*Computers Math. Applic.* Vol. 23, No. 2–5, pp. 75–93, 1992
Printed in Great Britain. 

0097-4943/92 $5.00 + 0.00


# CONCEPTUAL GRAPHS AS A UNIVERSAL KNOWLEDGE REPRESENTATION

JOHN F. SOWA
IBM Systems Research
500 Columbus Avenue, Thornwood, NY 10594, U.S.A.

**Abstract**—Conceptual graphs are a knowledge representation language designed as a synthesis of several different traditions. First are the semantic networks, which have been used in machine translation and computational linguistics for over thirty years. Second are the logic-based techniques of unification, lambda calculus, and Peirce's existential graphs. Third is the linguistic research based on Tesnière's dependency graphs and various forms of case grammar and thematic relations. Fourth are the dataflow diagrams and Petri nets, which provide a computational mechanism for relating conceptual graphs to external procedures and databases. The result is a highly expressive system of logic with a direct mapping to and from natural languages. The lambda calculus supports the definitions for a taxonomic system and provides a general mechanism for restructuring knowledge bases. With the definitional mechanisms, conceptual graphs can be used as an intermediate stage between natural languages and the rules and frames of expert systems—an important feature for knowledge acquisition and for help and explanations. During the past five years, conceptual graphs have been applied to almost every aspect of AI, ranging from expert systems and natural language to computer vision and neural networks. This paper surveys conceptual graphs, their development from each of these traditions, and the applications based on them.

## 1. A COMMON FRAMEWORK FOR AI

Natural languages and symbolic logic are both universal knowledge representations. Natural languages set the standard for flexibility and expressive power: they deal with every aspect of human life, including everything that could be said in any artificial language. Symbolic logic sets the standard for precision and generality in formal systems: any program running on a digital computer and even the computer itself could be defined in logic. The design goal for conceptual graphs is to provide a universal knowledge representation language that combines the expressive power of natural languages with the precision of symbolic logic. The book *Conceptual Structures* [1] presents them as a synthesis of the formalisms used in AI and cognitive science. After surveying philosophical and psychological issues in the first two chapters, it develops the formal theory of conceptual graphs and applies it to logic, linguistics, and knowledge engineering. The last chapter analyzes the limitations of conceptual thinking and returns to themes from the first: the trade-offs between discrete symbols and continuous imagery and between logical and associative modes of thought. Throughout the book, the common thread is the formalism of conceptual graphs and their application to language, reasoning, perception, and behavior.

That book started an international movement with six annual workshops since 1986—the most recent ones sponsored by AAAI, IJCAI, and ECAI. Since anything related to conceptual graphs and their applications is within the scope of the workshops, the topics presented there have been as broad as the field of AI itself:

- Natural language: parsing, generation, discourse analysis, text generation, tense and aspect, lexical choice, contexts, metaphor, analogies, thematic relations, and machine translation.
- Logic and reasoning: inference engines, inheritance theory, truth maintenance, induction, modality, uncertainty, plausible reasoning, and reasoning about sets and numeric quantifiers.

- Knowledge engineering: knowledge acquisition, machine learning, extracting knowledge from texts, requirements analysis, systems analysis, database design, and semantic data models.
- Hybrid systems: conceptual programming, logic programming with conceptual graphs, and pattern recognizers that combine neural networks with conceptual graphs.
- Applications: expert systems, information retrieval, intelligent help, computer vision, robotics, computer-integrated manufacturing, and communication in distributed systems.

In scope, the conceptual graph workshops are like miniature IJCAI's. But at large conferences, the AI community is fragmented, and people rarely attend talks outside their narrow specialties. At the conceptual graph workshops, all the presentations start with a common formalism, and people with different specialties talk with one another on a deeper level. Because of their generality as a system of logic and their readability as a graphic notation, conceptual graphs have been proposed as a standard for information interchange by the ANSI X3H4 Committee on Information Resource Dictionary Systems.

Conceptual graphs appeal to both scruffies and neats. For the neats, they are a system of logic with a model-theoretic semantics. For the scruffies, they are data structures that can be manipulated with the heuristic techniques of AI. The next section of this paper discusses the sources of conceptual graphs in both the scruffy and the neat traditions. Section 3 gives examples of the graphs and their relationship to frames, predicate calculus, and other systems. The last section surveys applications of conceptual graphs that have been implemented in the last five years.

## 2. FOUNDATIONS OF CONCEPTUAL GRAPHS

The box and circle notation for conceptual graphs was invented by John Sowa, but the formal structures are a synthesis of a century of work in logic, linguistics, philosophy, and AI. The logical foundations are based on the work by the philosopher and logician C.S. Peirce. In 1883, Peirce developed the first linear notation for first-order logic. With later variations by Schröder, Peano, and Russell, it evolved into the modern system of predicate calculus. In Peirce's original notation, the sentence *A farmer owns a donkey* could be represented by the following formula:

$$\Sigma_x \Sigma_y \, (\mathbf{farmer}_x \bullet \mathbf{donkey}_y \bullet \mathbf{owns}_{xy}) .$$

Except for the differences in parentheses and operator symbols ($\Sigma$ for $\exists$ and $\bullet$ for $\wedge$), this formula is equivalent to the modern notation for predicate calculus:

$$(\exists \mathbf{x})(\exists \mathbf{y})(\mathbf{farmer(x)} \wedge \mathbf{donkey(y)} \wedge \mathbf{owns(x,y)}) .$$

Yet Peirce felt that the predicate notation for logic was unduly complex. In 1882, he had developed his *relational graphs*, which could express the sentence *A farmer owns a donkey* in a much simpler way:

**farmer—owns—donkey.**

In this notation, the bars represent existential quantifiers: the bar on the left corresponds to $(\exists \mathbf{x})$, and the one on the right corresponds to $(\exists \mathbf{y})$. Figure 1 shows a more complex graph for the sentence *A farmer owns and beats a donkey.*

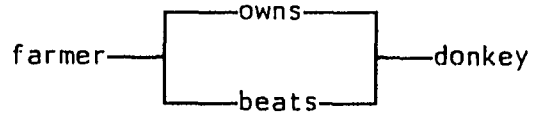


Figure 1. Relational graph for "A farmer owns and beats a donkey."

The interlinked bars on the left represent $(\exists \mathbf{x})$, and the ones on the right represent $(\exists \mathbf{y})$. Figure 1 corresponds to the following formula in predicate calculus:

$$(\exists \mathbf{x})(\exists \mathbf{y})(\mathbf{farmer(x)} \wedge \mathbf{donkey(y)} \wedge \mathbf{owns(x,y)} \wedge \mathbf{beats(x,y)}) .$$

As this formula illustrates, relational graphs can represent conjunctions and existential quantifiers. Peirce tried marking the nodes and arcs with special symbols that would show negations, disjunctions, and universal quantifiers; but he could not find a general way to represent all possible combinations of those operators. For the next 15 years, he did most of his research in logic with the linear notation. Then in 1897, he invented *existential graphs* with the simple, but powerful mechanism of graphs nested within contexts that were parts of larger graphs. The nested contexts enabled him to extend the power of the graphs to all of first-order logic and later to modal and higher-order logic as well. Figure 2 shows an existential graph for the sentence *If a farmer owns a donkey, then he beats it.*

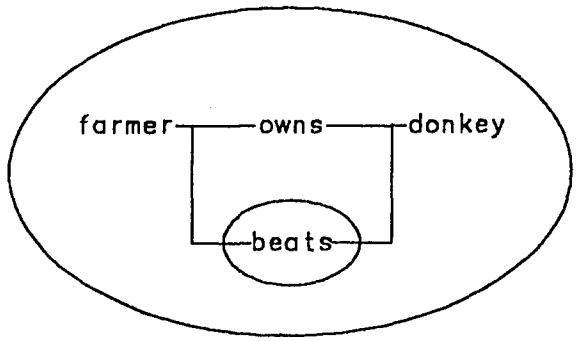


Figure 2. Existential graph for "If a farmer owns a donkey, then he beats it."

In Figure 2, each oval represents a negation. The nest of two ovals represents implication, since $\sim (p \wedge \sim q)$ is equivalent to $(p \supset q)$. The *if*-oval contains one bar for the farmer and one for the donkey, both of which extend into the *then*-oval. The two bars correspond to the variables **x** and **y** in the predicate calculus:

$$\sim (\exists x)(\exists y)(\mathbf{farmer}(x) \wedge \mathbf{donkey}(y) \wedge \mathbf{owns}(x,y) \wedge \sim \mathbf{beats}(x,y)).$$

With the symbol $\supset$ for implication, the existential quantifiers for *a farmer* and *a donkey* must be replaced by universal quantifiers in front of the formula:

$$(\forall x)(\forall y)\ ((\mathbf{farmer}(x) \wedge \mathbf{donkey}(y) \wedge \mathbf{owns}(x,y)) \supset \mathbf{beats}(x,y)).$$

If this formula were translated directly into English, the result would sound unnatural: *For all x and all y, if x is a farmer and y is a donkey and x owns y, then x beats y.* For representing implications, Peirce's ovals have a more direct mapping to English than the predicate calculus form with the $\supset$ operator.

In developing existential graphs, Peirce was searching for the simplest primitives that would support all of logic; he was not trying to preserve linguistic naturalness. Remarkably, his nested contexts turn out to be isomorphic to the *discourse representation structures* (DRS) by Hans Kamp [2], who was trying to represent quantification and anaphora in natural languages. Figure 2 shows the DRS for Peirce's Figure 2.

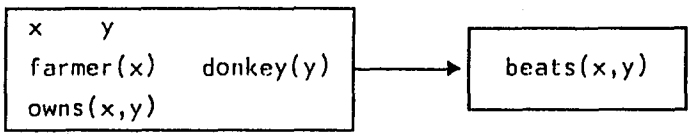


Figure 3. DRS for "If a farmer owns a donkey, then he beats it."

In Figure 3, the variables **x** and **y** represent *discourse referents*, which correspond to Peirce's bars. The two boxes are contexts, and the arrow represents implication. Kamp's implication does not correspond to the symbol $\supset$ in predicate calculus, since Kamp makes the additional assumption that variables in the consequent are within the scope of discourse referents in the antecedent. That

assumption also holds for Peirce's contexts, but not for predicate calculus with the ⊃ operator. There is no room in this article to summarize all of Kamp's examples and arguments, but the important point is the one-to-one relationship between Kamp's boxes and variables and Peirce's ovals and bars. Kamp's search rules for resolving pronouns and other anaphoric references in DRS form also apply to Peirce's contexts. When independent efforts proceeding from different directions lead to isomorphic structures, that is evidence for the importance of those structures.

Although Peirce and Kamp could represent the global structure of logic and discourse with their nested contexts, they did not address the finer linguistic details inside the contexts. To represent that level, Lucien Tesnière [3] developed graph structures for his system of dependency grammar. For over 20 years, Igor Mel'čuk [4] has been using them for detailed studies of lexical patterns in Russian and French. For computational linguistics, David Hays [5] adopted them and influenced Roger Schank, who shifted the emphasis from syntactic dependencies to conceptual dependencies [6]. But despite their linguistic and computational sophistication, the graphs by Tesnière, Hays, Schank, and Mel'čuk are at the same logical level as Peirce's relational graphs of 1882. Peirce's contexts provide the missing structures that are needed to support all of logic. By combining the contexts with the dependency graphs, conceptual graphs not only support Kamp's discourse theory; they also provide a formalism that can represent Schank's scripts [7] and memory organization packets [8].

Tesnière concentrated on the syntactic relations between words and phrases, but he did not distinguish the different kinds of semantic relations. For the following sentences, his dependency graphs would have unlabeled arcs linking the verb to the subject and object:

- *Tom built a house.*
- *He owned the house.*
- *He liked the house.*
- *He sold the house.*

Syntactically, these four sentences have the same subject-verb-object pattern. Semantically, however, the underlying relations are very different. Building and selling are actions, whose subjects are *agents*; but owning and liking are states, which do not have agents. Since liking is a mental state, its subject would be an *experiencer*. Since owning is a legal state, its subject is merely in a *state* of ownership. Since the object of building does not exist until after the action is performed, it is the *result* of the action. But for the other verbs, the object is a passive participant that exists before the action or state; it is usually called the *patient*. To distinguish these semantic relations, the arcs on the graphs should be labeled with *agent* and *patient* or their abbreviations AGNT and PTNT.

For his system of *correlational nets*, Silvio Ceccato [9] labeled the arcs with 56 different relation types. Besides the linguistic relations, he included time and place relations and structural relations like part-whole or element-collection. Although Ceccato was the first to implement a graph system with labeled relations, he mixed the low-level linguistic relations with more complex relations such as puller to thing pulled. The linguists Fillmore [10] and Gruber [11] later analyzed the relations in greater depth and classified them more systematically. Their *case roles* or *thematic relations* have become a standard feature of linguistic theory. The classification of the relations and their use in word definition and semantic structure is still an active area of research; see [12] for a recent collection of papers on the topic.

For conceptual graphs, the linguistic relations are taken as primitives, but high-level relations such as puller to thing pulled can be defined in terms of them. High-level relations are especially common in expert systems, where the rules and frames tend to use complex relations such as original-cost-in-US-dollars or number-of-loading-cycles. Those relations are far removed from the linguistic primitives. But the high-level concept and relation types can be defined in terms of the low-level types. In the traditional notation, a function $f$ could be defined by an equation like the following:

$$f(x) = ax^2 + bx + c.$$

In this equation, $f(x)$ names the function $f$ and its formal parameter $x$ at the same time. The expression on the right is the body of the definition, in which the parameter $x$ is replaced by the value of the argument whenever the function is invoked. Alonzo Church [13] developed the

*λ-calculus* as a way of separating the name of a function from its definition:

$$f = (\lambda x)(ax^2 + bx + c).$$

On the left is the name $f$, and on the right is a *λ-expression* that defines $f$. As merely a notation, the λ-calculus is no better than the traditional definitions. More important is the ability to perform operations on the definition itself; that is only possible when the definition and the name can be separated. When they are separated, the λ-expressions can be freely transformed by Church's rules for expansion and contraction, which generate intermediate structures for which names would be an encumbrance. With conceptual graphs, the λ-calculus provides a way of reorganizing knowledge by defining the concepts and relations of one knowledge base in terms of the primitives of another; Church's rules can then transform one into the other [14]. For natural language, λ-expressions are also used in representing generalized quantifiers, relative clauses, elliptical expressions, metaphor, and metonymy [15].

Pattern matching has always been an important AI technique, and graphs are especially good for supporting pattern matches over complex structures. Alan Robinson's unification algorithm for theorem proving [16] is a general and elegant form of pattern matching that was adapted to graph unification as the *maximal join* algorithm [17]. Besides the ability to represent logic and language, a computational system requires links to other computer facilities and the outside world. Those are provided by *actor nodes* attached to conceptual graphs. They are associated with external procedures and database relations, which may be triggered by a *marker passing* algorithm inspired by Petri nets [18].

As this survey shows, conceptual graphs were derived from a synthesis of logic and linguistics: the logic is based on Peirce's graphs, unification, and the λ-calculus; the linguistics is based on dependency graphs and case grammar. The synthesis was not achieved all at once: the first version of conceptual graphs was developed in 1968 as a term paper for Marvin Minsky's course on AI at MIT. It introduced the box and circle notation with nested boxes for subordinate clauses in English. The only two graph influences on that system were Quillian [19] and Hays [5]. In the first published version [17], the formation rules and graph unification were added under the influence of lectures given at IBM by Alan Robinson. To support database query, the actor nodes were added, and the terminology was chosen for consistency with relational database theory: the joins and projections of conceptual graphs are the intensional counterparts of the joins and projections of database relations. At a workshop on graph grammars, the λ-calculus for definitions was first presented [20]. At another conference [21], Hintikka's game-theoretical semantics was adapted to the graphs; Peirce's logic was not yet fully integrated with the graphs. Since Roger Schank has always eschewed formalism, his work has not contributed any formal features to the conceptual graphs. However, his research has had a strong influence on the way the graphs are used to support AI applications: Schank and his students are second only to Peirce in the number of references in the index of *Conceptual Structures*. When that book was finished in 1983, some recent linguistic developments were not incorporated in it: generalized quantifiers, situation semantics, and Kamp's discourse representation theory. Yet they did not require any new features in the formalism: since the conceptual graph contexts were based on Peirce's, they were isomorphic to Kamp's; the λ-calculus provided the basis for representing generalized quantifiers; and situation semantics led to a clarification of the meaning of the graphs, but it did not require any changes in them.

Although the box and circle notation or the equivalent linear form is the most obvious feature of conceptual graphs, the notation is not part of the formal theory. In *Conceptual Structures*, no definition, assumption, theorem, or proof ever refers to the notation; any notation that conforms to the definitions may be used. Some semantic networks always include the type-subtype arrows in the diagrams. For conceptual graphs, those arrows are usually omitted because they tend to make the diagrams too cluttered. Yet that is a purely stylistic preference, and the type-subtype arrows are sometimes drawn for emphasis. To show conceptual relations, Schank's graphs [22] have various kinds of arrows instead of labeled circles. That is another stylistic preference that could also be adopted, if desired. Since Kamp's boxes are isomorphic to Peirce's ovals and hence to the conceptual graph boxes, Kamp's DRS notation is also acceptable. The purpose of the

conceptual graph theory is not to exclude or replace all other theories, but to accommodate them and show how they are interrelated. A great deal of research has been done in those traditions, and when appropriate, it should be accepted, adopted, and used.

## 3. EXAMPLES OF CONCEPTUAL GRAPHS

A distinctive feature of conceptual graphs is the amount of structure inside the concept and relation nodes. Figure 4 shows three concepts. The first contains only the *type label* **CAT**; it may be read as the English phrase *a cat*. The second has a colon, which separates the type from the *referent*, which in this case names the cat **Yojo**. The third concept has a *plural referent*, which names a set of two cats, **Yojo** and **Tiger Lily**.


Figure 4. Concepts with type and referent fields.

There is a *formula operator* $\phi$, which maps conceptual graphs to formulas in the predicate calculus. It maps the first concept to a formula that says there exists a cat: **(∃x)cat(x)**. It maps the second concept to a formula that says Yojo is a cat: **cat(Yojo)**. It maps the third concept to a formula that says there exists a set **x** consisting of Yojo and Tiger Lily and every **y** in **x** is a cat:

$$(\exists x)(x=\{\text{Yojo, Tiger Lily}\} \wedge (\forall y)(y \in x \supset \text{cat}(y))).$$

Besides names and sets, the referent field may contain other information that determines what the concept refers to: a quantifier such as ∀ for *every* or **@most** for *most*; an *indexical referent* such as **#** for the definite article *the*; a variable such as ***x** or ***y** to show a *coreference link*; or even a nested conceptual graph to represent one of Peirce's contexts. Besides a type label like **CAT**, the type field could contain a λ-expression that defines a new type.

Boxes represent concepts, and circles represent conceptual relations. Figure 5 shows a conceptual graph for the sentence *A cat chased a mouse*. It contains four concept boxes: one for the cat, one for the chase, one for the mouse, and a large one that encloses the others. The agent relation **(AGNT)** links the chase to the cat; patient **(PTNT)** links the chase to the mouse; and past **(PAST)** attaches to the box that represents a situation in the past.

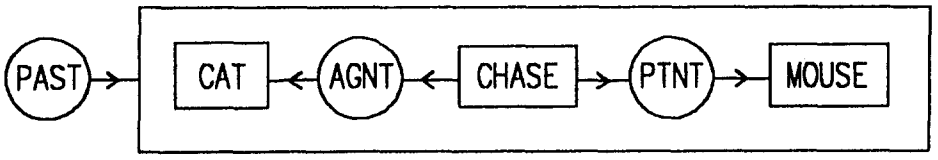


Figure 5. Conceptual graph for "A cat chased a mouse."

A concept that encloses another graph is called a *context*. This one has an implicit type label **SITUATION**, which was omitted to save space; its referent field contains the nested graph that describes the situation. Besides the box and circle notation, there is a more compact linear form that is easier to type. Figure 5 would be represented with square brackets for the concepts and rounded parentheses for the relations:

**(PAST)→[ [CAT]←(AGNT)←[CHASE]→(PTNT)→[MOUSE] ].**

In the mapping to predicate calculus, the formula operator $\phi$ assigns a constant or quantified variable to each concept. Type labels on the concepts become monadic predicates, and relations with $n$ arcs become $n$-adic predicates. Following is the formula that corresponds to Figure 5:

**past((∃x)(∃y)(∃z)(cat(x) ∧ chase(y) ∧ mouse(z) ∧ agnt(y,x) ∧ ptnt(y,z))).**

This formula treats *chase* as a predicate that applies to an *event variable* **y**. The **PAST** relation becomes a predicate that applies to an entire formula. The arrows on the conceptual relations have no semantic significance. Their only meaning is to distinguish the arguments: the arrow pointing towards the circle marks the first argument of the predicate form, and the arrow pointing away marks the last argument; if a relation has more than two arguments, the arrows are numbered. As a mnemonic, the arrow pointing towards the circle may be read "has a," and the one pointing away may be read "which is."

The concepts [CAT] and [MOUSE] refer to some indefinite individuals of type **CAT** and **MOUSE**. The sentence *Yojo is chasing Mickey* would be represented with names in the referent fields:

$$\texttt{[CAT: Yojo]}\leftarrow\texttt{(AGNT)}\leftarrow\texttt{[CHASE]}\rightarrow\texttt{(PTNT)}\rightarrow\texttt{[MOUSE: Mickey]}.$$

The sentence *Every cat chases three mice* would be represented by a quantifier $\forall$ for *every* and a *plural referent* for the mice:

$$\texttt{[CAT :}\forall\texttt{]}\leftarrow\texttt{(AGNT)}\leftarrow\texttt{[CHASE]}\rightarrow\texttt{(PTNT)}\rightarrow\texttt{[MOUSE: \{*\}@3]}.$$

The symbol {*} represents a set of unspecified individuals; the concept type shows that they must all be mice; and @3 says that the count or cardinality of the set is 3. The operator $\phi$ maps this graph to the following formula:

$$\begin{aligned}&(\forall x)(\mathtt{cat}(x) \supset (\exists y)(\mathtt{set}(y) \wedge \mathtt{count}(y,3)\\ &\quad \wedge (\forall z)(z \in y \supset (\mathtt{mouse}(z)\\ &\qquad \wedge (\exists w)(\mathtt{chase}(w) \wedge \mathtt{agnt}(w,x) \wedge \mathtt{ptnt}(w,z))))))).\end{aligned}$$

As this example shows, the mapping from English to conceptual graphs is short and simple, but the mapping to predicate calculus is much more complex. The quantifiers in the formula show that for each cat **x** there exists a set **y** of three mice, and for each mouse **z** in **y** there exists an instance of chasing **w** of the mouse **z** by the cat **x**. Note that plural referents generate two variables: **y** represents the set of mice, and **z** ranges over individual mice in the set **y**.

In conceptual graphs, the scope of quantifiers is not determined by linear order, but by precedence: universal quantifiers have high precedence, the quantifiers for plurals have middle precedence, and the default existentials have low precedence. When it is necessary to override the default precedence, context boxes are used to delimit the scope. For plurals, the markers **Dist** and **Col** change the precedence for the distributive and collective interpretations. If every cat chased all three mice at once, the collective marker **Col** would be inserted in front of the plural referent:

$$\texttt{[CAT: }\forall\texttt{ ]}\leftarrow\texttt{(AGNT)}\leftarrow\texttt{[CHASE]}\rightarrow\texttt{(PTNT)}\rightarrow\texttt{[MOUSE: Col\{*\}@3]}.$$

The marker **Col** lowers the precedence of the quantifiers for the plural so that there is only one instance of chasing for all three mice:

$$\begin{aligned}&(\forall x)(\mathtt{cat}(x) \supset (\exists y)(\exists z)(\mathtt{chase}(y) \wedge \mathtt{set}(z)\\ &\quad \wedge \mathtt{agnt}(y,x) \wedge \mathtt{count}(z,3) \wedge (\forall w)(w \in z \supset\\ &\qquad (\mathtt{mouse}(w) \wedge \mathtt{ptnt}(y,w))))).\end{aligned}$$

The addition of the prefix **Col** to one concept node caused a major rearrangement of the resulting formula. In this case, there is only one instance of chasing **y** for each cat, and all three mice are patients of **y**. Local changes in an English sentence lead to local changes in the conceptual graph, but they may cause a global reshuffling of the quantifiers in the formula. The ability to preserve locality is a major reason why conceptual graphs have a simpler mapping to natural language. The details of unscrambling the precedence levels and expanding the plural referents are handled by the formula operator $\phi$ [15].

The previous examples include more details than most logic textbooks, which typically give a simpler formula for the sentence *A cat chased a mouse*:

$$(\exists x)(\exists y)(\mathtt{cat}(x) \wedge \mathtt{mouse}(y) \wedge \mathtt{chased}(x,y)).$$

A linguist would consider this formula too simplified to represent the significant features of language. When the tense is bundled into the predicate **chased(x,y)**, there is no way to represent all possible tenses in a systematic way. For a manner adverb like *eagerly*, the predicate **chased(x,y)** has no place for attaching the adverb, but the conceptual graph allows the adverb to be attached to the verb node. These are the same reasons why treating verbs as conceptual relations is linguistically unsatisfactory:

[CAT]→(CHASED)→[MOUSE].

This graph, which maps to the previous formula by $\phi$, has the same weaknesses: the relation (CHASED) buries the tense inside the relation node, and it leaves no room for attaching adverbs or other adjuncts. Although a graph like that would not be used for linguistics, it might be used in an expert system rule. To relate it to the linguistic form, a $\lambda$-expression could be used to define CHASED in terms of the more primitive concepts and relations:

CHASED = (λx,y) [ANIMATE: *x] [ENTITY: *y]
(PAST)→[ [*x]←(AGNT)←[CHASE]→(PTNT)→[*y] ].

This definition says that CHASED is a label for a $\lambda$-expression with two parameters x and y. The parameter x must refer to something of type ANIMATE or some subtype like CAT, and the parameter y must refer to something of type ENTITY or one of its subtypes. The second line says that in the past, x was the agent of chasing y. With this definition, the $\lambda$-expression for CHASED could be expanded to recover the details.

The relation PAST does not show all the details that linguists require for tenses and aspects. A more detailed *temporal logic* would show how the event in the situation box occurs at a point in time that precedes the context-dependent speech time. Those details could be shown by defining the PAST relation with a $\lambda$-expression:

PAST = (λx) [TIME: #speech-time]←(SUCC)←[TIME]←(PTIM)←[SITUATION:*x].

This definition says that PAST is a label for a $\lambda$-expression with one parameter x, which refers to a situation. The # symbol marks contextually defined referents for concepts. In this case, #speech-time refers to the time of speech for the current context. The definition may be read "The speech time is the successor (SUCC) of a time, which is the point in time (PTIM) of a situation x."

In situation semantics [23], it is important to distinguish between a situation in the real world and a proposition that describes a situation. Figure 5 is an abbreviated graph whose expanded form makes that distinction. The omitted type label, which is implied by the relation PAST, is SITUATION. When a graph is the referent of a concept of type SITUATION, the graph is not the situation. Instead, the graph states a proposition that describes the situation. Those relationships are shown explicitly in the fully expanded graph:

[TIME: #speech-time]←(SUCC)←[TIME]-
(PTIM)←[SITUATION]→(DSCR)→[PROPOSITION]-
(STMT)→[GRAPH: [CAT]←(AGNT)←[CHASE]→(PTNT)→[MOUSE]].

Since this graph is too long to fit on one line, the hyphens show the continuations on subsequent lines. It says that the contextually defined speech time is the successor of a time, which is the point in time of a situation, which has a description (DSCR), which is a proposition, which has a statement (STMT), which is a graph for a cat chasing a mouse. In effect, the type label GRAPH quotes a conceptual graph and treats it as a literal. The type labels PROPOSITION and SITUATION, however, indicate that the graph is being used to state a proposition or describe a situation.

As these examples illustrate, conceptual graphs represent simple things simply, but they are flexible enough to represent knowledge at any level of detail. With the standard guidelines for mapping to and from natural language, Figure 5 is the preferred graph. It captures the significant information without burying it in a mass of detail. Most parsers that map natural language into

conceptual graphs generate graphs at that level. When the details are irrelevant, they can be hidden by $\lambda$-contraction. When the details are needed, they can be recovered by $\lambda$-expansion.

Frames are simpler than graphs, and a typical frame maps directly into a conceptual graph. As an example, consider the following frame, taken from an example for the Cyc project [24]. This frame shows the attributes of a car owned by a person named Fred:

```
TheStructuredIndividualThatIsFredsCar
   instanceOf: Camaro
   allInstanceOf: Camaro, Chevy, AmericanCar, Automobile,
      Vehicle, Device, IndividualObject, Thing
   owner: Fred
   yearOfManufacture: 1988
   originalCostInUS$: 15000
   parts: FredsCarsSteeringWheel,FredsCarsLeftFrontTire,...
   setOfParts: TheSetOfPartsOfFredsCar
```

The first word *TheStructuredIndividualThatIsFredsCar* is the name of an entity. Its type is **Camaro**, and all its supertypes are listed on lines 3 and 4 of the frame. The next three lines state its owner, year of manufacture, and original cost. Line 8 lists the names of the entities that are parts of the car, and the set of all parts has a separate name *TheSetOfPartsOfFredsCar*.

When a frame is mapped to a conceptual graph, slots in the frame become dyadic relations; constraints on the slots become type labels of concepts; and values in the slots go into the referent fields of concepts. The frame for Fred's car can be represented by the following graph:

```
[CAMARO: TheStructuredIndividualThatIsFredsCar]-
   (OWNR)→[PERSON: Fred]
   (WHEN-MANUFACTURED)→[YEAR: 1988]
   (ORIGINAL-COST)→[MONEY: @ $15,000 US]
   (PART)→[ENTITY: {FredsCarsSteeringWheel,FredsCarsLeftFrontTire,...}]-
            (NAME)→[WORD:TheSetOfPartsOfFredsCar].
```

The first line has a concept of type **CAMARO** whose referent field contains the name **TheStructuredIndividualThatIsFredsCar**. The hyphen after that concept indicates that the relations attached to it are continued on subsequent lines. The **@** symbol in the concept of type **MONEY** indicates that **$15,000** is not the name of an individual instance of money, but a measure of some amount of money. The set of parts, enclosed in braces, is the referent of a concept of type **ENTITY**, and the name of that set is a **WORD** with referent **TheSetOfPartsOfFredsCar**. The frame, but not the conceptual graph, also includes the path in the type hierarchy from **CAMARO** to **THING**. That could be shown in a separate statement:

```
CAMARO < CHEVY < AMERICANCAR < AUTOMOBILE
   < VEHICLE < DEVICE < INDIVIDUALOBJECT < THING.
```

The definitional mechanisms based on $\lambda$-calculus allow high-level relations in the frame to be mapped into the more primitive conceptual relations like **AGNT**, **PTNT**, or **PTIM**. The **WHEN-MANUFACTURED** relation requires a nested context for the point in time when the entity was manufactured:

```
WHEN-MANUFACTURED = (λx,y)
    [ENTITY: *x]
    [TIME: *y]←(PTIM)←[SITUATION: [MANUFACTURE]→(RSLT)→[*x]].
```

This definition relates an **ENTITY** **x** to a **TIME** **y**, which is the point in time of a **SITUATION**, in which there existed an act of **MANUFACTURE**, which had a result **x**. Then the **ORIGINAL-COST**

relation can be defined in terms of **WHEN-MANUFACTURED**:

```
ORIGINAL-COST = (λx,y)
  [ENTITY: *x] [MONEY: @ *y]
  [*x]→(WHEN-MANUFACTURED)→[YEAR]←(PTIM)←[SITUATION:
       [*x]→(STAT)←[COST]→(PTNT)→[*y] ].
```

This definition relates an **ENTITY** x to an amount of **MONEY** y, where x is linked by **WHEN-MANUFACTURED** to a **YEAR**, which is the point in time of a **SITUATION**, in which x was in the state of **COST** with patient y. With these definitions, the conceptual graph for Fred's car can be expanded to a form that contains only the linguistic relations that map directly to natural language:

```
[CAMARO: TheStructuredIndividualThatIsFredsCar *x]-
  (PTNT)←[OWN]←(STAT)←[PERSON: Fred]
  (PART)→[ENTITY: {FredsCarsSteeringWheel, FredsCarsLeftFrontTire,...}]-
     (NAME)→[WORD:TheSetOfPartsOfFredsCar];

[YEAR: 1988]-
  (PTIM)←[SITUATION: [MANUFACTURE]→(RSLT)→[*x] ]
  (PTIM)←[SITUATION: [*x]→(STAT)←[COST]-
     (PTNT)→[MONEY: @$15,000 US] ].
```

Note that the original frame had a flat structure, but this graph contains nested structures. The nesting resulted from expanding the **ORIGINAL-COST** and **WHEN-MANUFACTURED** relations, which represent situations at times in the past. High-level relations are useful for suppressing irrelevant detail. But if those details have significant interactions with other entities in the domain, it is also important to recover them when needed. The λ-expressions provide a way of defining any relation in terms of an arbitrarily complex graph; all of the complexity, including the nested structures, can be recovered when the definitions are expanded.

Frames have no standard mapping to English, but conceptual graphs that contain only low-level relations do have a direct mapping to English. The expanded version of the frame can be translated to English, line-for-line:

```
TheStructuredIndividualThatIsFredsCar is a Camaro;
it is owned by Fred;
it has as parts a set of entities consisting of
FredsCarsSteeringWheel, FredsCarsLeftFrontTire,...;
and the set of parts is named TheSetOfPartsOfFredsCar.
In the year 1988, it was manufactured,
and it cost $15,000 US.
```

This kind of English may not be elegant, but it is readable enough for a specification document. It would also be suitable for a help facility that would explain the knowledge base by translating relevant excerpts into English. The ability to generate explanations automatically requires a knowledge representation language that has a direct mapping to natural language.

Although frames are convenient for many purposes, they cannot express all possible combinations of negations, disjunctions, and universal quantifiers. Conceptual graphs, however, can represent all of logic. Figure 6 shows the conceptual graph for the sentence *If a farmer owns a donkey, then he beats it*; it corresponds to Peirce's graph in Figure 2 and Kamp's DRS in Figure 3. As in Peirce's form, Figure 6 has a nest of two negative contexts, here explicitly marked with the ¬ symbol. The concepts **[FARMER]** and **[DONKEY]** occur only in the outer context (the *if*-part). To show references to those concepts in the *then*-part, two concepts of type ⊤ (the top of the

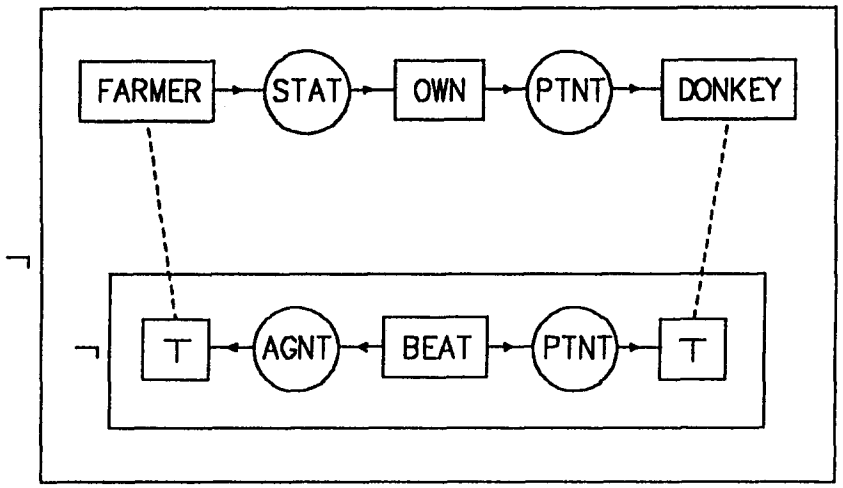


Figure 6. Conceptual graph for "If a farmer owns a donkey, then he beats it."

type lattice) are shown in the inner context. They are linked by dotted lines, called *coreference links*, to the concepts [FARMER] and [DONKEY].

Concepts of type ⊤ typically represent pronouns in English (in this example, *he* and *it*). See the paper by [15] for more detail about pronouns and discourse referents. In the linear form, the dotted lines are represented by variables like *x and *y. The concept [⊤] becomes [⊤:*x], which may be abbreviated as simply [*x]. Figure 6 then becomes

¬[[FARMER: *x]→(STAT)→[OWN]→(PTNT)→[DONKEY: *y]
¬[[*x]←(AGNT)←[BEAT]→(PTNT)→[*y]]].

To improve readability, the symbols IF and THEN are used as synonyms for the negative context markers ¬[; the period at the end represents the two closing brackets ]]. The result is the following graph:

IF [FARMER: *x]→(STAT)→[OWN]→(PTNT)→[DONKEY: *y]
THEN [*x]←(AGNT)←[BEAT]→(PTNT)→[*y].

This linear form can be read directly as an English sentence: *If a farmer x owns a donkey y, then x beats y.* Since the symbols IF and THEN abbreviate Peirce's contexts, the scopes of the quantifiers are correct.

Like the Peirce-Kamp form, the default quantifiers on concept nodes are existential. Unlike the Peirce-Kamp form, conceptual graphs can directly represent all quantifiers expressed in natural language. Consider the sentence *Every farmer who owns a donkey beats it.* The quantifying word is *every*, the restrictor is the phrase *farmer who owns a donkey*, and the scope is the rest of the sentence. To represent that sentence in the Peirce-Kamp form, it would first be transformed to the *if-then* form of Figures 2, 3, and 6. Conceptual graphs, however, allow generalized quantifiers in the referent field. The phrase *every farmer* would be represented by the concept [FARMER: ∀]. To represent the phrase *every farmer who owns a donkey*, the type field must contain a λ-expression that defines the special type of donkey owning farmers:

[(λx) [FARMER:*x]→(STAT)→[OWN]→(PTNT)→[DONKEY]:∀].

The relative clause is included in the λ-expression that defines the concept type, and the quantifier is placed in the referent field. The scope of the quantifier is the entire context in which the concept occurs. Following is the graph for the whole sentence *Every farmer who owns a donkey beats it*:

[(λx) [FARMER:*x]→(STAT)→[OWN]→(PTNT)→[DONKEY]:∀]
←(AGNT)←[BEAT]→(PTNT)→[⊤:#].

The concept [T:#] represents the pronoun *it*. The # referent indicates something in the current context, in this case, the donkey. By the rules for expanding the ∀ quantifier [1] followed by Kamp's rules for resolving pronouns, this graph can be expanded into exactly the same form as Figure 6, which represents the logically equivalent sentence *If a farmer owns a donkey, then he beats it.*

In discourse representation theory, Kamp did not provide means of representing the ∀ quantifier directly. He had to translate the sentence *Every farmer who owns a donkey beats it* directly into the DRS form in Figure 3. That translation is a complex, non-context-free mapping: it requires the quantifier to be translated to the more primitive DRS form during an early stage of semantic interpretation. With the ∀ quantifier and the # symbols in the referent field, conceptual graphs allow the semantic interpreter to proceed in a purely context-free way:

- Each natural language phrase is mapped into a conceptual graph independent of the way it may be nested inside any other structure.
- Then the graphs for the separate constituents are joined to form the interpretation of the sentence as a whole.
- Finally, the quantifiers can be expanded, and the discourse referents can be resolved to form the final semantic interpretation.

By postponing the resolution of anaphora, conceptual graphs provide an intermediate stage that allows other processing to be performed. The simple cases are handled in a way that is equivalent to Kamp's, but complex references can be deferred until further background knowledge is considered.

The actor nodes on conceptual graphs are derived by *skolemization*: whenever an existentially quantified variable **y** depends on a universally quantified variable **x**, it is possible to define a function $f$ from **x** to **y**. To illustrate that principle, consider a formula for *Every employee earns a salary*:

$$(\forall x)(\exists y)\ (\mathtt{employee}(x) \supset (\mathtt{salary}(y) \wedge \mathtt{earns}(x,y))).$$

By introducing a *Skolem function* $f(x)$, the variable **y** and the quantifier (∃**y**) can be eliminated:

$$(\forall x)\ (\mathtt{employee}(x) \supset (\mathtt{salary}(f(x)) \wedge \mathtt{earns}(x,f(x)))).$$

For every employee **x**, the function $f(x)$ determines **x**'s salary. In an Entity-Relationship diagram, the function $f$ corresponds to a diamond node linking the entity type **EMPLOYEE** to the entity type **SALARY**. In conceptual graphs, there are no variables in the pure graph notation. Therefore, there are no variables to eliminate with Skolem functions. Instead, a Skolem function is represented by an actor node bound to the conceptual graph. It is connected with dotted lines to distinguish it from the links of the associated conceptual graph. Figure 7 shows a conceptual graph with its bound actor (the **BENF** relation indicates the beneficiary).

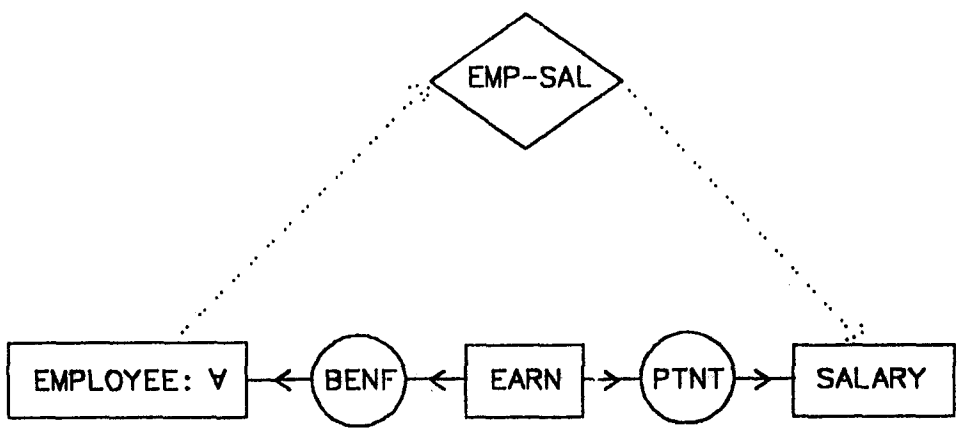


Figure 7. A conceptual graph with a bound actor.

In the linear notation, actors are enclosed in angle brackets to distinguish them from concepts and conceptual relations. To write Figure 7 in a linear form, certain nodes have to be repeated, and variables ***x** and ***y** are needed to show which nodes correspond:

```
[EMPLOYEE: ∀*x]←(BENF)←[EARN]→(PTNT)→[SALARY: *y]
[*x]→<EMP-SAL>→[*y].
```

A function is an $n$-to-one relation. Instead of representing it with arrows, E-R diagrams place **n** and **1** next to the diamond node. That same convention could be used with conceptual graphs:

```
[EMPLOYEE]-n-<EMP-SAL>-1-[SALARY].
```

The conceptual graph expresses the purely logical relationships, and the actors specify procedures or database relations that can compute the values upon request. As this example illustrates, the conceptual graph is derived directly from the English sentence. Then the actor node is derived by Skolemizing the quantifiers on the graph.

## 4. APPLICATIONS OF CONCEPTUAL GRAPHS

Since *Conceptual Structures* appeared, over 200 papers, reports, and books on conceptual graphs have been published, and 137 talks have been presented at the six annual workshops. At least three students have finished Ph.D. dissertations on conceptual graphs, several more are in progress, and many more M.S. projects are either finished or in progress. Implementing a major AI system requires a great deal of work, and the largest efforts have been done as group projects. Following are some of the larger projects, listed in the order in which they were started:

- At the IBM Paris Scientific Center, Jean Fargues designed a conceptual graph system called Kalipsos. It includes a Prolog-like inference engine based on conceptual graphs, a French parser and generator, and tools for knowledge acquisition [25–30].
- At Deakin University in Australia, Brian Garner directed a series of student projects using conceptual graphs. As part of his Ph.D. research, Eric Tsui [31] built a large conceptual graph system; other students extended it and applied it in several M.S. projects. Garner and his group have been among the most prolific authors with over two dozen reports and articles ranging from expert systems applications to natural language parsing and case-based reasoning (see the papers by Garner *et al.* [32–37]).
- At the University of Minnesota, James Slagle has been directing student projects and hosting a series of biweekly workshops on conceptual graphs with graduate students from the university and researchers from Unisys, Control Data, and other companies in the Minneapolis-St. Paul area. Slagle and his students have been working on reasoning with sets and numeric quantifiers [38], situation models for diagnostics [39], and improved techniques for knowledge engineering [40]. The Unisys group has been developing conceptual graph tools [41], representing temporal intervals [42], and applying the tools in expert system development. At Control Data, Murphy and Nagle [43] implemented a blackboard system for computer vision that used conceptual graphs as the language for representing messages.
- At the IBM Rome Scientific Center, Pietro Dell'Orco started a project on using conceptual graphs for an Italian question-answering system. Two professors at Ancona and Rome, Paola Velardi and Maria Teresa Pazienza, have collaborated on it, and Francesco Antonacci has been continuing the project and applying it to information retrieval [44–46].
- At New Mexico State University, Roger Hartley and Michael Coombs developed the technique of *model-generative reasoning* (MGR). The models are generated by joining conceptual graphs—a combinatorial process that requires guidance to avoid exponential overhead. To provide the guidance, they use a method of *conceptual programming*, where the procedures are expressed by actors, which are themselves defined by conceptual graphs. They have applied MGR to problems ranging from robot control to DNA sequence analysis [47–49].

Besides these projects that have been in progress for five years or more, several new projects have been started at other universities around the world. But a listing of projects by title would probably be less informative than a detailed description of just a few. The following projects illustrate the variety of ways of using conceptual graphs, both as a formal system and as a heuristic tool. The selection of these projects for more detailed descriptions by no means implies that the other projects are less significant.

Some of the most instructive examples compare conceptual graphs to other techniques. Smith [50] presented a major commercial project that used conceptual graphs at Reuters in the U.K. The company supplies financial information to dealers, banks, and brokers worldwide. The information is packaged as services delivered through terminals linked in various ways. To configure a package of products and services for each client, Reuters decided to build a system called CAMES (Client AdMin Expert System). They have about 2000 product codes that had to be configured for customer offerings. They succeeded in building a conceptual graph configurator after failing with a rule-based system:

- Reuters originally hired an AI consulting firm to build a prototype using a conventional rule-based expert system shell. But that system was rejected because the maintenance and development effort proved to be unacceptable. They evaluated other rule-based shells and concluded that none of them would have been substantially better.
- The conceptual graph system has a more modular structure, which proved to be much easier to develop and extend. As a result, the CG approach succeeded where the rule-based prototype failed. Reuters has now installed the CAMES system in production use, and it has proven to be faster and more accurate in configuring orders than a group of ten clerks.

This project illustrates several important points: Conceptual graphs succeeded where rule-based expert systems failed; Prolog is a flexible language for implementing shells with reasoning strategies that are different from the usual rule-based languages; and the approach has proven to be practical for a large commercial project.

The original system failed because it was not sufficiently modular: the knowledge about each product was scattered across many different rules; each time a new product was added or changed, several rules had to be changed; and it was impossible to print out and examine all the information that the system had about any particular product. Furthermore, much of the validation process had to be coded in a procedural language, where the information was even more difficult to find, examine, and update. The rule-based prototype required an excessive amount of development and maintenance effort when it reached only 100 products, and Reuters concluded that it would be hopeless to try to extend the system to their full product line. The main reason why the conceptual graph system succeeded is that the knowledge representation was more uniform and modular:

- For each product, there was a central point for finding all the information about the product, displaying it, and maintaining it. The graphs for each product could be examined and updated without considering any other part of the system.
- For each client, the result of the configurator was a graph that described all the products and features at the client's site. As the development continued, extra levels of testing and validation could be added to the system without affecting the information or the methods used to build the graphs.
- Conceptual graphs provided a uniform knowledge representation that could be used in all aspects of the system. The schemata for each product, the definitions for each feature, and the rules for validating the results were all represented in the same notation.
- Actor nodes on conceptual graphs provided a convenient way of accessing external information in a relational database. Since the actors encapsulate the details of how they operate, they could be implemented with a Prolog interface to SQL that was transparent to the graph operations.

For each product, CAMES has several schemata represented as conceptual graphs. Following is a schema for a product of type **BONR-A**:

```
schema for BONR-A(x) is
    [ADD-ON: *x]-
        (HAS)←[SUBSCRIBER]
        (SUPP)→[BONDS: CGBONDS]
        (NEED)→[BASIC]
        (REQT)→[BONDS-DATABASE].
```

This graph says that **BONR-A** is an add-on product that supplies a service **BONDS**, with the service code **CGBONDS**. The relation **NEED** shows that **BON-R** requires the presence of **BASIC**, which is another **PRODUCT**, and **BONDS-DATABASE**, which is a type of **SERVICE**. The same product may have other schemata that represent different views or purposes. Following is another schema for **BONR-A**:

```
schema for BONR-A(x) is
    [PRODUCT: *x]-
      (PART)←[SITE]
      (EXPT)→[CONTROLLER-CODE]-
                  (CODE)←[CONTROLLER]→(ATTR)→[INSTALLED].
```

This graph says that **BONR-A** is a product that is part of a site, and it has an expectation **CONTROLLER-CODE**, which is a code for a **CONTROLLER**, which has attribute **INSTALLED**. Each of these graphs describes one aspect of the product, and the complete description is generated by joining all of them.

The maximal joins that combine schemata preserve type constraints, but they may generate models that violate global constraints. Those additional constraints can also be represented by conceptual graphs, which are projected into the resulting models. The following graph represents the constraint that no two products should supply the same service:

```
¬[[SERVICE]-
      (SUPP)←[PRODUCT: *a]
      (SUPP)←[PRODUCT: *b]→(DFFR)→[*a]].
```

Literally, this graph says that it is false that there exists a service supplied by a product *a and by a product *b that differs from *a. This constraint can be checked by projecting it into the model for the current client. For example, the following graph might have been generated for some subscriber **UK00001**:

```
[SUBSCRIBER: UK00001]-
  (HAS)→[MONR-FF: {#1}@1]-
            (SUPP)→[MONEY-RATES: CGMONR]
            (SUPP)→[FINANCIAL-FUTURE: CGFF=*a],
  (HAS)→[FFA: {#2}@1]-
            (SUPP)[FINANCIAL-FUTURES: CGFF=*a].
```

By projecting the previous constraint into this graph, the following violation is found:

```
[FINANCIAL-FUTURES: CGFF]-
  (SUPP)←[MONR-FF: {#1}@1]
  (SUPP)←[FFA: {#2}@1].
```

This graph can then be translated into the following error message:

**MONR-FF and FFA supply the same service FINANCIAL-FUTURES.**

With conceptual graphs, only 10 constraints were needed to achieve a level of validation that required many more rules and procedures in the original prototype.

Although the schemata represented in conceptual graphs have a strong similarity to frames, they are more flexible. The main difference is the formation rules that allow conceptual graphs to be joined to form larger conceptual graphs. Frames cannot support such a uniform representation: when two frames are linked by a common variable, the result is not a bigger frame, but a pair of two frames. The frames therefore lose the uniform representation that is maintained by the rules for conceptual graphs. That uniformity makes the applications simpler and more modular:

- Local constraints on type matches are automatically enforced by the rules for joining graphs. It is not possible to generate a configuration that violates those constraints, and no extra rules are needed to enforce them.
- Global constraints are enforced by projections into the graphs that result from the joins. Once a graph has been created by a join, the order in which the graphs were joined or the number of joins is irrelevant. Therefore, projections can enforce constraints that would be difficult to implement or even to state in rule-based or frame-based systems.

Another project used conceptual graphs for knowledge acquisition and compared them with the traditional AI techniques of rapid prototyping [40]. The authors had two different teams do knowledge acquisition for an actual audit planning task for an electronics company. The CG team used conceptual graphs as the intermediate specification language, and the W team used a more traditional AI approach with five stages of problem identification, analysis, formalization, implementation, and testing. Both teams took the same amount of time (approximately 140 hours), and both produced a working prototype. But the CG team also had a formal specification in conceptual graphs in addition to the prototype. An independent group rated the process and the results of the CG team better than the W team in almost every respect. Most significantly, the CG team had a more complete implementation, whereas the W prototype "did not deal with many essential issues and cues" and it did not "correctly classify issues." The only area where the CG team was rated lower was that the linear notation for CG's was not considered sufficiently readable by the domain experts. The graphic form, however, was considered quite readable.

After the comparison was finished, another person was able to use the conceptual graph specification to build another prototype in a different language (KEE), even though he had not participated in the original knowledge acquisition task. Since the W team did not produce an independent specification, it was not possible for anyone else to implement another system from their results. This study showed that conceptual graphs are expressive enough to represent the complete application specification. Furthermore, even with only paper and pencil, they improved the quality of the resulting system without delaying the implementation. If the developers had access to the tools of the Reuters project, the conceptual graph specification would have been the implementation.

For the expert system applications, conceptual graphs were used as a system of logic. Their graph structure also supports comparisons and pattern matching for associative and analogical reasoning. Leishman [51] built a system for doing analogies with conceptual graphs and showed that it could support all the classical analogies reported in the AI literature. For metaphor, analogies are important, but Way [52] developed a theory of *dynamic type hierarchies* that would modify the concept types. Understanding poetic language is merely one application; even more important is the ability of metaphor to enrich the knowledge base with new concept types in the hierarchy. As an example, consider the metaphor *My car is thirsty.* This sentence violates the constraint that every instance of **THIRSTY** is an attribute of an **ANIMAL**:

[THIRSTY:∀]←(ATTR)←[ANIMAL].

This metaphor cannot be interpreted by finding a missing relationship between the car and some unstated animal. Instead, the concept corresponding to the car must be linked directly to the

concept [THIRSTY] by the ATTR relation, even though such a link would violate the constraint. To permit that link, Way's system would compare the types ANIMAL and CAR to find their minimal common supertype, MOBILE-ENTITY. Then it would refine the type hierarchy with a new type t that would be a subtype of MOBILE-ENTITY and a supertype of both ANIMAL and CAR.

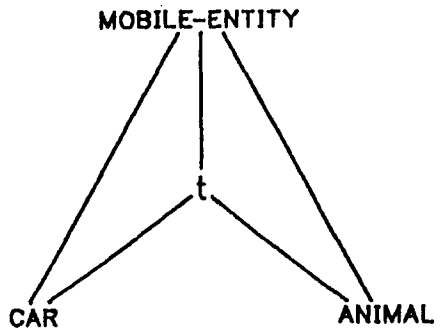


Figure 8. New type t introduced by metaphor.

Figure 8 shows the place for the new type t in the hierarchy, but it does not show the differentia that distinguishes t from the supertype MOBILE-ENTITY. To define t, the system must search for properties of ANIMAL related to THIRSTY:

$$[\text{THIRSTY}] \leftarrow (\text{ATTR}) \leftarrow [\text{ANIMAL}] \rightarrow (\text{STAT}) \rightarrow [\text{REQUIRE}] \rightarrow (\text{PTNT}) \rightarrow [\text{LIQUID}].$$

This graph says that thirsty is an attribute of an animal that requires liquid. The right side of the graph can serve as the body of a $\lambda$-expression that defines t as a mobile entity that requires liquid:

$$\text{t} = (\lambda \text{x})\ [\text{MOBILE-ENTITY: *x}] \rightarrow (\text{STAT}) \rightarrow [\text{REQUIRE}] \rightarrow (\text{PTNT}) \rightarrow [\text{LIQUID}].$$

The new concept type refines the type hierarchy. Then any conceptual pattern for an ANIMAL that needs LIQUID could be generalized to t. Besides saying that a car is thirsty, one could say that it drinks, it stops for a drink, or it guzzles. After these patterns have been generalized to the type t, the standard AI techniques of inheritance would allow any MOBILE-ENTITY that requires liquid to inherit them. Trucks, planes, and ships, for example, could all be called gas guzzlers. This application uses the structural properties of the graphs for supporting analogies, the $\lambda$-expressions for defining new types, and the smooth mapping from natural languages to conceptual graphs.

With the increasing interest in neural networks, some researchers have abandoned symbolic representations for the lower-level neural networks. Lendaris [53,54], however, developed a hybrid learning system that combined conceptual graphs with neural networks. Unlike most neural networks that match inputs directly to outputs, his system used concepts as an intermediate representation. In the first stage, it would match inputs to concepts; in the second stage, it would match conceptual graphs formed from those concepts to the outputs. The hybrid system had a significantly reduced error rate and a faster learning rate than networks that matched inputs to outputs directly.

## References

1. J.F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley, Reading, MA, (1984).
2. H. Kamp, Events, discourse representations, and temporal references, *Languages* **64**, 39–64 (1981).
3. L. Tesnière, *Éléments de Syntaxe Structurale*, Librairie C. Klincksieck, Paris, (1959).
4. I.A. Mel'čuk, *Dependency Syntax*, State University of New York Press, Albany, (1988).
5. D.G. Hays, Dependency theory: A formalism and some observations, *Language* **40** (4), 511–525 (1964).
6. R.C. Schank and L.G. Tesler, A conceptual parser for natural language, *Proc. IJCAI-69*, pp. 569–578, (1969).

7. R.C. Schank and R.P. Abelson, *Scripts, Plans, Goals and Understanding*, Lawrence Erlbaum Associates, New York, (1977).
8. R.C. Schank, *Dynamic Memory*, Cambridge University Press, New York, (1982).
9. S. Ceccato, Automatic translation of languages, presented at the *NATO Summer School*, Reprinted in *Information Storage and Retrieval* **2** (3), 105–158 (1964).
10. C.J. Fillmore, The case for case, In *Universals in Linguistic Theory*, Edited by E. Bach and R.T. Harms, Holt, Rinehart and Winston, New York, pp. 1–88, (1968).
11. J.S. Gruber, *Lexical Structures in Syntax and Semantics*, North-Holland Publishing Co., New York, (1976).
12. W. Wilkins, Ed., *Syntax and Semantics: Thematic Relations*, Academic Press, New York, (1988).
13. A. Church, *The Calculi of Lambda Conversion*, Princeton University Press, Princeton, NJ, (1941).
14. J.F. Sowa, Definitional mechanisms for restructuring knowledge bases, In *Methodologies for Intelligent Systems 5* (Edited by Z.W. Ras), North-Holland Publishing Co., Amsterdam, (1990).
15. J.F. Sowa, Towards the expressive power of natural language, In *Principles of Semantic Networks* (Edited by J.F. Sowa), Morgan-Kaufmann Publishing Co., San Mateo, CA, (1991).
16. J.A. Robinson, A machine-oriented logic based on the resolution principle, *J. of the ACM* **12**, 23-41 (1965).
17. J.F. Sowa, Conceptual graphs for a data base interface, *IBM Journal of Research and Development* **20** (4), 336–357 (1976).
18. C.A. Petri, Kommunikation mit Automaten, Ph.D. dissertation, University of Bonn, (1962).
19. M.R. Quillian, Carnegie Institute of Technology. Abridged version, In *Semantic Information Processing*, (Edited by M. Minsky), Ph.D. dissertation, MIT Press, Cambridge, MA, pp. 227–270, (1966).
20. J.F. Sowa, Definitional mechanisms for conceptual graphs, *Graph Grammars and Their Application to Computer Science and Biology*, (Edited by V. Claus, H. Ehrig and G. Rozenberg), Springer-Verlag, Berlin, pp. 426–439, (1979).
21. J.F. Sowa, Semantics of conceptual graphs, *Proc. of the 17th Annual Meeting of the Association for Computational Linguistics*, pp. 39–44, (1979).
22. R.C. Schank, Ed., *Conceptual Information Processing*, North-Holland, Amsterdam, (1975).
23. J. Barwise and J. Perry, *Situations and Attitudes*, MIT Press, Cambridge, MA, (1983).
24. D.B. Lenat and R.V. Guha, *Building Large Knowledge-Based Systems*, Addison-Wesley, Reading, MA, (1990).
25. J. Fargues, M-C. Landau, A. Duguord and L. Catach, Conceptual graphs for semantics and knowledge processing, *IBM Journal of Research and Development* **30** (1), 70–79 (1986).
26. J. Fargues, Des graphes pour coder le sens des phrases, *Pour la Science* **137**, 52–60 (1989).
27. J.F. Nogier, Generating language from a conceptual representation, *Proc. 9th International Conference on Expert Systems and Their Applications*, 133–144 (1989).
28. J.F. Nogier, *Génération de langage et graphes conceptuels*, Editions Hèrmes, Paris, (1990).
29. J. Fargues and A. Perrin, Synthesizing a large concept hierarchy from French hyperonyms, *COLING-90 Proceedings*, pp. 112–117, (1990).
30. M-C. Landau, Semantic representation of texts, *COLING-90 Proceedings* Vol. 2, pp. 239–244, (1990).
31. E. Tsui, Canonical graph models, Ph.D. Thesis, Division of Computing and Mathematics, Deakin University, Australia, (1988).
32. B.J. Garner, K.E. Larkin and E. Tsui, Prototypical knowledge for case-based reasoning, *Proc. DARPA Workshop on Case-Based Reasoning*, Pensacola Beach, Florida, (1989).
33. B.J. Garner, D. Lukose and E. Tsui, Parsing natural language through pattern correlation and modification, *Proc. of the 7th International Workshop on Expert Systems and Their Applications*, Avignon 87, France, pp. 1285–1299, (May, 1987).
34. B.J. Garner and S. Kutti, Application of dynamic knowledge in designing distributed operating systems, *Proc. of Applications of AI V, Orlando, SPIE Proc. 784* (Edited by J.F. Gilmore), pp. 391–397, (1987).
35. B.J. Garner and E. Tsui, A self-organising dictionary for conceptual structures, *Proc. of Applications of AI V, Orlando SPIE Proc. 784*(Edited by J.F. Gilmore), pp. 356–363, (May, 1987).
36. B.J. Garner and E. Tsui, General purpose inference engine for canonical graph models, *Knowledge Based Systems* **1** (5), 266–278 (1988).
37. B.J. Garner and E. Tsui, Knowledge acquisition and reasoning with a canonical graph model in personal financial planning, In *Expert Systems in Economics, Banking, and Management*, (Edited by L.F. Pau, Y.H. Pao, J. Motiwala and H.H. Teh), North-Holland, Amsterdam, (1989).
38. B. Tjan, D.A. Gardiner and J.R. Slagle, Direct inference rules for conceptual graphs with extended notation, *Proc. 1990 Workshop on Conceptual Graphs*, Boston, (1990).
39. D.A. Gardiner and J.R. Slagle, Situation model creation, *Proc. 1990 Workshop on Conceptual Graphs*, Boston and Stockholm, (1990).
40. J.R. Slagle, D.A. Gardiner and K. Han, Knowledge specification of an expert system, *IEEE Expert* **5** (4), 29–38 (1990).
41. J.W. Esch, Graphical displays of conceptual structures, *Proceedings of the Fifth Annual Workshop on Conceptual Graphs*, pp. 33–42, AAAI, Boston, (1990).
42. J.W. Esch and T.E. Nagle, Representing temporal intervals using conceptual graphs, *Proceedings of the Fifth Annual Workshop on Conceptual Graphs*, pp. 43–52, AAAI, Boston, (1990).
43. M.E. Murphy and T.E. Nagle, Automating interpretation of reconnaisance sensor imagery, *Advanced Imaging*, 19–26 (1988).

44. F. Antonacci, M.T. Pazienza, M. Russo and P. Velardi, Representation and control strategies for large knowledge domains: An application to NLP, *Applied Artificial Intelligence* **2** (3/4), 213–249 (1988).
45. P. Velardi, M.T. Pazienza and M. DeGiovanetti, Conceptual graphs for the analysis and generation of sentences, *IBM Journal of Research and Development* **32** (2), 251–267 (1988).
46. M.T. Pazienza and P. Velardi, Integrating conceptual graphs and logic in a natural language understanding system, In *Natural Language Understanding and Logic Programming II,* (Edited by V. Dahl and P. Saint-Dizier), North-Holland Publishing Co., Amsterdam, (1988).
47. M.J. Coombs and R.T. Hartley, The MGR algorithm and its application to the generation of explanations for novel events, *International Journal of Man-Machine Studies* **27**, 679–708 (1987).
48. M.J. Coombs and R.T. Hartley, Explaining novel events in process control through model based reasoning, *International Journal of Expert Systems* **1** (2), 87–109 (1988).
49. R.T. Hartley and M.T. Coombs, Reasoning with graph operations, In *Principles of Semantic Networks,* (Edited by J.F. Sowa), Morgan Kaufmann Publishers, San Mateo, CA, (1991).
50. B. Smith, CAMES—Expert system administration of money market services, *Proceedings of the Sixth Annual Workshop on Conceptual Graphs*, SUNY Binghamton, Binghamton, NY, (1991).
51. D. Leishman, Analogy as a constrained partial correspondence over conceptual graphs, *Proc. 1st International Conference on Principles of Knowledge Representation and Reasoning,* Toronto, Morgan-Kaufman, San Mateo, CA, pp. 223–234, (1989).
52. E.C. Way, *Dynamic Type Hierarchies: An Approach to Knowledge Representation through Metaphor*, Kluwer Academic Press, (1991).
53. G.G. Lendaris, Neural networks, potential assistants to knowledge engineers, *Heuristics* **1** (2) (1988).
54. G.G. Lendaris, Conceptual graphs as a vehicle for improved generalization in a neural network pattern recognition task, *Proc. 5th Annual Workshop on Conceptual Structures*, AAAI, Boston, pp. 90–91, (1990).