses will be sent.

```
error :in-reply-to <expression>
      :sender <word>
      :receiver <word>
      :comment <string>
      :code <integer>
```

A performative of this type indicates that the sender can not understand or considers to be illegal the message referenced by the `:in-reply-to` parameter. The :COMMENT parameter can be used to return a string further describing how the sender considers the message to be ill formed.

## References

[1] Genesereth, M. R., Fikes, R. E., et al.: "Knowledge Interchange Format Version 3.0 Reference Manual", Logic-92-1, Stanford University Logic Group, 1991.

[2] Genesereth, M. R.: "An Agent-Based Approach to Software Interoperation", Logic-91-6, Stanford University Logic Group, 1991.

## A Example Agent Policies

Agent-based software needs more than just a language for agents to describe their belief and wants. Agents need motivation for performing these communicative acts in terms of expectations about a helpful response. The shared expectations about message-passing behavior, e.g., helpfulness, responsiveness, commitment, etc., comprise the agents' protocols.

There is no single collection of protocols necessary for agenthood. The protocols of a particular system should be optimized for the constituent programs and the task at hand. In this specification, we merely list several protocols that may be useful in many applications. Other protocols, say for skepticism, bidding, reimbursement, and security, should be defined in this manner.

**honesty** a message's KQML semantics apply to the sender.

**gullibility** agents adopt the beliefs of others that are consistent with their own.

**helpfulness** agents adopt the goals of others that are consistent with their own

**responsiveness** agents will eventually respond to every received performative for which a response is expected

> NOTE: this protocol folds in two important constraints: that an agent will eventually process every performative, and that it will generate some sort of response whenever responses are expected. The purpose of the latter constraint is to force a response like "sorry" to performatives that just happen to not produce any other responses. Of course, the meaning of this is totally wrapped-up in the word "expected"; the intent is that response(s) are expected from a performative like "ask", but not "tell". "advertise" is trickier, but even though responses are possible, or even commonplace, they are not "expected".

**empathy** agents have a built-in way of determining what performatives are needed by others (i.e., without needing an explicit performative to which to respond)

**pertinence** agents will not send performatives that they believe will not benefit others

**identity** agents will never register a networking name that is identical with the name of another agent on the same network

Figure 7: The ABSE federation architecture

## B Example Agent Architectures and Implementations

### B.1 Content-based routing architecture (ala DRPI)

*(to be completed by a representative of DRPI)*

applications talk to router interface libraries, communicating content, however they want. We can call it KQML if we want, but it is language- specific intraprocess communication so no need to overregulate it. Router interface libraries communicate with router agents using advertise/publish/subscribe (so-called declarations). Routers talk to each other using advertise/publish/subscribe, maybe broadcast, for the exchange of declarations, and using packages for delivery to end-agents.

### B.2 Agent-Based System Engineering (ABSE)

The ABSE project is a collaboration between the Stanford University Logic Group and Hewlett-Packard Palo Alto Research Laboratories. The ABSE architecture is a network of application agents (referred to simply as *agents*, below) connected through *facilitator* agents.

Agents and facilitators are linked together in what is often called a *federation architecture*. Figure **??** illustrates this architecture for the simple case in which there are just three machines, one with three agents and two with two agents apiece. As suggested by the diagram, agents do not communicate directly with each other. Instead, they communicate only with their local facilitators, and facilitators communicate with each other. In effect, the agents form a "federation" in which they surrender their communication autonomy to the facilitators; hence, the name of the architecture.

Messages from agents to facilitators may be directed or undirected. Undirected messages have content but no addresses. It is the responsibility of the facilitators to route such messages to agents able to handle them. In performing this task, facilitators can go beyond simple pattern match – they can translate messages, they can decompose problems into subproblems, and they can schedule the work on those subproblems. In some cases, this can be done interpretively (with messages going through the facilitator); in other cases, it can be done in one-shot fashion (with the facilitator setting up specialized links between individual agents and then stepping out of the

Figure 8: The PACT architecture

picture).

To accomplish the above. facilitators handle the reserved KQML performatives **forward**, **broker-one**, **broker-all**, and **register**. In addition, facilitators exploit the definitions of the reserved parameters **:content**, **:language**, and **:ontology** to perform representation language translation. When taken together, agents handle a wide variety of reserved KQML performatives, including **evaluate**, **ask-about**, **reply**, **deny**, and **generator**.

## B.3 Palo Alto Collaborative Testbed

The PACT experiments show how pre-existing engineering software systems can be combined to constitute a distributed system of integrated design information and services. The PACT architecture encapsulates each component system with an *information* agent, which serves to bridge the idiosynchrosies of access to that system's knowledge and abilities (see Figure 8). Information agents use KQML as their agent communication language, with KIF as the exclusive representation language. Information agents are connected as needed, in part through an ABSE post-office agent (cf. Section ??).

The experiments involved four geographically distributed engineering teams, collaborating on scenarios of design, fabrication, and redesign of a robotic manipulator. Each of the four design environments in PACT was used to model a different aspect of the manipulator (controller software, rigid body dynamics, encoder circuitry, sensors, and power system) and to reason about it from the standpoint of a different engineering discipline. Collaborative design tasks were performed including dynamics model exchange between the controls agent and dynamics agent, fine-grained cooperative distributed simulation exercising each aspect supported by the four tools, and finally design modifications suggested by the simulation. Each team was supported by its own computational environment linked via the PACT framework [Singh and Genesereth][Genesereth92] over the Internet.

The challenge in PACT was to take four existing systems, each already a specialized framework, and to integrate them via a flexible, higher-level framework. Framework building requires commitments from each party desiring participation in the shared environment to establish interface agreements and protocols of interaction. To drive the experiments with concrete goals, scenarios of interoperation among the various concurrent engineering tools, initially thwarted by tool isolation, were proposed. Next a series of interpersonal interactions were conducted among the developers of the various tools to identify the necessary information that bridged tool perspectives and enabled the execution of the driving design scenarios. Once the types of enabling knowledge had been identified (components, connectivity, attribute features, time varying values, equational functional models, etc.), agreements were reached on the form of the shareable knowledge. As a result of these interactions, an implicit ontology was created reflecting offline agreements. Over the course of the PACT experiments, the ontologies were explicitly encoded in KIF and Ontolingua Gruber92].After the form and semantics of the knowledge content had been agreed upon, an KQML-like language Agent Communication Language was specified to allow expressions of attitude toward knowledge content such as belief, disbelief, and interest.

Each tool has been wrapped up as an information agent available as a service to other agents. Tool-specific wrappers were constructed for each tool to translate into and out of the shared ontology and to manage the tool's application programmer interface for reacting to requests and updates expressed within the KQML-like Agent Communication Language.

PACT employs a rich suite of performatives, indicating the diversity of the PACT architecture.

**Networking** The PACT framework provides an infrastructure postal service to allow agents to delegate all message delivery responsibilities. To utilize the postal service, individual agents employ the `register` performative to make the postal service aware of its presence. The postal service is capable of handling `forward` messages addressed to any registered agent. Other message traffic is point-to-point between agents to reduce the overhead of a centralized bottleneck (e.g. during a distributed simulation).

**Notification** The PACT experiments exercised concurrent engineering design scenarios. Embodied within the concept of concurrent engineering environment is the tenet that all affected parties of recent design changes will be notified of the change so they may assess the impact. Notification is triggered by detection of change in information of interest. The traditional query oriented approach for requesting existing known information does not support notification, since the interest is assumed to expire after the answer is returned. What is needed is a performative whose semantics convey the monitoring nature of a notification request. Consequently, heavy use is made of the `subscribe` performative to convey the conditions triggering a notification. As a simple example of the utility of `subscribe` within PACT, one agent (NextCut) posts a persistent interest in the type of motor applying torque to the manipulator arms. This way if the motor changes, the consequences of the change in the motor's features can be evaluated.

**Facilitation** PACT is currently building sophistication into the infrastructure to provide mechanisms for locating registered agents with capabilities suited to fulfilling specific information interests. This way an agent with an information need would allow the infrastructure to broker the service request to agents who have stated capabilities matching the request. To enable this process, service provider agents would be forced to advertise their capabilities via the `advertise` performative. Followup to these advertisements occurs using the `broker-one` and `recommend-one` performatives.

**Generator, Multi-response** To provide flexibility on the packaging of transmitted knowledge and support a local tool's paradigm, provide a variety of mechanisms to specify the form of interesting information. (For architectures which cannot handle asynchrony, provide mechanism to get all answers back at once, allow asynchronous incremental transmission for forward-chaining agents, or support generators).

## B.4 Information bus architecture (ala TIB)

*(to be completed by Jay Weber)*

based on publish & subscribe, advertising of label names (with a hierarchical naming scheme) assumed a priori, clients subscribe to these names and servers publish. [Does TIB partition clients and servers? No real reason to, except it does consolidate recipients of subscriptions.]

# C KQML APIs and Alternate Syntaxes

Architectures that use pre-existing message-passing platforms and agent implementations may find it easier and/or more efficient to use an alternative to the list syntax described in this specification.

## C.1 The ABSE Lisp API

## C.2 The DRPI TCP/IP API

## C.3 KQeMaiL

There have been implementations that use e-mail as a transport mechanism between agents.

Is is reasonable to construe the UNIX sendmail process as an agent that processes package performatives.

We have a mapping between package performatives and Internet e-mail messages.

## C.4 CORBA dynamic invocation interface

The Common Object Request Broker Architecture is a new standard for distributed object-oriented processes that has broad support from software/hardware vendors, standards organizations, and potential users. The CORBA is a reasonable and effective platform for agent-oriented software as well.

There is a simple mapping from the list syntax of this spec to the CORBA dynamic invocation interface.

## D Future Work

This section notes some of the capabilities which are recognized as being needed or desired in KQML and some thoughts on how they might be realized. Any material here is highly speculative. It is included to provide the reader with an accurate vision of the complete language.