Architectures for intelligent systems

by J. F. Sowa

People communicate with each other in sentences that incorporate two kinds of information: propositions about some subject, and metalevel *speech acts* that specify how the propositional information is used—as an assertion, a command, a question, or a promise. By means of speech acts, a group of people who have different areas of expertise can cooperate and dynamically reconfigure their social interactions to perform tasks and solve problems that would be difficult or impossible for any single individual. This paper proposes a framework for intelligent systems that consist of a variety of specialized components together with logic-based languages that can express propositions and speech acts about those propositions. The result is a system with a dynamically changing architecture that can be reconfigured in various ways: by a human knowledge engineer who specifies a script of speech acts that determine how the components interact; by a planning component that generates the speech acts to redirect the other components; or by a committee of components, which might include human assistants, whose speech acts serve to redirect one another. The components communicate by sending messages to a Linda-like blackboard, in which components accept messages that are either directed to them or that they consider themselves competent to handle.

In the years since its founding conference in 1956, the field of artificial intelligence (AI) has generated an impressive collection of valuable components, but no comparably successful architecture for assembling them into intelligent systems. As examples, the following list illustrates the range of AI components that were designed and implemented in the 1950s and 1960s:

> Parsers, theorem provers, inference engines, search engines, learning programs, classification tools, statistical tools, neural networks, pattern matchers, problem solvers, planning systems, game-playing programs, question-answering systems, dialog managers, machine-translation systems, knowledge acquisition tools, modeling tools, and robot guidance systems

Over the past 40 years, the performance, reliability, and generality of these components have been vastly improved. Their theoretical foundations are much better understood, and they have found their way into applications that are no longer considered part of AI. Yet despite attempts to integrate the components into general-purpose intelligent systems, the results are disappointing: the commercially successful systems are limited to special-purpose applications, and the more general systems have not progressed beyond the stage of clever demos. Nothing remotely resembling the HAL computer in the movie

2001: A Space Odyssey exists today, and there are no credible designs for building one soon.

The lack of progress in building general-purpose intelligent systems could be explained by several different hypotheses:

1. Simulating human intelligence on a digital computer is impossible.
2. The ideal architecture for true AI has not yet been found.
3. Human intelligence is so flexible that no fixed architecture can do more than simulate a single aspect of what is humanly possible.

Many people have presented strong, but not completely convincing arguments for the first hypothesis.[1,2,3] In the search for an ideal architecture, others have implemented a variety of, at best, partially successful designs. The purpose of this paper is to explore the third hypothesis: propose a flexible modular framework that can be tailored to an open-ended variety of architectures for different kinds of applications. The tailoring could be done either by a human knowledge engineer who uses specialized AI languages or by semiautomated design tools in collaboration with a human editor who has little or no training in AI. Such a system would not be as intelligent as HAL, but it should be valuable for a wide range of important applications.

The idea of a flexible modular framework (FMF) is not new. It is, in fact, the underlying philosophy of the UNIX** operating system and its descendants. That philosophy is characterized by four design principles:

1. There is a small kernel that provides the basic services of resource allocation and process management.
2. There is a large, open-ended collection of highly modular utilities that can be used by themselves or combined with other modules.
3. Glue languages, also called scripting languages, are used for linking modules to form larger modules or complete applications.
4. There is a uniform data representation, based on character strings, that constitutes the storage format of UNIX files and the content transmitted by UNIX pipes.

The first three principles are as valid today as they ever were, but the fourth has been modified to accommodate modules that require data with more structure than linear strings, especially database management systems (DBMSs) and graphical user interfaces (GUIs). UNIX systems implement the DBMS and the GUI as independent modules, but their nonlinear data structures cannot be communicated via pipes. Other operating systems make different compromises: the IBM AS/400* implements the DBMS in the kernel, and the Macintosh** and Microsoft Windows** systems implement the GUI in the kernel.

The LISP language, which was the primary language of AI since the late 1950s, pioneered techniques that entered the mainstream of commercial computing when they were adopted by other languages ranging from PL/I (programming language one) to Java**. For AI systems, LISP served as both an implementation language and a glue language for AI components and complete systems. Unlike the UNIX character strings, the basic data structures of LISP consist of tree-like lists, which can be supplemented with cross links to form arbitrary graphs. During the 1970s and 1980s, the trees and graphs of LISP proved to be rich enough to support the operating systems of the LISP machines with their stunning graphics. Those graphics techniques, which were invented at Xerox PARC (Palo Alto Research Center), have been copied in all modern GUIs, including those of the Macintosh and Windows.

Although LISP was, and still is, a highly advanced programming language, it is not by itself a knowledge representation (KR) language. The Prolog language is a step closer to a KR language. It supports the same kinds of data structures as LISP, but it has a built-in inference engine for the Horn-clause subset of logic, which can be used to express the rules of an expert system or the grammars of natural languages. For many applications, Prolog has been used as a KR language, either directly or with some syntactic sugar to make its notation more palatable. Yet Prolog still has limitations that make it unsuitable as the glue language for intelligent systems: procedural dependencies, a nonstandard treatment of negation, and the limited expressive power of Horn-clause logic. Like LISP, Prolog is better suited to implementing the components of intelligent systems than representing the knowledge they process.

The most promising candidate for a glue language is Elephant 2000, which McCarthy[4] proposed as a design goal for the AI languages of the new millennium. Sentences in the Elephant language include "*requests, questions, offers, acceptances of offers, permissions* as well as *answers to questions* and other assertions of fact. Its outputs also include *promises* and

statements of *commitment* analogous to promises." As an inspiration for Elephant, McCarthy cited the *speech acts* of natural languages,[5,6] but he believed that Elephant sentences should be written in a formally defined version of logic, rather than the much more informal natural languages. UNIX only supports one kind of speech act: a command that invokes some program. The UNIX scripting languages add loops and conditionals, which determine the sequence of commands to execute. Prolog supports two kinds of speech acts: assertions for stating facts and goals for issuing commands or asking questions. Besides those special cases, Elephant provides a framework that can support the full range of speech acts described in Reference 5.

Although the glue language for intelligent systems should be at a higher level than the scripting languages of UNIX, the four design principles of the UNIX philosophy can serve as guidelines. Following are AI generalizations of the four principles:

1. Like the UNIX kernel, an AI kernel must support resource allocation and process management. But unlike UNIX, which invokes a specific module for each command, an AI kernel should have a pattern-directed or associative method for determining when a module should be invoked. In many AI systems, a *blackboard* or *bulletin board* is used to post messages, which any appropriate component can access when it detects a characteristic pattern. The Linda language[7] is an example of an efficient blackboard system that has been widely used for scheduling parallel computations by clusters of computers.

2. Like UNIX, an AI system should have an open-ended collection of modular components, but the components should be traditional AI tools of the kind developed over the past 40 years. New kinds may also be needed, but they could also be invoked by a glue language like Elephant in combination with a Linda-like blackboard. More conventional components, such as a DBMS, GUI, and various networks, could be invoked by the same mechanisms. The Jini** system of Java, for example, uses a version of the Linda operators for invoking components distributed across a network.

3. An AI glue language, as McCarthy emphasized, should be based on a version of logic that is rich enough to include all first-order logic plus metalevels that can talk about the object level and state whatever speech act is intended. Two such languages are conceptual graphs (CGs) and the Knowledge Interchange Format (KIF), which are being standardized as logically equivalent notations for the same model-theoretic foundations. Other versions of logic, which are discussed in the section on expressive power and computational complexity, can be translated to or from subsets of CGs and KIF. For communication with people who are not logicians, those logics can also be translated to or from versions of controlled natural languages (CNLs), which can serve as readable notations for the underlying logic.

4. Instead of the linear character strings stored in files and transmitted by pipes, logic provides a much richer notation that can represent all the data structures needed for a DBMS, GUI, or network protocol. The messages posted to a Linda-like blackboard could include any logical expression, which in extreme cases might represent an arbitrarily large graph or even the conjunction of any or all the data in a DBMS. The metalevel capabilities of logic, which can represent any desired speech act, can state what should, would, could, or must be done with the data.

This paper shows how a framework based on these four principles can support a family of architectures that can easily be tailored for different kinds of applications. The next section discusses three logically equivalent notations for an Elephant-like glue language: controlled natural language for the human interface; conceptual graphs for components that use graph-based algorithms; and KIF for components that use other notations for logic. The section "Using controlled natural languages" surveys the use of CNLs as a front end to AI systems. The section "Graph algorithms" shows how graph algorithms can simplify or clarify the techniques for searching, querying, and theorem proving. The section "Expressive power and computational complexity" discusses techniques for handling the computational complexities in different applications of logic. Finally, the section "A flexible modular framework" discusses the kinds of components needed for a flexible modular framework and how a glue language communicating through a blackboard can be used to combine them, relate them, and drive them.

Notations for logic

McCarthy's Elephant language requires a highly expressive version of logic, but he did not propose any

Table 1 Four types of propositions used in syllogisms and corresponding sentence patterns

Type	Name	Pattern
A	Universal affirmative	Every *A* is *B*.
I	Particular affirmative	Some *A* is *B*.
E	Universal negative	No *A* is *B*.
O	Particular negative	Some *A* is not *B*.

particular notation for it. Although various notations—graphical, linear, or CNL-like—can express equivalent semantic information, the choice of notation can have a major influence on both the human interfaces and the kinds of algorithms used in the computations. This section illustrates three notations for logic: conceptual graphs, KIF, and controlled natural languages. All three of them can express exactly the same semantics in logically equivalent ways, but they have complementary strengths and weaknesses that make them better suited to different kinds of tasks. Any or all of them could be used in messages passed through a Linda-like blackboard.

For his syllogisms, the first version of formal logic, Aristotle defined a highly stylized form of Greek, which became the world's first controlled natural language. During the Middle Ages, Aristotle's sentence patterns were translated to controlled Arabic and controlled Latin, and they became the major form of logic until the twentieth century. Table 1 lists the names of the four types of propositions used in syllogisms and the corresponding sentence patterns that express them.

With letters such as *A* and *B* in the sentence patterns, Aristotle introduced the first known use of variables in history. Each letter represents some category, which the Scholastics called *praedicatum* in Latin and which became *predicate* in English. If necessary, the verb form *is* may be replaced by *are*, *has*, or *have* in order to make grammatical English sentences. Although the patterns may look like English, they are limited to a highly constrained syntax and semantics: each sentence has exactly one quantifier, at most one negation, and a single predicate that is true or false of the individuals indicated by the subject.

Although Aristotle's syllogisms are the oldest version of formal logic, they are still an important subset of logic, which forms the foundation for description logics, such as DAML and OIL. For frame-like inheritance, the major premise is a universal affirmative statement with the connecting verb *is*; the minor premise is a universal or particular affirmative with *is*, *has*, or other verbs. Many constraints for a DBMS or an expert system can be stated as a universal negative statement with any of the verbs. For constraint checking and constraint inheritance, the major premise is the constraint, and the minor premise is a statement in one of the other three patterns.

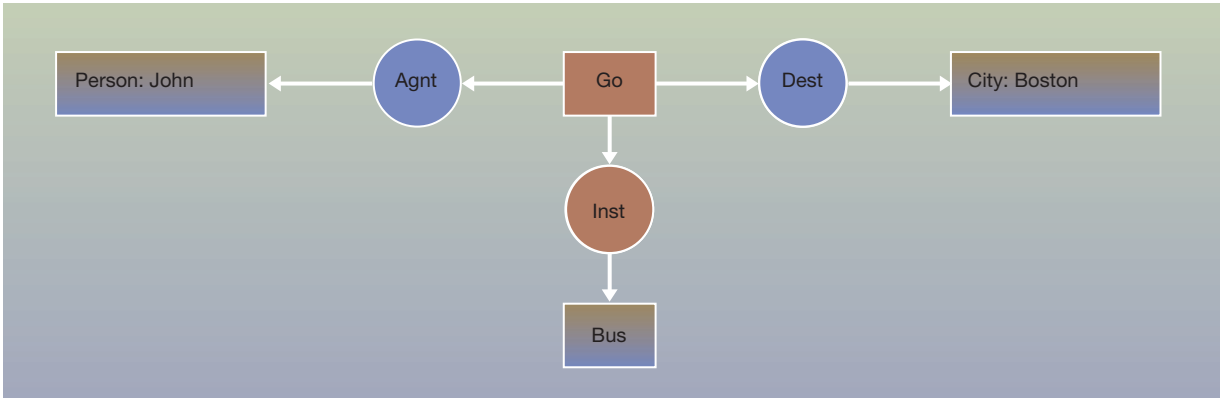
Another important version of logic is the Horn-clause subset, which is widely used for defining expert system rules and SQL (Structural Query Language) views. The basic syntax has an if-then pattern: the if-part of the rule is a conjunction of one or more statements, which may have some negations; the then-part is a conjunction of one or more statements, which may not have negations. Following are two such rules for a library database, written in Attempto Controlled English:[8,9]

```
If a copy of a book is checked out to a borrower
   and a staff member returns the copy
then the copy is available.
If a staff member adds a copy of a book to the library
   and no catalog entry of the book exists
then the staff member creates a catalog entry
          that contains the author name of the book
             and the title of the book
             and the subject area of the book
   and the staff member enters the id of the copy
   and the copy is available.
```

The Attempto system translates these rules to an executable form in Prolog. Anyone who can read English can read controlled English as if it were English, but controlled languages are formal languages that require some training for an author to stay within their limitations. Tools have been developed that can help an author translate full natural language to a CNL.

In the late nineteenth century, three logically equivalent, but structurally very different notations for first-order logic (FOL) were developed. The first was the tree-like *Begriffsschrift* by Frege,[10] and the second was the algebraic notation by Peirce.[11,12] With minor modifications by Peano,[13] Peirce's version became the most commonly used notation for logic during the 20th century. The third notation was Peirce's *existential graphs* of 1897, which he called his *chef d'oeuvre*. KIF is a sorted version of Peirce's algebraic notation, and conceptual graphs are a sorted version

Figure 1 A conceptual graph in the display form



of Peirce's graph notation. For comparison, Figure 1 is a CG representation of the controlled English sentence, John is going to Boston by bus.

The boxes in Figure 1 are called *concepts*, and the circles are called *conceptual relations*. The default quantifier for each concept is the existential, which says that something of the specified type exists; the concept [City: Boston] means that there exists a city named Boston. Each conceptual relation has one or more *arcs*: (Agnt) links a concept that represents an action to the concept that represents its agent; (Inst) links the action to its instrument; and (Dest) links an action that involves motion to its destination. All the relations in Figure 1 are *dyadic*, but in general, a conceptual relation may have any number of arcs.

Although the display form is quite readable, it is not easy to type or to transmit across a network. Therefore, two interchange formats have been developed: the Conceptual Graph Interchange Format (CGIF) maps directly to and from the display form; and the Knowledge Interchange Format (KIF) maps directly to and from the algebraic notation for predicate calculus. Following is the CGIF representation of Figure 1:

```
[Go: *x] [Person: 'John' *y] [City: 'Boston' *z] [Bus: *w]
    (Agnt ?x ?y) (Dest ?x ?z) (Inst ?x ?w)
```

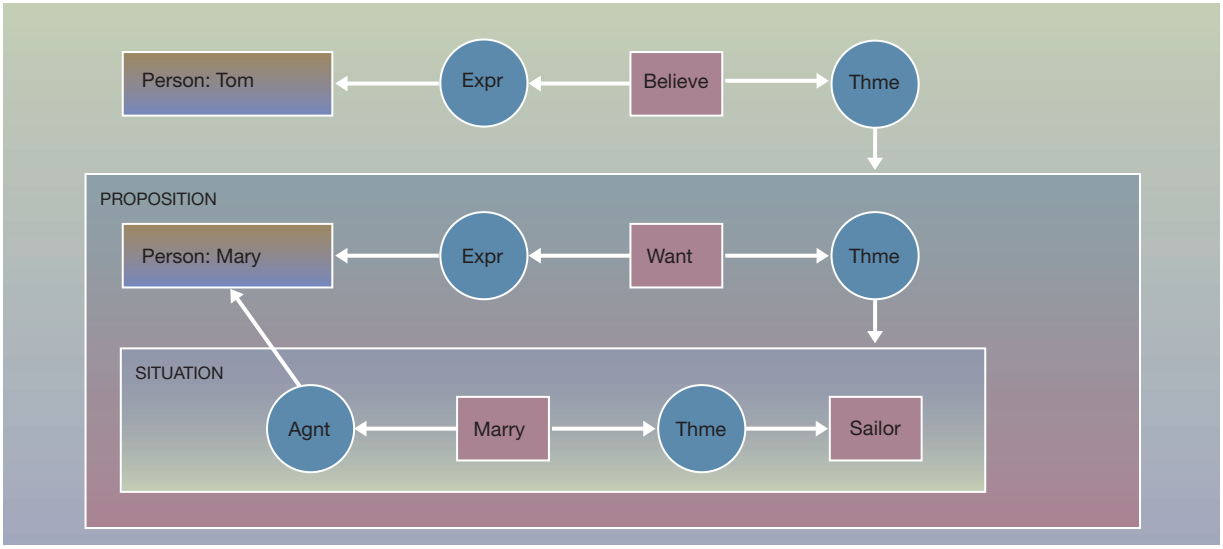
This statement captures every detail of the display form except the two-dimensional layout, which is not semantically relevant. If desired, the layout information could be included as structured comments inside the brackets and parentheses that enclose the nodes of the graph. The connections between concepts and relations, which are shown directly by the arcs of the graph in Figure 1, are shown indirectly by labels, such as ?x and ?y in CGIF. Those labels are translated to variables in KIF, as in the following example:

```
(exists ((?x go) (?y person) (?z city) (?w Bus))
        (and (name ?y John) (name ?z Boston)
             (agnt ?x ?y) (dest ?x ?z) (inst ?x ?w)))
```

KIF notation is used for many theorem provers and inference engines that are based on predicate calculus. The translations between KIF and CGIF preserve the semantics: a mapping from KIF to CGIF and back to KIF might not generate an identical statement, but it will generate a statement that is logically equivalent.

KIF and conceptual graphs can represent the full range of operators and quantifiers of first-order logic, and they have been extended with metalevel features that can be used to define extensions to FOL, including modal logic and higher-order logic. The metalevel features are necessary for representing the speech acts of Elephant 2000, which uses logic to talk about the use of logic. In natural languages, metalevels are marked by a variety of syntactic features that delimit the context of the metalanguage from the context of the object language. The most obvious delimiters are quotation marks, but similar contexts are introduced by verbs that express what some

Figure 2 A conceptual graph with two nested contexts



agent says, thinks, believes, requests, wants, promises, or hopes. As an example, the following English sentence contains two nested levels, which are enclosed in brackets for emphasis:

Tom believes [Mary wants [to marry a sailor]].

This sentence is represented by the CG in Figure 2.

The context of Tom's belief is represented by a concept of type Proposition, which contains a nested CG that states the proposition. The context of Mary's desire is represented by a concept of type Situation, which is described by a proposition that is stated by the nested CG. The (Expr) relation represents the experiencer of a mental state, and the (Thme) relation represents the theme. In general, the theme of a belief or an assertion is a proposition, but the theme of a desire must be something physical, such as a situation. Following is the CGIF equivalent of Figure 2:

```
[Person: *x1 'Tom'] [Believe *x2] (Expr ?x2 ?x1)
    (Thme ?x2 [Proposition:
        [Person: *x3 'Mary'] [Want *x4] (Expr ?x4 ?x3)
            (Thme ?x4 [Situation:
                [Marry *x5] (Agnt ?x5 ?x3)
                (Theme ?x5 [Sailor]) ]) ])
```

And following is the equivalent KIF statement.

```
(exists ((?x1 person) (?x2 believe))
   (and (name ?x1 Tom) (expr ?x2 ?x1)
        (thme ?x2
        (exists ((?x3 person) (?x4 want)
                 (?x8 situation))
            (and (name ?x3 Mary) (expr ?x4 ?x3)
                 (thme ?x4 ?x8)
                 (dscr ?x8 (exists ((?x5 marry)
                                    (?x6 sailor))
                 (and (agnt ?x5 ?x3)
                      (thme ?x5?x6)))))))))
```

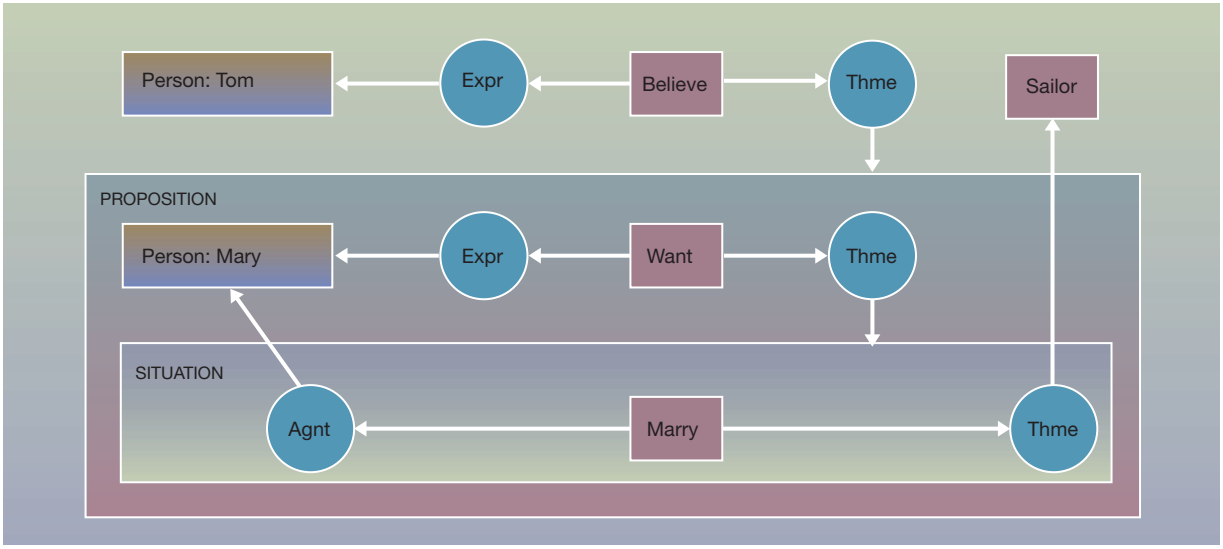
The context boxes delimit the scope of quantifiers and other logical operators. The sailor, whose existential quantifier occurs inside the context of Mary's desire, which itself is nested inside the context of Tom's belief, might not exist in reality. Following is another sentence that makes it clear that the sailor does exist:

There is a sailor that Tom believes Mary wants to marry.

This sentence corresponds to the CG in Figure 3.

The English sentence mentions the sailor before introducing any verb that creates a nested context.

Figure 3 A conceptual graph that asserts the sailor's existence



Therefore, the concept [Sailor] in Figure 3, with its implicit existential quantifier, is moved outside any nested context. In the CGIF and KIF notations, the concept or the quantifier that refers to the sailor would be moved to the front of the statement. Another possibility, represented by the sentence Tom believes there is a sailor that Mary wants to marry, could be represented by moving the concept [Sailor] into the middle context, which represents Tom's belief. In CGIF and KIF, the corresponding concept or quantifier would also be moved to the context of Tom's belief.

As these examples illustrate, conceptual graphs in the display form are more readable than either CGIF or KIF. There are two reasons for the improved readability:

1. *Direct connections*. The arcs of the graph show connections directly without the need for labels or variables. In Figure 1, for example, the four concept boxes map to four distinct labels or variables in CGIF and KIF. To show the links to the relations, CGIF requires 10 occurrences of those labels, and KIF requires 12; furthermore, those occurrences are scattered throughout the linear strings.
2. *Nested enclosures*. As Figures 2 and 3 show, the contexts are shown more clearly with nested enclosures than with nested parentheses or brackets. By using both brackets and parentheses, CGIF has a slight advantage over KIF, but neither notation can compete with the nested boxes of the display form.

Besides human readability, graphs also have theoretical and computational advantages, which are discussed in the section on graph algorithms.

Using controlled natural languages

During the 1980s, the dominant approach to knowledge acquisition required two kinds of highly trained, highly paid professionals. At the top of Figure 4, a knowledge engineer is interviewing a subject matter expert in order to capture her knowledge and encode it in the arcane formats of an AI system. Meanwhile, computational linguists, who were designing natural-language tools, tried to make them translate NL documents into similar encodings without requiring any human intervention. At the bottom of Figure 4, a physician who is examining a patient scribbles some notes on a sheet of paper, which some clerk will later transcribe for the computer. Then the NL tools will attempt to convert those notes to the formats specified by the knowledge engineer.

There are two things wrong with Figure 4: the top row requires far too much human effort, and the bottom row is expected to process unrestricted natural

Figure 4 Twentieth-century approaches to knowledge acquisition and NL processing

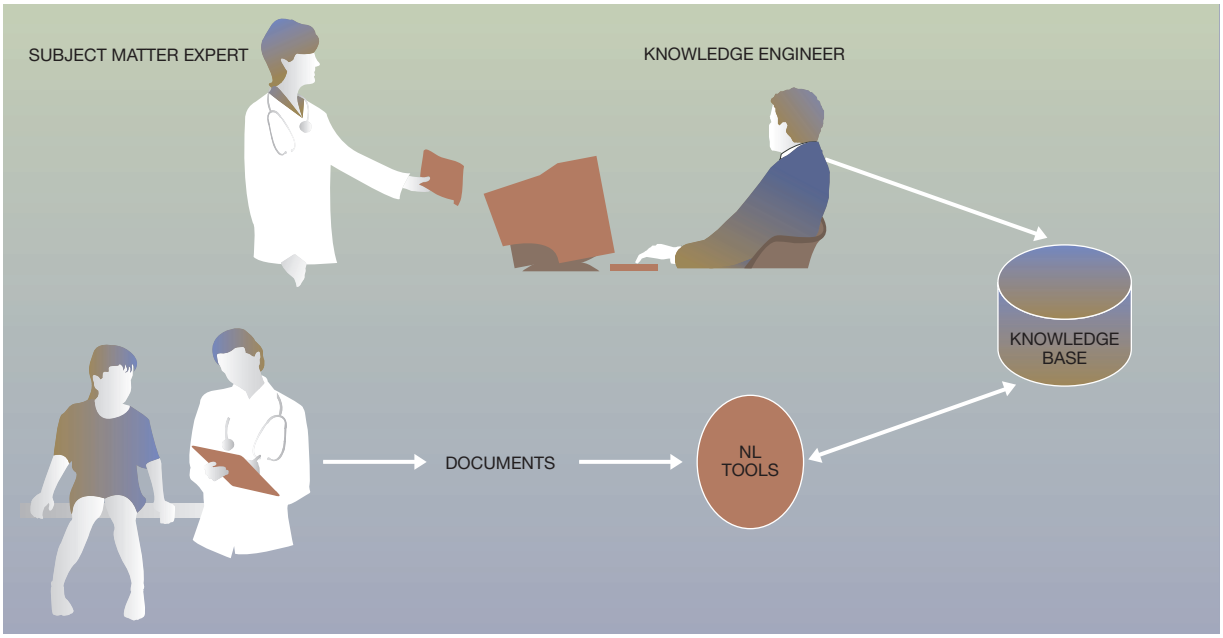
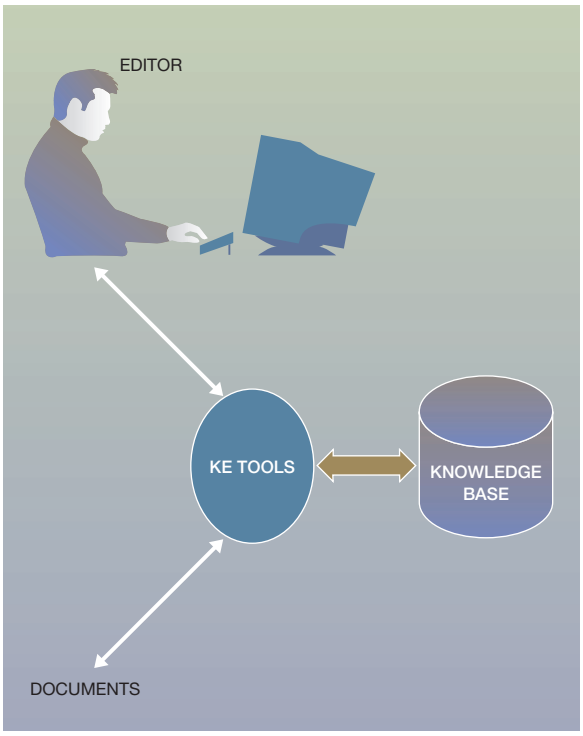


Figure 5 Replacing two experts with one editor



language without any human assistance. To reduce the cost of two high-priced experts, some developers merged the two roles at the top row into one: either the subject matter expert learned knowledge engineering, or the knowledge engineer learned enough about the subject matter to extract knowledge from documents. Yet people with expertise in both fields became even more expensive to find, hire, and train. Figure 5 shows a better alternative: simplify the tools and the training required by the people who use them. Instead of designing complex NL tools that process documents without human intervention, AI researchers developed simpler *knowledge extraction* (KE) tools that can extract knowledge from documents with assistance from just one human editor. Furthermore, the editor communicates with the KE tools in a controlled natural language, which people can read without special training.

The editor in Figure 5 represents various people who at different times might play different roles with respect to the subject matter, the computer system, and the people and activities involved with them. Each of the three people mentioned in Figure 4 has a different kind of expertise. Any of them might use KE tools to edit their knowledge or to write a note, a report, or a book that someone else might edit with

the aid of KE tools. Following are the three kinds of knowledge, the roles of the two experts in Figure 4, and the way that KE tools can help the editor in Figure 5 do the work of both:

- *Semantic knowledge*. The subject matter expert contributes the terminology and background knowledge that is typically recorded in textbooks, research reports, and reference manuals. That knowledge represents the semantics of the subject matter and its links to the natural language vocabulary. An editor in Figure 5 could use the KE tools to extract that knowledge from the documents, or the experts who write the documents could use the KE tools to generate a printable document and a knowledge base at the same time.
- *Episodic knowledge*. The physician at the patient's bedside contributes knowledge of particular instances or episodes in the day-to-day application of the subject matter. Instead of writing notes on a pad of paper, as in Figure 4, such people could enter that information into a computerized tablet or voice recognition system. The KE tools could process the information immediately and respond with a paraphrase in a controlled natural language, which the operational personnel would correct and approve.
- *Patterns of language and logic*. The knowledge engineer in Figure 4 is a specialist in translating unformatted natural language to database tables, if-then rules, and procedural sequences. That kind of knowledge could be codified in a library of patterns or templates represented as conceptual graphs.[14] The KE tools would apply the language patterns to extract information from documents and use the associated logic patterns to reformat it in a CNL. Since the KE process is not foolproof, a human editor must review, correct, and approve the output before it goes into the knowledge base.

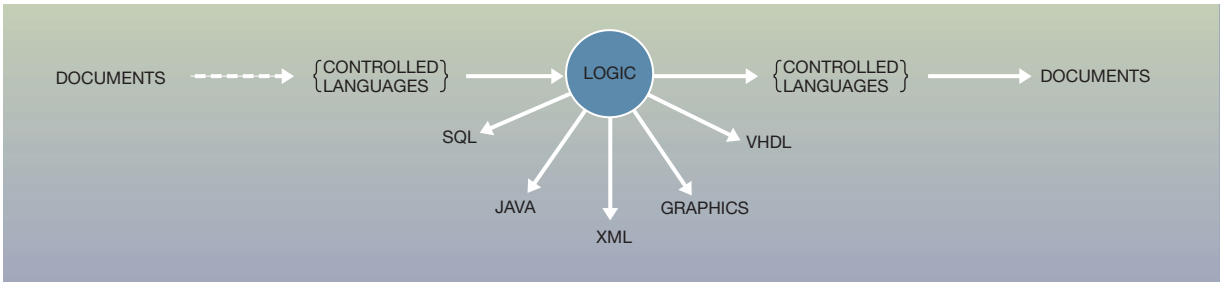
From an editor's point of view, a KE system looks like an intelligent word processor combined with sophisticated tools for searching, classifying, summarizing, and paraphrasing. After the output has been revised by an editor, who might be the original author of the documents, the result can be stored in a knowledge base or be written as an annotation to the documents.

As examples of KE tools, Doug Skuce[15–17] has designed an evolving series of knowledge extraction and document management tools. All input to the knowledge base, whether generated by the KE tools or entered directly by an editor, is represented in a CNL called ClearTalk. The KE tools have the following advantages over the older systems represented by Figure 4:

- *Reduced training for people*. A controlled natural language is a subset of the corresponding natural language. Anyone who can read English can immediately read ClearTalk, and the knowledge editors who write ClearTalk can learn to write it in a few hours. The ClearTalk system itself does most of the training through use: the restrictions shown by menus are enforced by immediate syntactic checks. By consistently using ClearTalk for all its output, the system reinforces the acceptable syntactic patterns.
- *Reduced complexity in the system*. During the knowledge extraction process, the KE tools can ask the editor to resolve ambiguities in the documents, to select relevant passages, and to correct misinterpretations. After the knowledge has been translated to ClearTalk with human assistance, the restricted syntax of ClearTalk eliminates the syntactic ambiguities of ordinary language. The semantic ambiguities are eliminated by the system, which enforces a single definition for every term. As a result, the system can automatically translate ClearTalk to and from logic and various computational languages.
- *Self-documenting systems*. People can read ClearTalk without special training, and a computer system can translate it automatically to a notation for logic, such as CGs or KIF. As a result, the comments and the implementation become identical, and there is never a discrepancy between what the human reads and what the machine is processing.
- *Document annotations*. The double-headed arrow in Figure 5 indicates that the ClearTalk output from the KE tools can also be written as an annotation to the original source documents. Those annotations can serve as humanly readable comments or as input to other ClearTalk systems.

As an example, the students in Skuce's operating systems course used the KE tools to map information from on-line Linux** manuals to a knowledge base for a Linux help facility. The people who wrote the manuals were experts, but the students who edited the knowledge base were novice users of both Linux and the KE tools. As another example, Skuce built a simple knowledge base about animals for his 9-year-old daughter's school project. She and her class could browse the knowledge base on the Web, and they had no difficulty in understanding every fact presented in ClearTalk.

Figure 6 Flow of information from documents to computer representations



Over the past 30 years, many natural-language query systems have been developed that are much easier to use than SQL. Unfortunately, one major stumbling block has prevented them from becoming commercially successful: the amount of effort required to define the vocabulary terms and map them to the appropriate fields of the database is a large fraction of the effort required to design the database itself. However, if appropriate KE tools are used to design the database, the vocabulary needed for the query system can be generated as a by-product of the design process. As an example, the RÉCIT system[18,19] uses KE tools to extract knowledge from medical documents written in English, French, or German and translates the results to a language-independent representation in conceptual graphs. The knowledge extraction process defines the appropriate vocabulary, specifies the database design, and adds new information to the database. The vocabulary generated by the KE process is sufficient for end users to ask questions and get answers in any of the three languages.

Translating an informal diagram to a formal notation of any kind is as difficult as translating informal English specifications to executable programs. But it is much easier to translate a formal representation in any version of logic to controlled natural languages, to various kinds of graphics, and to executable specifications. Walling Cyre and his students have developed KE tools for mapping both the text and the diagrams from patent applications and similar documents to conceptual graphs.[20–22] Then they implemented a scripting language for translating the CGs to circuit diagrams, block diagrams, and other graphic depictions. Their tools can also translate CGs to VHDL, a hardware design language used to specify very high-speed integrated circuits (VHSICs).

Design and specification languages have multiple metalevels. As an example, the Unified Modeling Language (UML) has four levels: the metametalanguage defines the syntax and semantics of the UML notations; the metalanguage defines the general-purpose UML types; a systems analyst defines application types as instances of the UML types; finally, the working data of an application program consists of instances of the application types. To provide a unified view of all these levels, Olivier Gerbé and his colleagues at the DMR Consulting Group implemented design tools that use conceptual graphs as the representation language at every level.[23–27] For his Ph.D. dissertation, Gerbé developed an ontology for using CGs as the metametalanguage for defining CGs themselves.[28] He also applied it to other notations, including UML and the Common KADS system for designing expert systems. Using that theory, Gerbé and his colleagues developed the Method Repository System as an authoring environment for editing, storing, and displaying the methods used by the DMR consultants. Internally, the knowledge base is stored in conceptual graphs, but externally, the graphs can be translated to Web pages in either English or French. About 200 business processes have been modeled in a total of 80 000 CGs. Since DMR is a Canadian company, the language-independent nature of CGs is important because it allows the specifications to be stored in the neutral CG form. Then any manager, systems analyst, or programmer can read them in his or her native language.

No single system discussed in this paper incorporates all the features desired in a KE system, but the critical research has been done, and the remaining work requires more development effort than pure research. Figure 6 shows the flow of information from documents to logic and then to documents or to var-

ious computational representations. The dotted arrow from documents to controlled languages requires human assistance. The solid arrows represent fully automated translations that have been implemented in one or more systems.

For all these tools, the unifying representation language is logic, which could be represented in KIF, CGs, or other notations specialized for various tools. Aristotelian syllogisms together with Horn-clause rules provide sufficient expressive power to specify a Turing machine, and they support efficient computational mechanisms for executing the specifications. For database queries and constraints, statements in full first-order logic can be translated to SQL. All these subsets, however, use the same vocabulary of natural-language terms, which map to the same ontology of concepts and relations. From the user's point of view, the system communicates in a subset of natural language, and the differences between tools appear to be task-related differences rather than differences in language.

Graph algorithms

For many purposes, graphs are a natural representation that is isomorphic to the structure of an application: maps with cities as nodes and highways as arcs; flow diagrams through programs, electrical wiring, and plumbing; the valence bonds between atoms of an organic molecule; the communication links in a computer network; the reference patterns between documents and Web sites on the Internet; and the semantics of natural languages with their complex phrase structures and anaphoric references. When such networks are represented by strings or matrices, the resulting data structures tend to make inefficient use of storage space, execution time, or both. This section surveys five important components of an intelligent system that can benefit from graph-based algorithms:

1. Storage, retrieval, and query
2. Deductive reasoning for logical inference and theorem proving
3. Inductive reasoning for learning new kinds of structures
4. Abductive reasoning for discovering analogies
5. Representing natural-language semantics

In all five of these areas, the direct connectivity of CGs and their nested contexts support algorithms that are simpler and more efficient than algorithms on linear strings and tables. For these reasons, most reasoning systems in AI, even those that use linear notations externally, use tree and graph data structures internally.

During the 1970s, the database field was embroiled in a controversy between the proponents of the new relational DBMS, which stored data in tables, and the proponents of older DBMS systems, which stored data in networks or hierarchies. For many applications, the network and hierarchical systems had better performance, but the relational systems became the universal standard because their logic-based query languages, such as SQL, were far easier to use than the navigational systems, which required a link-by-link traversal of the networks. The battle for network DBMSs was finally lost when one of the staunchest defenders claimed that ease of use was not important because "programmers enjoy a challenge." Today, network systems have come back into vogue as the foundation for object-oriented DBMSs, which represent the connections between objects more directly than the now standard RDBMS. Yet the query languages for OODBMSs (object-oriented DBMSs) require the same kind of link-by-link traversals as the navigational methods of the 1970s. Unlike the logic-based SQL standard, the OODBMS query languages require far more programming effort, which must be specialized to the formats of each vendor.

To support a more natural interface between humans and computers, Sowa[29,30] proposed conceptual graphs as an intermediate language between natural languages and logic-based computer languages. For question-answering systems, a CG derived from a natural language question could be translated to logic-based query languages such as SQL or be matched against the graphs of a network DBMS. In principle, CGs could provide a high-level interface for any DBMS—relational, network, or object-oriented. However, there were two obstacles to using CGs as the universal interface to every DBMS: the natural language processors were not sufficiently robust to generate them, and the algorithms for searching network databases were too slow.

The breakthrough in performance that made a CG database efficient was accomplished by Levinson and Ellis,[31,32] who developed algorithms that could search a lattice of graphs in logarithmic time. Instead of navigating the networks link by link, their systems could take any query graph q and determine where it fit within the lattice. As a result, it would return two pointers: one would point to the lower sublattice of all graphs that are more specialized than q, and the

other would point to the upper sublattice of all graphs that are more generalized than q.

For deduction and theorem proving, Peirce[33] discovered graph-based rules of inference, which are generalizations and simplifications of the rules of natural deduction by Gentzen.[34] The beauty of Peirce's rules is that they make a perfect fit with a system that stores and retrieves graphs in a generalization hierarchy: Peirce's rules are based on the conditions in which any graph p may be replaced by a generalization of p or a specialization of p. Furthermore, the negation of any context reverses the ordering for all graphs in the context: if p is a generalization of q, then $\sim p$ is a specialization of $\sim q$. Esch and Levinson[35,36] presented algorithms for combining Peirce's rules with search and retrieval from a generalization hierarchy, and one of Levinson's students, Stewart,[37] implemented those algorithms in a high-speed theorem prover for first-order logic. Every proposition that was proved, either as a theorem or as an intermediate result, was stored in its appropriate place in the generalization hierarchy together with a pointer to its proof. During a proof, each possible step that could be generated by Peirce's rules was used as a query graph q to determine whether q or any specialization of q had already been proved. If so, the proof was done.

The contexts of conceptual graphs are based on Peirce's logic of existential graphs and his theory of indexicals.

A high-speed search and retrieval mechanism for generalization hierarchies of graphs can also be used as the basis for structural learning algorithms. Unlike neural networks and statistical algorithms, whose learning consists of changing numerical weights, a graph-based algorithm can learn arbitrarily large structures represented as graphs. To demonstrate that principle, Levinson[38] used his search algorithms in a learning program that would learn to play board games, such as chess, by starting with no knowledge about the game other than the ability to make legal moves. His chess program, called Morph, learned chess by playing games with a tutor called Gnu Chess, which was a master-level program, but it could not improve its performance by learning. At the end of each game, Morph was told whether the game was won, lost, or drawn (usually lost, especially in the early stages of learning). Then Morph would estimate the values of all the intermediate positions achieved during the game by backpropagation from the final value (1.0 for a win, 0.5 for a draw, or 0 for a loss), and save the chess positions with their estimated values as graphs in the hierarchy.

When it made a move, Morph would determine all the possible moves, look up the corresponding positions in the hierarchy, find the closest matching positions, and consider their previously estimated values. Morph would then make the move that led to the position with the best estimated value. After playing a few thousand games with its tutor, Morph would have a sufficient database of moves with estimated values to play a decent game of chess.

To find analogies, Majumdar[39] implemented a system called VivoMind, which represents knowledge in dynamic conceptual graphs. What makes the graphs dynamic are algorithms that pass messages along the nodes of a graph. Each node in a CG corresponds to an object that can pass messages to neighboring nodes. The result is an elegant generalization of the marker-passing algorithms originally implemented by Quillian[40] and further developed by Fahlman[41] and Hendler.[42] For finding analogies, VivoMind has proved to be faster than other analogical reasoners on all the usual test cases. For a knowledge base with N relations, most analogy finders take time proportional to N cubed, but VivoMind finds the same analogies in time proportional to $N \log N$.

The contexts of conceptual graphs are based on Peirce's logic of existential graphs and his theory of indexicals. Yet the CG contexts happen to be isomorphic to the similarly nested discourse representation structures (DRSs), which Hans Kamp[43,44] developed for representing and resolving indexicals in natural languages. When Kamp published his first version of DRS, he was not aware of Peirce's graphs. When Sowa[30] published his book on conceptual graphs, he was not aware of Kamp's work. Yet the independently developed theories converged on semantically equivalent representations; therefore, Sowa and Way[45] were able to apply Kamp's techniques to conceptual graphs. Such convergence is common in science; Peirce and Frege, for example, started from very different assumptions and converged on equivalent semantics for FOL, which 120 years later is still the most widely used version of logic. Independently

developed, but convergent theories that stand the test of time are a more reliable basis for standards than the consensus of a committee.

Although graphs are one of the most versatile representations, many good tools use other notations. A framework for intelligent systems should take advantage of different structural properties: some algorithms are more efficient on graphs, and some algorithms are more efficient on strings or tables. The logical equivalence of KIF and CGIF facilitates the mapping from one to the other. Their generality facilitates the integration with other languages that have more restricted expressive power, such as SQL (Structured Query Language), DAML, OIL, RDF (Resource Description Framework), and others. Components based on any of those languages can be integrated with a system that uses KIF and CGs as its primary languages.

Expressive power and computational complexity

The limitations of AI systems have often been blamed on the complexity of the required computations. Various solutions have been proposed, ranging from highly parallel networks that mimic the mechanisms of the human brain, to restricted languages that limit the complexity of the problem definition. A modular architecture could support components that use such strategies for special purposes: neural networks, for example, have been highly successful for pattern recognition, and restricted languages can be highly efficient for specific kinds of problems. For central communications among all components, however, the Elephant language used in the blackboard must be the most expressive, since it must transmit any information that any other component might use or generate. That extreme expressive power raises a question about the complexity of the computations needed to process it.

Computational complexity, however, is not a property of a language, but a property of the problems stated in that language. First-order logic has been criticized as computationally intractable because the proof of an arbitrary FOL theorem may take an exponentially increasing amount of time. That criticism, however, is misleading, since large numbers of problems stated in full FOL are easily solvable. Placing restrictions on the logic or the notation cannot make an intractable problem solvable; they merely make it impossible to state. The expressive power of Elephant does not slow down the communications from one component to another. The components that receive a communication are responsible for determining what they can do with it.

For certain kinds of problems, first-order logic can be the most efficient way to express them and to solve them. A typical example is answering a query in terms of a relational database. The answer to an SQL query that uses the full expressive power of FOL can be evaluated in at most polynomial time, with the exponent of the polynomial equal to the number of quantifiers in the query. If the quantifiers range over an indexed domain, the evaluation can often be done in logarithmic time. Evaluating a constraint against a relational database is just as efficient as evaluating a query; in fact, every constraint can be translated to a corresponding query that asks for all instances in the database that violate the constraint. In commercial SQL systems, queries and constraints with the expressive power of FOL are routinely evaluated with databases containing gigabytes and terabytes of data.

Although the time to solve an intractable problem may be very long, the time to detect the complexity class of a problem can be very short. Callaghan[46] took advantage of syntactic criteria to subdivide the Levinson-Ellis graph hierarchies into several disjoint subhierarchies, each of which is limited to one complexity class. For each subhierarchy, he determined appropriate algorithms for efficiently classifying and searching that hierarchy. To determine the complexity class of any graph, Callaghan computed a *signature* or descriptor to determine its complexity properties. Each graph's descriptor would specify easily computable prerequisites (necessary conditions) that any matching graph must meet. By precomputing the descriptor of a query graph, Callaghan accomplished several goals at once: determining the complexity of the search (tractability or decidability); narrowing the search to a particular class of graphs that have compatible descriptors; or determining whether the query graph lies outside the known complexity classes.

Besides the subhierarchies of graphs supported by the Levinson-Ellis algorithms, Callaghan's approach can accommodate any external subsystem for which a suitable descriptor can be computed by simple syntactic tests. Among those subsystems are the relational databases, which are highly optimized for data stored in tables. In fact, the Levinson-Ellis hierarchies are complementary to an RDBMS: the kinds of data that are most efficient with one are the least efficient with the other. Other important subsystems include the specialized query languages of many ver-

sions of description logics. If a query graph lies outside of any of the known classes, it can be sent to a general first-order theorem prover. As a result, this approach can accept any query expressible in first-order logic, determine its complexity class, and send it to the most efficient subsystem for processing it.

Mapping a smaller logic to a more expressive logic is always possible, but the reverse mapping usually requires some restrictions. To map information from a large, rich knowledge base to a smaller, more efficiently computable one, Peterson, Andersen, and Engel[47] developed a system they called the *knowledge bus*. Their source was the Cyc** knowledge base,[48,49] which contains over 500 000 axioms expressed in full FOL with temporal, higher-order, and nonmonotonic extensions. Their target was a hybrid system that combined a relational database with an inference engine based on the Horn-clause subset of FOL. To map from one to the other, the knowledge bus performs the following transformations:

- *Extracting a subontology*. To extract an ontology for a particular application, the knowledge bus starts with a *seed* consisting of the concept types explicitly mentioned in the application. Then it searches through Cyc to determine which axioms might deduce information about any of the seed types. Finally, it extracts those axioms together with the types and predicates used in them. For the sample application, it extracted approximately 1 percent of the total Cyc knowledge base: 1531 types, 1267 predicates, and 5532 axioms.
- *Separating rules and constraints*. Since the Horn-clause inference engine cannot process arbitrary FOL statements, the knowledge bus separates the axioms into two classes: 4667 Horn-clause rules that are used for inferences, and 875 FOL statements that are used as database constraints. Both the inferencing and the constraint checking can be done efficiently, in at most polynomial time.
- *Restrictions and modifications*. For temporal reasoning, the knowledge bus adds extra arguments for starting and ending times to the Cyc time-dependent predicates. To eliminate the higher-order features, it introduces constants of type Assertion. And to simulate the Cyc nonmonotonic features, it uses a version of negation as failure.

For a particular application, the knowledge bus extracts a small subset of the Cyc knowledge base that can be processed more efficiently by simpler tools. Although some information and some potential inferences are lost, the extracted subset has a well-founded semantics that is guaranteed to be free of contradictions. Furthermore, the resulting subset is more portable: the inference engine can be used as an extension to any relational database, and Engel[50] has developed techniques for mapping the definitions and axioms to Java classes that can be used in Web-based applications.

A flexible modular framework

A framework based on Elephant and Linda would subsume anything that could be done with a more conventional scripting language. Natural languages can specify procedures with a sequence of imperative statements linked by adverbs such as *then* and *next*. A translation of those statements into KIF or CGs would specify the same procedure. But natural languages and their translations into logic could also specify more complex speech acts that could dynamically reconfigure the components of an intelligent system and their ways of interacting.

In the original Linda system, the operators access a blackboard that contains *tuples*, which consist of sequences of arbitrary data. For a system that supports multiple languages, the first element of the tuple should identify the language so that the Linda system could immediately determine how to interpret the remainder. A general format would have six elements:

1. *Language*—A character string that specifies the language, such as "KIF," "CGIF," "English," or "Deutsch"
2. *Source*—A character string that identifies the sender
3. *Message Id*—A character string generated by the sender
4. *Destination*—A character string that identifies the intended receiver, if known. For pattern-directed communications, this string is empty, and the message is matched to the patterns of available receivers.
5. *Speech Act*—A character string that states the speech act
6. *Message*—An arbitrary expression in the specified language that states the propositional content of the message

The Linda pattern matcher could use an ordinary string comparison for the first five elements of the tuple, but it would require a more general logical unification (of CGs or KIF statements) for the sixth. Unification of messages in controlled natural lan-

guages might be difficult to define, and the pattern matcher might need to translate the message to CGs or KIF if a pattern match is necessary.

To simulate a conventional scripting language, the destination would always be specified, and the speech act would always be "command." To access a relational database, the speech act would be "assertion" for an update, "question" for a query, or "definition" for creating a new table with a new format. At the end of his book, Austin[5] specified a large number of possible speech acts, and he insisted that his list was not exhaustive. Following are his five categories, his description of each, and a few of his examples:

1. *Verdictives* "are typified by the giving of a verdict, as the name implies, by a jury, arbitrator, or umpire."

   Examples: acquit, convict, calculate, estimate, measure, assess, characterize, diagnose

2. *Exercitives* "are the exercising of powers, rights, or influence."

   Examples: appoint, demote, excommunicate, command, direct, bequeath, claim, pardon, countermand, veto, dedicate

3. *Commissives* "are typified by promising or otherwise undertaking."

   Examples: promise, contract, undertake, intend, plan, propose, contemplate, engage, vow, consent, champion, oppose

4. *Behabitives* "are a very miscellaneous group, and have to do with attitudes and *social behavior*."

   Examples: apologize, thank, deplore, congratulate, welcome, bless, curse, defy, challenge

5. *Expositives* "make plain how our utterances fit into the course of an argument or conversation, how we are using words, or, in general, are expository."

   Examples: affirm, deny, state, assert, ask, identify, remark, mention, inform, answer, repudiate, recognize, define, postulate, illustrate, explain, argue, correct, revise, tell, report, interpret

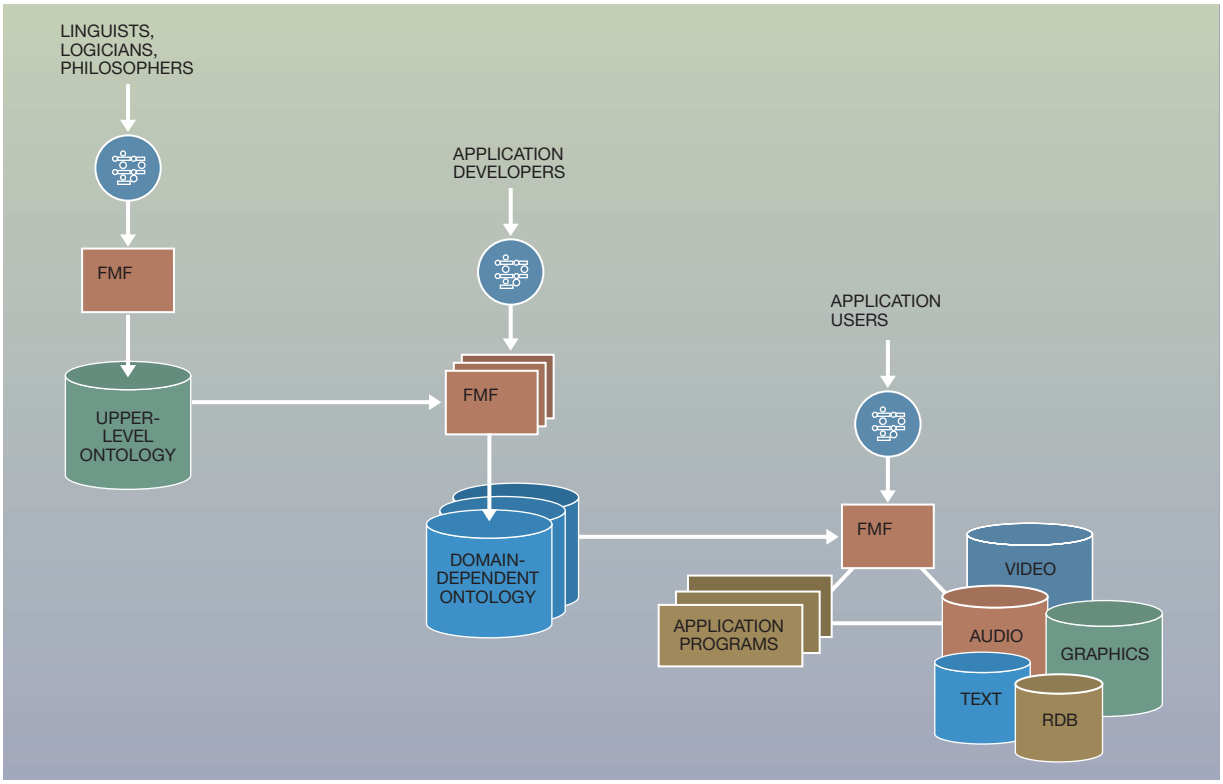
The verbs listed in these examples illustrate the kinds of speech acts that people commonly perform, but in most cases, they omit the verb that specifies the speech act. A man who stands up in a meeting to shout something in an angry voice seldom begins with the words "I protest." Yet the people in the audience would recognize that the new speaker is protesting rather than agreeing with the previous speakers. Computers, however, need to be told how to interpret such speech, and an explicit statement of the speech act would enable them to respond more "intelligently."

To illustrate the kinds of speech acts in an AI system, Figure 7 shows a kind of system discussed by Sowa.[51] The boxes labeled FMF represent the same flexible modular framework that has been adapted to different kinds of tasks by changing the roles and the kinds of speech acts expected of the users. At the upper left, linguists, logicians, and philosophers are using the FMF to define a general ontology. Logicians could use FMF to enter the definitions and axioms for logical operators, set theory, and basic mathematical concepts and relations. Linguists could use it to enter the grammar rules of natural languages and the kinds of semantic types and relations. Philosophers could use FMF to collaborate with the linguists and logicians in analyzing and defining the fundamental ontologies of space, time, and causality common to all domains of application. The major speech act for these users would be definition, but they might also ask questions about how to use the system, and they might use verdictives to evaluate the work of their colleagues.

In the center of Figure 7, application developers use FMF to enter domain-dependent information about specific applications. Some of them would use FMF to define generic ontologies for industries such as banking, agriculture, mining, education, and manufacturing. Others would start with one or more generic ontologies and combine them or tailor them to a particular business, project, or application. The users in this mode would perform the same kinds of speech acts as the linguists, logicians, and philosophers. But they might put more emphasis on commissives, which would commit them to strict deadlines and performance goals.

At the bottom right of Figure 7 the application users might interact with the FMF in an unpredictable number of ways. A business user with a job to do would have different requirements from those of a recreational user. Both, however, might react with behabitives, such as grumbling, complaining, or cursing, when the system does not do what they wish. But

Figure 7 Tailoring an FMF for different purposes



unlike more conventional systems, an FMF could apologize, sympathize, and commiserate.

The examples shown in Figure 7 do not begin to exploit the kinds of opportunities offered by an FMF that is able to recognize and respond to a wide range of speech acts. An important reason for building an FMF is to explore new ways of interaction, either between computers and humans or among mixed committees of human and computer participants. The explicit recognition and marking of speech acts enables the components of an FMF to interact, negotiate, and cooperate more intelligently among themselves and with their human users.

Implementing the FMF

A major advantage of a flexible modular framework is that it does not have to be implemented all at once. The four design principles, which enabled UNIX-like systems to be implemented on anything from a wearable computer to the largest supercomputers, can also support the growth of intelligent systems from simple beginnings to a complete "society of mind," as Minsky[52] called it. For an initial implementation, each of the four principles could be reduced to the barest minimum, but any of them could be enhanced incrementally without disturbing any previously supported operations:

1. The first component that must be implemented is a blackboard for passing messages. Even the pattern matcher might be omitted in the first implementation, and messages could only be sent to named destinations. For greater power and flexibility, a pattern matcher is necessary, but the basic Linda systems use only a simple pattern matcher that is far less sophisticated than most AI systems. The greatest power would come from patterns stored in a hierarchy of graphs based on the Levinson-Ellis algorithms, which could accommodate millions of patterns that might invoke intelligent agents distributed anywhere across the Internet.

2. Any components that accept inputs and generate outputs could be accommodated in an FMF. The first implementation might support only conventional components that do exactly what they are told, such as the traditional collection of UNIX utilities. More sophisticated components for reasoning, planning, problem solving, and natural-language processing could be added incrementally. The initial implementation would look like a pattern-matching front end to a UNIX command line, but it could grow arbitrarily far. Each stage in the growth would continue to have the full functionality of every preceding stage. Nothing would become obsolete as more intelligence is added by the new components.

3. The Elephant language, which serves as the glue for linking components, could also grow incrementally. An initial version could consist of just the three verbs implemented in conventional computer systems: *tell*, *ask*, and *do*. Those verbs correspond to the declarative, interrogative, and imperative moods of English. They also correspond to the three operators supported by a Linda blackboard: *output*, *input*, and *execute*. Those verbs would be necessary to support many, if not most of the messages in even the most sophisticated systems. As more sophisticated components are added to the FMF, other verbs could be added to support more complex interactions: *authorize* for secure communications; *reply*, *lock*, and *commit* for transactions that require multiple exchanges; *explain* for help facilities; and *promise* for future commitments.

4. The communication language for writing messages could be conventional first-order logic, which supports a wide range of simpler subsets, such as lists, frames, and rules. The richer CG language includes FOL, but with nested contexts and metalevel statements about the contents of any context. New languages and dialects of languages could be added whenever a translator becomes available for mapping them to and from the patterns of CGs that are used by the blackboard communication center.

As an FMF is being developed, it can accommodate any mixture of a variety of components: newly designed components specially tailored for the FMF; legacy systems enclosed in a wrapper that translates their I/O formats to the common language of the blackboard; commercial products that perform specific services; experimental components that are being designed and tested in research projects; an open-ended variety of client interfaces specialized for different applications; remote servers distributed anywhere across the Internet.

A blackboard is an ideal platform for supporting hot-swap or plug-n-play components. When a new component is added to the FMF, it would send a message to the blackboard to identify itself and the patterns of messages it accepts. It could then be invoked by any other component whose message matches the appropriate pattern. To take advantage of that flexibility, the Jini system uses the Linda operators to accommodate any kind of I/O device that might be attached to a network. But for intelligent systems, it is even more important to have that flexibility at the center instead of the periphery.

Any server anywhere on the Internet could be converted to an intelligent agent by using an FMF as its front end. It could then respond to requests from other FMF servers anywhere else on the Internet. Each FMF would be, in Minsky's terms, a society of mind, and the entire Internet would become a society of societies. Human users could have a personal FMF running on their own computers, which could communicate with any other FMF to request services. The traditional help desks, in which a human expert answers the same questions repeatedly for multiple users, could be replaced by a human teacher or editor, as in Figure 5, who would build a knowledge base. That knowledge base would drive a specialized FMF, which could be consulted by the personal FMF of anyone who asks a relevant question. The intelligence accessible to any user would then be the combined intelligence of his or her personal FMF together with every FMF accessible to it across the Internet.

Cited references and notes

1. T. Winograd and F. Flores, *Understanding Computers and Cognition*, Ablex, Norwood, NJ (1986).
2. R. Penrose, *The Emperor's New Mind*, Oxford University Press, Oxford (1989).
3. H. L. Dreyfus, *What Computers Still Can't Do: A Critique of Artificial Reason*, MIT Press, Cambridge, MA (1992).
4. J. McCarthy, "Elephant 2000: A Programming Language Based on Speech Acts," http://www-formal.stanford.edu/jmc/elephant.html.
5. J. L. Austin, *How to Do Things with Words*, Second Edition,

J. O. Urmson and M. Sbisá, Editors, Harvard University Press, Cambridge, MA (1975).
6. J. R. Searle, *Speech Acts: An Essay in the Philosophy of Language*, Cambridge University Press, Cambridge, MA (1969).
7. N. Carriero and D. Gelernter, *How to Write Parallel Programs*, MIT Press, Cambridge, MA (1992).
8. N. E. Fuchs, U. Schwertel, and R. Schwitter, "Attempto Controlled English—Not Just Another Logic Specification Language," *Proceedings of the 8th International Workshop on Logic-Based Program Synthesis and Transformation* (LOPSTR'98), Manchester, UK, June 1998, *Lecture Notes in Computer Science*, Vol. 1559, Springer-Verlag, New York (1998).
9. Dissertation, Institut fur Informatik, Universitat Zurich (1998), http://www.ifi.unizh.ch/groups/CL/schwitter/DissBook.ps
10. G. Frege, *Begriffsschrift*, English translation in J. van Heijenoort, Editor, *From Frege to Gödel*, Harvard University Press, Cambridge, MA (1967), pp. 1–82.
11. C. S. Peirce, "On the Algebra of Logic," *American Journal of Mathematics* **3**, 15–57 (1880).
12. C. S. Peirce, "On the Algebra of Logic," *American Journal of Mathematics* **7**, 180–202 (1885).
13. G. Peano, *Aritmetices principia nova methoda exposita*, Bocca, Torino (1889).
14. J. F. Sowa, "Relating Templates to Language and Logic," *Information Extraction: Towards Scalable, Adaptable Systems*, M. T. Pazienza, Editor, *Lecture Notes in Artificial Intelligence*, Vol. 1714, Springer-Verlag, Berlin (1999), pp. 76–94.
15. D. Skuce and T. Lethbridge, "CODE4: A Unified System for Managing Conceptual Knowledge," *International Journal of Human-Computer Studies* **42**, 413–451 (1995).
16. D. Skuce, "Intelligent Knowledge Management: Integrating Documents, Knowledge Bases, and Linguistic Knowledge," *Proceedings of the Eleventh Workshop on Knowledge Acquisition, Modeling and Management* (KAW'98), Banff, Alberta, Canada, April 1998, http://ksi.cpsc.ucalgary.ca/KAW/KAW98/KAW98Proc.html.
17. D. Skuce, "Integrating Web-Based Documents, Shared Knowledge Bases, and Information Retrieval for User Help," *Computational Intelligence* **16**, No. 1, 95–113 (2000).
18. A.-M. Rassinoux, *Extraction et Représentation de la Connaissance tirée de Textes Médicaux*, Éditions Systèmes et Information, Geneva (1994).
19. A.-M. Rassinoux, R. H. Baud, C. Lovis, J. C. Wagner, and J.-R. Scherrer, "Tuning up Conceptual Graph Representation for Multilingual Natural Language Processing in Medicine," *Conceptual Structures: Theory, Tools, and Applications*, M.-L. Mugnier and M. Chein, Editors, *Lecture Notes in Artificial Intelligence*, Vol. 1453, Springer-Verlag, Berlin (1998), pp. 390–397.
20. W. R. Cyre, S. Balachandar, and A. Thakar, "Knowledge Visualization from Conceptual Structures," *Conceptual Structures: Current Practice*, W. F. Tepfenhart, J. Dick, and J. F. Sowa, Editors, *Lecture Notes in Artificial Intelligence*, Vol. 835, Springer-Verlag, Berlin (1994), pp. 275–292.
21. W. R. Cyre, "Capture, Integration, and Analysis of Digital System Requirements with Conceptual Graphs," *IEEE Transactions on Knowledge and Data Engineering* **9**, No. 1, 8–23 (1997).
22. J. Hess and W. Cyre, "A CG-based Behavior Extraction System," *Conceptual Structures: Standards and Practices*, W. Tepfenhart and W. Cyre, Editors, *Lecture Notes in Artificial Intelligence*, Vol. 1640, Springer-Verlag, Berlin (1999).
23. O. Gerbé and M. Perron, "Presentation Definition Language Using Conceptual Graphs," *Conceptual Structures: Applications, Implementation, and Theory*, G. Ellis, R. A. Levinson, and W. Rich, Editors, *Lecture Notes in Artificial Intelligence*, Vol. 954, Springer-Verlag, Berlin (1995).
24. O. Gerbé, B. Guay, and M. Perron, "Using Conceptual Graphs for Methods Modeling," *Conceptual Structures: Knowledge Representation as Interlingua*, P. W. Eklund, G. Ellis, and G. Mann, Editors, *Lecture Notes in Artificial Intelligence*, Vol. 1115, Springer-Verlag, Berlin (1996), pp. 161–174.
25. O. Gerbé, "Conceptual Graphs for Corporate Knowledge Management," *Conceptual Structures: Fulfilling Peirce's Dream*, D. Lukose, H. Delugach, M. Keeler, L. Searle, and J. Sowa, Editors, *Lecture Notes in Artificial Intelligence*, Vol. 1257, Springer-Verlag, Berlin (1997), pp. 474–488.
26. O. Gerbé, R. Keller, and G. Mineau, "Conceptual Graphs for Representing Business Processes in Corporate Memories," *Conceptual Structures: Theory, Tools, and Applications*, M.-L. Mugnier and M. Chein, Editors, *Lecture Notes in Artificial Intelligence*, Vol. 1453, Springer-Verlag, Berlin (1998), pp. 401–415.
27. O. Gerbé and B. Kerhervé, "Modeling and Metamodeling Requirements for Knowledge Management," *Proceedings of OOPSLA'98 Workshops*, Vancouver, Canada, October 1998, ACM, New York (1998).
28. O. Gerbé, *Un Modèle uniforme pour la Modélisation et la Métamodélisation d'une Mémoire d'Entreprise*, Ph.D. dissertation, Département d'informatique et de recherche opérationelle, Université de Montréal (2000).
29. J. F. Sowa, "Conceptual Graphs for a Database Interface," *IBM Journal of Research and Development* **20**, No. 4, 336–357 (1976).
30. J. F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley Publishing Co., Reading, MA (1984).
31. R. A. Levinson and G. Ellis, "Multi-Level Hierarchical Retrieval," *Knowledge-Based Systems* **5**, No. 3, 233–244 (1992).
32. G. Ellis, R. A. Levinson, and P. J. Robinson, "Managing Complex Objects in Peirce," *International Journal of Human-Computer Studies* **41**, 109–148 (1994).
33. C. S. Peirce, papers on existential graphs in *Collected Papers of C. S. Peirce*, C. Hartshorne and P. Weiss, Editors, Harvard University Press, Cambridge, MA, Vol. 4, pp. 320–410 (1897–1906). See also MS 514, available at http://www.jfsowa.com/peirce/ms514.htm.
34. G. Gentzen, "Untersuchungen über das logische Schließen," translated as "Investigations into Logical Deduction" in *The Collected Papers of Gerhard Gentzen*, edited and translated by M. E. Szabo, North-Holland Publishing Co., Amsterdam (1969), pp. 68–131.
35. J. Esch and R. A. Levinson, "An Implementation Model for Contexts and Negation in Conceptual Graphs," *Conceptual Structures: Applications, Implementation, and Theory*, G. Ellis, R. A. Levinson, and W. Rich, Editors, *Lecture Notes in Artificial Intelligence*, Vol. 954, Springer-Verlag, Berlin (1995), pp. 247–262.
36. J. Esch and R. A. Levinson, "Propagating Truth and Detecting Contradiction in Conceptual Graph Databases," *Conceptual Structures: Knowledge Representation as Interlingua*, P. W. Eklund, G. Ellis, and G. Mann, Editors, *Lecture Notes in Artificial Intelligence*, Vol. 1115, Springer-Verlag, Berlin (1996), pp. 229–247.
37. J. Stewart, *Theorem Proving Using Existential Graphs*, M.S. thesis, Computer and Information Science, University of California at Santa Cruz (1996).
38. R. A. Levinson, "General Game-Playing and Reinforcement

Learning," *Computational Intelligence* **12**, No. 1, 155–176 (1996).
39. A. Majumdar, "About VivoMind," http://www.genumerix.com/rfi/AboutVivoMind.htm.
40. M. R. Quillian, *Semantic Memory*, abridged version in M. Minsky, *Semantic Information Processing*, MIT Press, Cambridge, MA (1966), pp. 227–270.
41. S. E. Fahlman, *NETL: A System for Representing and Using Real-World Knowledge*, MIT Press, Cambridge, MA (1979).
42. J. A. Hendler, *Integrating Marker Passing and Problem Solving*, Lawrence Erlbaum Associates, Hillsdale, NJ (1987).
43. H. Kamp, "Events, Discourse Representations, and Temporal References," *Langages* **64**, 39–64 (1981).
44. H. Kamp and U. Reyle, *From Discourse to Logic*, Kluwer, Dordrecht (1993).
45. J. F. Sowa and E. C. Way, "Implementing a Semantic Interpreter Using Conceptual Graphs," *IBM Journal of Research and Development* **30**, No. 1, 57–69 (1986).
46. S. Callaghan, *Optimising Comparisons of Complex Objects by Precomputing Their Graph Properties*, Ph.D. dissertation, Royal Melbourne Institute of Technology, Melbourne, Australia (2001).
47. B. J. Peterson, W. A. Andersen, and J. Engel, "Knowledge Bus: Generating Application-Focused Databases from Large Ontologies," *Proceedings of the 5th Knowledge Representation Meets Databases* (*KRDB*) *Workshop*, Seattle, WA, May 1998. Available from http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-10/.
48. D. B. Lenat, "Cyc: A Large-Scale Investment in Knowledge Infrastructure," *Communications of the ACM* **38**, No. 11, 33–38 (1995). For further information, see http://www.cyc.com.
49. D. B. Lenat and R. V. Guha, *Building Large Knowledge-Based Systems*, Addison-Wesley Publishing Co., Reading, MA (1990).
50. J. Engel, *Programming for the Java Virtual Machine*, Addison-Wesley Publishing Co., Reading, MA (1999).
51. J. F. Sowa, *Knowledge Representation: Logical, Philosophical, and Computational Foundations*, Brooks/Cole, Pacific Grove, CA (2000).
52. M. Minsky, *The Society of Mind*, Simon & Schuster, New York (1985).

Accepted for publication April 15, 2002.

John F. Sowa *(electronic mail: sowa@bestweb.net).* Dr. Sowa retired from IBM in 1992 after thirty years of working on research and development projects. He is now the chief scientist at a start-up company, VivoMind LLC, which is developing software based on conceptual graphs. He has a B.S. degree in mathematics from MIT, an M.A. degree in applied mathematics from Harvard University, and a Ph.D. degree in computer science from the Vrije Universiteit Brussel. He is a Fellow of the American Association for Artificial Intelligence, and he has participated in ANSI and ISO standards projects for conceptual graphs, knowledge sharing, and ontology. He has published many books and papers on artificial intelligence, natural-language processing, and related topics.