An Implementation Model for Contexts and Negation in Conceptual Graphs

John Esch & Robert Levinson

UCSC-CRL-95-22
May 1, 1995

Unisys Government Systems Group
P.O. Box 64525 U1T23
St. Paul, MN 55164
(612) 456-3947
esch@email.sp.paramax.com

Department of Computer and Information Sciences
225 Applied Sciences Building
University of California
Santa Cruz, CA 95064
(408) 459-2087
levinson@cse.ucsc.edu

Keywords: contexts, negation, conceptual graphs, consistency, inference, retrieval, knowledge representation, lattice

Abstract

An implementation model for a retrieval and inference system based on the theory of conceptual graphs is presented. Several hard issues related to the full implementation of the theory are taken up and solutions presented. The solutions attempt to exploit existing but not fully recognized symmetries in CG theory. These symmetries include those between formation and inference rules, AND and OR, positive and negative, copy and restrict, general and specific, etc. Topics taken up include the implementation of Sowa's formation rules, the storage of a conceptual graph hierarchy involving contexts and negation as a conjunctive normal form (CNF) lattice, the extension of existing retrieval algorithms, such as Levinson's Method III and UDS, to handle complex referents and nested contexts, the checking of consistency, and the definition of Peirce's inference rules in terms of formation rules. A distinction is made between syntactic implication and semantic implication. The issues tackled in the paper lay the foundation for a full scale graph-based first-order logic theorem prover.

Contents

1 Introduction

> At a given level of a hierarchy, a particular system can be seen as an outside to systems below it, and as an inside to systems above it; thus, the status (i.e., the mark of distinction) of a given system changes as one passes through its level, in either the upward or downward direction. The choice of considering the level above or below corresponds to a choice of treating the given system as autonomous or controlled (constrained) [11].

Is your conceptual graphs database consistent? Does it store and process nested context graphs? How is negation handled? Are complex referents too complex to use? Does it properly map the mathematical relationship of Peirce's rules to Sowa's [21, 22, 23] formation rules? Although conceptual graph (CG) theory has been discussed in depth for the past ten years, and several implementations have been developed [5], these full implementation questions have largely been neglected.

In this paper we address some of these omissions. In particular, we give implementation methods for handling negation, arbitrarily nested contexts, the canonical formation rules, and lambda abstraction. Although, we don't give a full scale inference system, we do show how Peirce's inference rules can be incorporated within the same scheme, and how inference might proceed. Along these lines we give a mechanism for ensuring consistency (no logical contradictions) in a CG database. Some of the major contributing notions include:

1. The storing of the CG generalization hierarchy as a CNF lattice that exploits the duality between Boolean AND and OR for storage and retrieval efficiency.
2. The extension of previous implementation models [3, 18] that exploit containment links to avoid repeated storage of structures, to store nested contexts.
3. The association of labels (codes) with nodes and links that indicate whether the positive or negated version is being asserted or inferred.
4. The representation of a CG hierarchy as a hierarchy over equivalence classes of CGs rather than over individual structures as is discussed in [24].
5. A more elegant statement of the canonical formation rules that more clearly distinguishes restriction and join.
6. The treatment of contexts and complex referents in the same unifying manner.
7. The extension of existing retrieval algorithms for exploring the generalization hierarchy to cover the full variety of CGs [16, 3, 4].
8. A formalization of Peirce's inference rules in terms of formation rules.

In Section 2, Formation Rules, we restate the basic rules for transforming one CG to another. These are foundational because the generalization hierarchy/lattice is based on the partial ordering relation induced by the projection operator. For one graph to project into another, it must be possible to derive the second by applying a sequence of formation rules to the first.

Section 3, Lattice Terminology, explains the basic terminology needed to understand and use lattices to store the CG database. Section 4, Adding Contexts, adds complex referents which include contexts. Previous work defining a Universal Data Structure (UDS) to store the CG database [18] did not include complex referents with icons, indexes, and symbols. This section shows how various forms of them, e.g. names, variables, literals, and graphs for contexts, can be added to the generalization lattice. Section 5 is on Adding Negation. It describes how to extend the generalization hierarchy/lattice to cover the negative graphs that were not included in previous work either. In Section 6, Extended Algorithms, the basic structure of the previous Method III algorithm [4, 15] is extended, without increasing the complexity, to cover complex referents, such as names, variables, contexts, and negative graphs. Section 7 covers Peirce Inference Rules. With conjunction and negation it is possible to represent all logical expressions. Peirce defined a compact system of logic that John Sowa applied to CGs. This section restates Peirce's rules and shows how they are handled with the extended generalization hierarchy/lattice. And Section 8 concludes by giving future directions.

2 Formation Rules

The formation rules are a foundational part of CG theory because they define the partial ordering relation used to build the syntactic generalization hierarchy. They were originally defined by Sowa in 1984 [21] and refined in his latest book [23]. We have streamlined and clarified those definitions by defining three kinds of operations on conceptual graphs, equivalence preserving, specialization, and generalization. Each of the formation rules has a mutual inverse: Simplify with Copy, Restrict with Join, and Unrestrict with Detach. Some work in the same spirit is taken up in [2].

In the following, each of the formation rules is defined in terms of more primitive individual concept and relation operations. The notation type(r1) $\leq$ type(r2) is used. It means that r1 is a subtype of r2 and hence more specialized. We also assume that a relation subtype has the the same corresponding arcs as the relation type.

2.1 Equivalence Formation Rules

The ϕ function maps a CG to an equivalent predicate logic expression. So two CGs, mapped by ϕ to logically equivalent expressions, have the same meaning; i.e., they are in the same equivalence class. Consequently, transformation rules, where the initial and final CGs are in the same equivalence class, are called equivalence rules.

We define Relation Simplify, its inverse Relation Copy, Concept Simplify, its inverse Concept Copy, and use them to define Simplify & Copy equivalence rules.

Definition: Relation Simplify. If r1 and r2 are relations, type(r1) $\leq$ type(r2), and each pair of corresponding arcs connects to the same concept or coreferent concepts in the same context, then r2 and its arcs may be deleted.

It can be shown that the initial and final CGs are in the same equivalence class as follows. First, ϕ maps relations to a predicate corresponding to the relation's type. Second, ϕ maps the predicate's arguments to the referents of the concepts to which the relation is connected. Third, since each corresponding pair of arcs connect to the same or coreferent concepts, the corresponding pair of arguments will be the same. Thus, the predicates have the same arguments. Fourth, the predicate for r1 implies that for r2. And fifth, since type(r1) $\leq$ type(r2) and predicate(r1) => predicate(r2), we can delete r2.

Definition: Relation Copy. If relation r1 has arcs connected to concepts c1, c2, ... , cn, then relation r2 may be added where both type(r1) $\leq$ type(r2) and corresponding arcs also connected to c1, c2, ..., cn or concepts coreferent with them in the same context. Note that r2 doesn't have to be connected to the identical concepts, only corresponding coreferent ones in the same context.

Definition: Concept Simplify. If both c1 and c2 are coreferent concepts in the same context[1] and type(c1) $\leq$ type(c2), then concept c2 may be deleted and all relation arcs and coreferent links that were connected to c2 are connected to c1.

This works for the same reasons it works in relation simplify. The function ϕ maps concepts c1 and c2 to monadic predicates corresponding to their type labels. Since they are coreferent, these predicates will have the same argument and, since type(c1) $\leq$ type(c2), the more general predicate is implied and can be deleted.

Definition: Concept Copy. If type(c1) $\leq$ type(c2), any concept c1 may be split into c1 and a coreferent concept c2 in the same context with relation arcs and coreferent links that were connected to c1 connected to either c1 or c2. This is allowed because the conformance operator :: is transitive; that is, if referent x conforms to type c1, written c1::x, then x conforms to any supertype of c1, i.e. c2::x.

Next we combine these primitive operations to define the simplify and copy formation rules as a sequence of the corresponding primitive operations.

Definition: Simplify. The composition of a sequence of relation simplify and/or concept simplify operations on the same graph.

Definition: Copy. The composition of a sequence of relation copy and/or concept copy operations on the same graph.

Simplify and Copy are inverses; the composition sequence is just reversed. They are equivalence rules because they are really only syntactic changes to the graph. They do not change the meaning of the beginning graph, the final graph, or any graph in between because ϕ maps all to equivalent logical expressions.

2.2 Specialization Rules

There are two kinds of specialization, restricting the type or referent, or making sets of concepts coreferent. Restriction makes the type or referent of concepts more specialized. Again, the full restrict rule is defined in terms of a more primitive one.

Definition: Concept Restrict. In general, a concept may be restricted by 1) replacing its type with a subtype, 2) replacing a blank or generic referent with a name, variable, or individual referent, 3) replacing a name with a variable or individual referent, 4) replacing a variable with an individual referent, or 5) adding another name, variable, or individual referent.[2]

Definition: Restrict. A sequence of one or more concept restrict operations on the same graph.

Where restrict operates on the referents of a graph, join operates on the lines of identity structure of the graph. To join two concept nodes is to make them coreferent. They do not have to be physically merged, that's a syntactic operation, a concept simplify. The inverse of join (detach) is not to split two concepts leaving them coreferent (that's a concept copy), it is to sever the coreferent link. Consequently, the real effect of the join and detach operations are to make or unmake coreference links. The join formation rule is defined in terms of the more primitive concept join operation.

[1]They could be coreferent either because they have the same referent or because they have a coreferent link.

[2]The section on Adding Contexts goes into more detail.

Definition: Concept Join. If two concepts in the same context are not coreferent, make them coreferent. If either of the two concepts were part of a larger line of identity, then all concepts in the two lines of identity become part of the same line of identity.[3] [4]

Definition: Join. A sequence of one or more concept joins creating, extending, or merging one or more lines of identity in the same context.

2.3 Generalization Rules

The inverse of specialization is generalization. Consequently, the inverse of each of the specialization rules gives a generalization rule. The inverse of restrict is unrestrict, and the inverse of join is detach. The Opposite of restricting is to broaden, to make the type or referent of concepts more general.[5] Again, the full unrestrict rule is defined in terms of a more primitive one.

Definition: Concept Unrestrict. A concept may be unrestricted (broadened) by 1) replacing its type with a supertype, 2) replacing an individual referent with a variable, name, or blank (generic referent), 3) replacing a variable referent with a name or blank, or 4) replacing a name referent with a blank.

Definition: Unrestrict. A sequence of one or more concept unrestrict (broaden) operations on the same graph. Since join makes concepts coreferent, to be the inverse of join, detach must make them not be coreferent.

Definition: Concept Detach. Making a subset of a line of identity's concepts part of a new, distinct line of identity. The simplest case is to make two concepts in the same context, that are not coreferent with any other concepts, to be not coreferent.

The full detach rule, defined next, is the operation of severing possibly multiple coreferent links either by unlinking or by changing referent variables to be distinct. It's like copying whatever relations one wants, grabbing a handful of relations, pulling them to another part of the page, concept copying all concepts connected to those relations, and concept detaching all the copied concepts.

Definition: Detach. A sequence of one or more concept detach operations which partitions the corresponding lines of identity of one graph into distinct lines.

2.4 Summary of Formation Rules

Detach and Join are inverses of each other. Simplifying two already coreferent concepts does not change the semantics, the mapping to logic is the same. For a specialization to occur, the two concepts couldn't have been coreferent before the join. The old definition emphasized the physical aspect of merging the two concept nodes. However, the real act of joining was to make them concepts of the same referent.

[3]This definition specifically does not required the types and referents to be the same. If the referents were the same, they would already be coreferent, and the join would really be a concept simplify. The real act of joining is to make them coreferent.

[4]As for types, consider the definition of a pet-cat, $TYPE\ pet-cat(pc)\ IS\ [PET : *pc]...[CAT : *pc]$. This is the normal way to specify multiple supertypes. To do it, one has to make concepts with different type labels coreferent. All that doing so implies is that the referent of the two concepts conforms to two types. At the implementation level, all types are converted to bitcodes for fast comparison. [4].

[5]"Unrestrict" is the currently used term; however, in the future, the use of the term "broaden" might be more appropriate.

The CG formation rules can be simplified and clarified by considering simplify and copy to be forming syntactic variants, or equivalence rules. Join and detach are inverses that make and break coreference links, a form of specialization and generalization, respectively. And restrict and unrestrict (broaden) are inverses that specialize and generalize the types and referents of concepts.

3 Lattice Terminology

The CG hierarchy is stored in the form of a lattice based on the formation rules described above. The basic algorithms depend on the mathematical properties of lattices. Techniques for improving algorithm efficiency embed this lattice in a bit encoded lattice where operations are much more efficient [4]. In this section we give a few definitions and apply them to the CG lattice. The sections following this one, on adding contexts and negation, extend the use of the lattice structure.

A partially ordered set or *poset* $\langle P, \geq \rangle$ is a set P and a partial ordering relation $\geq$ over P. The least upper bound, LUB, of elements x and y $\in$ P is written as LUB(x,y). The greatest lower bound, GLB, of elements x and y $\in$ P is written as GLB(x,y). The LUB and GLB of subset $S \subseteq P$ is written as LUB(S) and GLB(S).

Note that the LUB (GLB) may not exist for two reasons: there are no common bounds, or there is not a unique least (greatest) bound. However, when they both exist we have a *lattice*, i.e. a non-empty poset $\langle P, \geq \rangle$ where LUB(x,y) and GLB(x,y) exist $\forall x,y \in P$. A *complete lattice* $L\langle P, \geq \rangle$ is a lattice where LUB(S) and GLB(S) exist $\forall S \subseteq P$. In particular, for $L\langle P, \geq \rangle$ to be complete it must have both a unique *top* element $\top \overset{d}{=}$ LUB(P) and unique *bottom* element $\bot \overset{d}{=}$ GLB(P).

The complete CG generalization lattice is based on the subsumption relation $\geq$ over the set of all graphs; that is $L\langle CG, \geq \rangle$. Graph g1$\in$CG subsumes graph g2$\in$CG, g1$\geq$g2, if there is a mapping π: g1$\rightarrow$g2, where πg1 is a subgraph of g2.[6]

If g1$\geq$g2, g1 is said to be more general that g2 and, conversely, g2 is said to be more specific than g1. The top, $\top$, is the most general graph, i.e. $\top \geq$g $\forall$g$\in$CG. Similarly, the bottom, $\bot$, is the most specific graph, i.e. $\forall$g$\in$CG g$\geq \bot$.

An ancestor of graph g is any graph that is more general than g. Thus, $\top$ is an ancestor of all graphs. A descendent of graph g is any graph that is more specific than g. Thus, $\bot$ is a descendent of all graphs.

A parent of g2$\in$CG is any ancestor g1$\in$CG which is an immediate predecessor of g2, i.e. if g1 = parent(g2), then there doesn't exist a g$\in$CG where g$\neq$g1 and g$\neq$g2 such that g1$\geq$g$\geq$g2. The set of immediate predecessors, IP, of g2 is the set of all parents of g2, i.e. IP(g2) = parents(g2), and g1$\geq$g2 $\forall$g1$\in$IP. A child is defined similarly. The set of immediate successors, IS, of g2 is the set of all children of g2, i.e. IS(g2) = children(g2), and g2$\geq$g3 $\forall$g3$\in$IS.

[6]As used here "subgraph" includes types being subtypes and the possibility of folding, i.e. two or more concepts in the more general graph mapping to the same concept in the specialization graph and relations being simplified. Also, π is not necessarily unique, [T]$\geq$g could map to any of g's concepts. Note that for any graph g$\in$CG, g $\geq$ g.

4 Adding Contexts and Complex Referents

In the black box view of a context [7] it is thought of as a special kind of concept, able to participate in CGs in the same way that any concept can. In the white box view of a context, it is thought of as a special kind of referent. This duality leads to thinking of contexts and concepts as abstraction duals [8, 9]. The idea of substitutability of the defining graph for an individual marker leads to the observation that a concept with a graph or graphs as its referent, called a context, is nothing more than a concept for which an individual has not yet been resolved, else the substitution could be made to simplify the context to a concept. In any case, it is necessary for the CG database to be able to represent contexts; that is, concepts where the referent contains graphs. More generally, it is necessary to represent concepts whose referents contain any of the many other possible combinations of names, variables, literals, graphs, contexts, and nested contexts.

The projection operator π implicitly defines the partial ordering relation $\geq$ that establishes the complete CG lattice. To add contexts to the CG lattice it is necessary to extend π to cover contexts. Extending π starts by adding concepts with various forms of referents. Possible referents are icon, index and symbol. The Concept Restrict Formation Rule maps a generic concept to one with some form of referent. For g1$\geq$g2 the projection operator π currently has the property that, for each concept c in g1, πc is a concept in g2 where TYPE(c)$\geq$TYPE(πc) and, if c is an individual, then REFERENT(c)=REFERENT(πc).

The last part of this needs to be extended for all the different kinds of referents. The extension must be general enough to handle all referent variations. The new definition is: REFERENT(c)$\geq$REFERENT(πc).

The question becomes, which referents are more general that others. A generic concept of some type T is more general than one of the same type with a referent, so: [T]$\geq$[T:name], [T]$\geq$[T:*x], [T]$\geq$[T:#1], [T]$\geq$[T:graph], and [T]$\geq$[T:graph].

A key idea is that combinations of icons, indexes, or graphs, are more specific that any of them individually, so: [T:name]$\geq$[T:name*x], [T:*x]$\geq$[T:name*x], [T:*x]$\geq$[T:*x#1], [T:#1]$\geq$[T:*x#1], [T:name1]$\geq$[T:name1 name2], [T:name2]$\geq$ [T:name1 name2], [T:graph1]$\geq$[T: graph1 graph2], [T:graph2]$\geq$[T: graph1 graph2], [graph1]$\geq$[graph1 graph2], and [graph2]$\geq$[graph1 graph2].

To summarize, every generic concept/context has, potentially, under it in the generalization lattice a sublattice consisting of all possible combinations in any order of icons, indexes, and graphs with the bottom being, implicitly, all of them together.[7] Similarly, every graph has, potentially, under it something like the cross product [6] of these individual sublattices with a common bottom where each concept of the graph has all possible referent values.

A *context* is a concept that has one or more graphs as its referent. Each of them is an immediate predecessor of the context. Thus, IP([T: g1 g2]) = {[T] g1 g2}. To show how contexts fit into and extend the CG lattice, we add to an example commonly used [4, 14]. It involves the 7 graphs given below. The lattice that results from inserting these 7 graphs into a CG database utilizing the original Method III algorithm is shown in Figure 4.1 (left).

G1: [PERSON]$\leftarrow$(AGNT)$\leftarrow$[EAT]
G2: [GIRL]$\leftarrow$(AGNT)$\leftarrow$[EAT] $\rightarrow$(MANR)$\rightarrow$[QUICKLY]
G3: [:Sue]$\leftarrow$(AGNT)$\leftarrow$[EAT] $\rightarrow$(OBJ)$\rightarrow$[PIE] $\rightarrow$(CONT)$\rightarrow$[APPLES]
G4: [:Sue]$\leftarrow$(AGNT)$\leftarrow$[EAT]-

[7]These concept bottoms are also context bottoms and can be use, in a bit encoded lattice, to limit the search to the context.

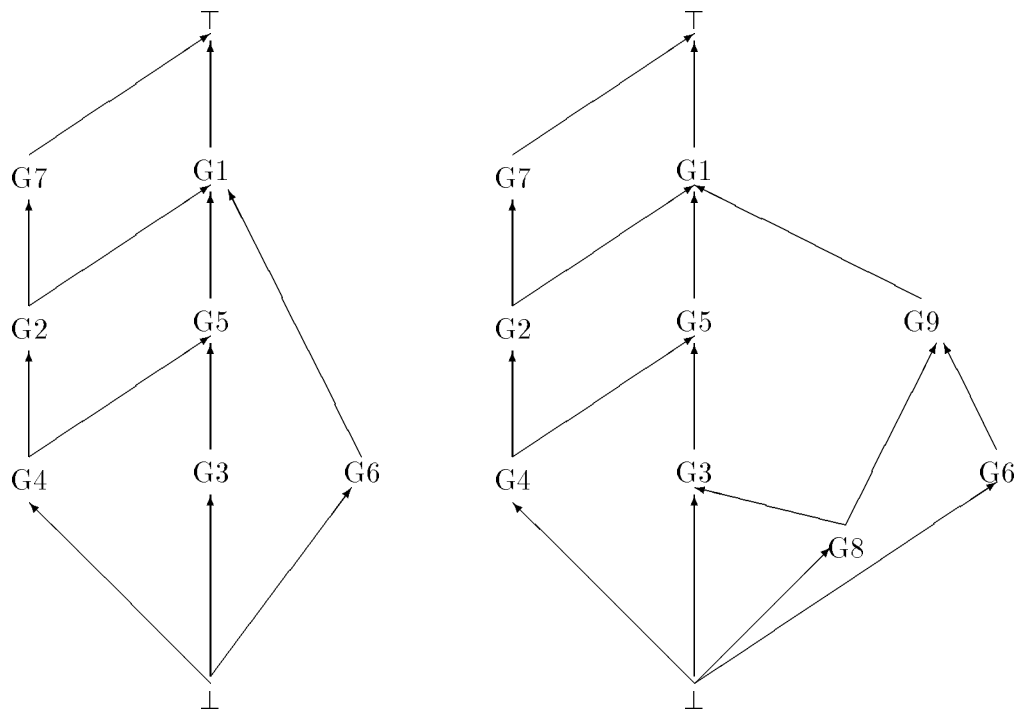

Figure 4.1: Lattice Of Initial Graphs. Lattice With Graph Inserted.

→(OBJ)→[PIE]
→(MANR)→[QUICKLY]
G5: [:Sue]←(AGNT)←[EAT]
G6: [:Dan]←(AGNT)←[EAT]→(OBJ)→[PIE]-
→(CONT)→[APPLES]
→(POSS)→[SUE]
G7: [EAT]→(MANR)→[QUICKLY]

We add to this set a graph containing a context. G8: [:Daniel]→(BELIEVE)→[
[:Sue]←(AGNT)←[EAT] →(OBJ)→[PIE] →(CONT)→[APPLES]
[:Dan]←(AGNT)←[EAT] →(OBJ)→[PIE] →(CONT)→[APPLES]]

When G8 is added to the previous CG database with the new algorithm, it results in the creation of G9, for Dan eating apple pie, and the lattice shown in Figure 4.1 (right).

Another key idea is how to handle *coreference*. John Sowa [21] handled the problem of coreference by propagating aliases down each line of identity from all dominant concepts. For example, [T1:*x]...[T2:*y] becomes [T1:*x *y] [T2:*y *x].[8]

This causes all dominated concepts to acquire multiple indexes, which places them lower in the generalization lattice than the corresponding concept without the coreferent link. That is, T1:*x]≥[T1:*x]...[T2:*y] and [T2:*y]≥[T:*x]...[T:*y].

The impact on the query/insert algorithm, of handling complex referents, is minimal because all these complex referent checks are part of the isomorphism test. In Method V, once bindings are found, they are propagated. [18]

[8]The original syntax used = signs as in [T:*x=*y].

5 Adding Negation

The best way to think about how CGs are stored, when negative contexts are included, is as a lattice over the syntactic structure of graphs, not as all possible semantic implications. The immediate predecessor, IP, and immediate successor, IS, links remain as described in the Lattice Terminology section. Since a node's IS links are part of its children's IP links, it is only necessary to consider either the IP or IS links. Below we concentrate on extending the IP links to cover negative contexts.

5.1 Representing Negative Assertions

To implement negation two bits, PA and NA, are added to each node to indicate whether the graph represented by the node has been asserted positively, negatively, both, or not at all. The interpretation of PA and NA is 00 - No Assertion, 10 - Positive Assertion, 01 - Negative Assertion, and 11 - Both Asserted (a logical inconsistency).[9] Graphically, these are shown as nothing, +, -, & +- in Figure 5.1.

It is natural in this system to represent arcs from A to NOT B. To extend the IP links to cover negative contexts, an IP arc from a node to a parent is labeled + when the positive version of the parent projects into the node, and labeled - when the negative version of the parent projects into the node. To represent the assertion $\neg$A, where A is some graph, graph A is added to the hierarchy and its negative assertion bit set. To represent a term $\neg$A, that is part of some other graph B, the IP link from B to A is labeled negatively. In Figure 5.1, nodes G10, G13 & G14 are negative assertions and nodes G3, G4, G11 & G12 have links as negative terms.

The full Boolean logic lattice has nodes for all possible disjunctions and conjunctions. Since the lattice is symmetric, each negative conjunct has a corresponding complementary disjunct. Thus, if each conjunctive node can be optionally negated, only half of the lattice is needed to represent all possible logical expressions.

Our proposal is to represent each graph in conjunctive normal form (CNF); that is, as conjuncts of the graph's IPs which may or may not be negated in the conjunction. Disjunctions are represented by allowing negative graphs.

Since one graph's IP link is another graph's IS link, we can think of labeled, bi-directional links between a parent and child; that is, between a graph (the child) and a part of its conjunct (the parent).

If a link is labeled +, the positive version of the parent is an IP of the child. This implies that the child has, as part of its conjunction, the positive version of the parent into which the parent projects to satisfy the subsumption relation.

If a link is labeled - , the negative version of the parent is an IP of the child. This implies that the child has, as part of its conjunction, the negative version of the parent into which the negative version of the parent projects to satisfy the subsumption relation.

[9]In some logics a contradiction does not undermine the integrity of the entire database and there are other more "avante gard" solutions, such as not letting the contradiction be used in proofs in which it is not relevant. [10]

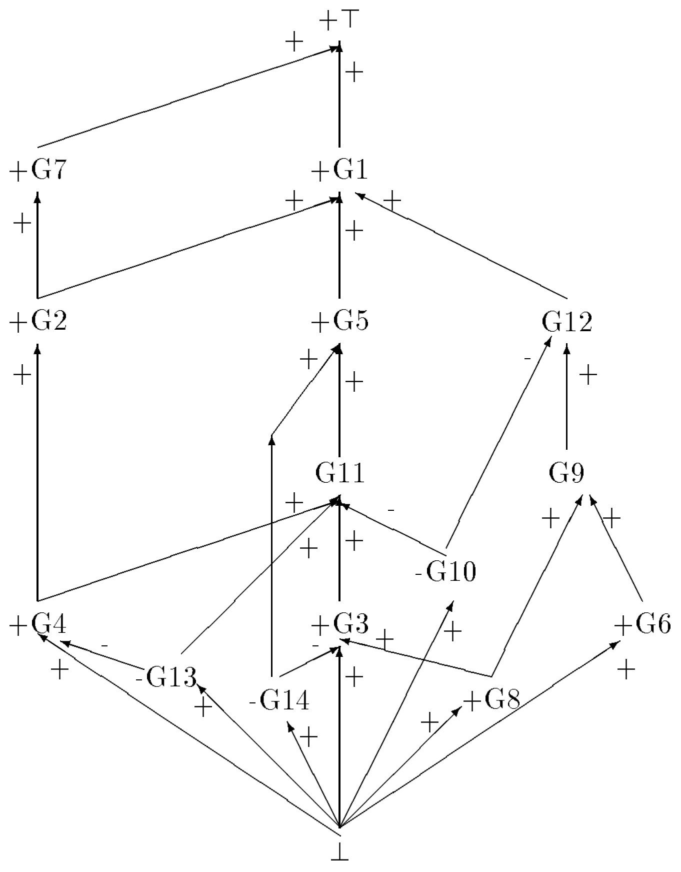

Figure 5.1: Lattice With Negative Graphs Inserted.

5.2 Negation Examples

The following example of disjunction involves negated contexts containing graphs. We use DeMorgan's Law to convert AvB to ¬(¬A&¬B). For example, the disjunction G10: [:DISJ{Sue,Dan}]←(AGNT)← [EAT]→(OBJ)→[PIE] becomes
¬[¬[G11] ¬[G12]] where G11: [:Sue]←(AGNT)← [EAT]→(OBJ)→[PIE] and
G12: [:Dan]←(AGNT)← [EAT]→(OBJ)→[PIE].

When G10 is added, it requires the insertion of G11 and G12 before G10 can be inserted as shown in the Figure 5.1. Note the - signs beside the links from G10 to G11 & G12 and the assertion indicators before each graph's label.

Implication also involves negation because the graph A⇒B is represented as ¬[A ¬[B]]. For example, the graphs G13: ¬[G11 ¬[G4]], for "If Sue eats pie, Sue eats pie quickly.", and G14: ¬[G5 ¬[G3]], for "If Sue eats, Sue eats apple pie." are also shown in Figure 5.1.

5.3 Distinguishing Syntactic and Semantic Lattices

At this point we would like to make a distinction between "syntactic" and "semantic" lattices over a set of CGs. The *syntactic lattice* is the structure implied due to the graphs, formation rules, and Peirce inference rules alone. It is formed from those links that can be inferred directly from two graphs, the formation rules, and Peirce's inference rules without regard to any other graphs in the lattice.

The syntactic lattice is true (and may be assumed such) in all domains because the links do not actually depend on domain knowledge. Which portion of the lattice is actually stored will depend on efficiency considerations.[10]

The *semantic lattice* adds inferences that require three or more graphs to establish and are not implied by the transitivity of the syntactic lattice. For example, given A, B and $A \Rightarrow B$ as graphs, the relationship between A and B is only known given the third graph, $A \Rightarrow B$. In other words, suppose neither A nor B have been asserted but $A \Rightarrow B$ has been asserted. Then both A and B will have assertion bits 00, neither positive or negative assertion. On asserting A later, one can think of B as a ramification and mark it as being inferred without adding any new links.

The syntactic lattice identifies links based on structure, formation, and Peirce rules.[11] These links are true regardless of the domain. They depend only on graph structures and not on what has been asserted and not asserted.[12] In the syntactic lattice a link means "this link holds in all databases".

Where the formation rules are foundational for the syntactic lattice, the inference rules are foundational for the semantic lattice because they define the relationship that establishes the semantic lattice. The semantic lattice models implications among graphs stored in the database. These implications are identified by using the truth values that have been asserted for the graphs, the implications directly stated (e.g. in the graph for $A \Rightarrow B$), and Peirce's inference rules. Here an inference means "given what has been asserted (axioms), this graph holds (is a theorem)". The most efficient computation of the semantic lattice is an important research question to be addressed in the future.

The two lattices may be stored together by adding two inference bits, PI & NI, to each node indicating whether the graph represented by the node has been inferred positively, negatively, both, or not at all. The interpretation of PI and NI is 00 - Not Inferred, 10 - Positive Inference, 01 - Negative Inference, and 11 - Both Inferred (a logical inconsistency).[13] [14]

6 Augmented Algorithms

In this section we describe how existing algorithms for operating on the generalization hierarchy may be augmented to handle contexts, negation, and assertion.

The algorithms described here search the CG lattice for where a new graph Q, called the query graph, fits into the lattice. The algorithm is divided into five phases. Phase 0 deals with Q's negation and assertion. Phase I finds the immediate predecessors of Q, IP(Q). Phase II finds the immediate successor of Q, IS(Q). Phase III inserts Q in the CG lattice. And Phase IV propagates implications. The consequence of finding IP(Q) and IS(Q) is that each $\text{IS}(Q) \Rightarrow Q \Rightarrow$ each IP(Q).

[10] An assumption is that $\neg\top = -$, but this is not shown. Also, by representing both nodes, one can have different encodings for them to improve computational efficiency.

[11] A subsequent section defines Peirce's inference rules in terms of the formation rules which define the projection operator and, consequently, the partial ordering relation. Thus, Peirce's inference rules are contained within the definition of projection and fit into the syntactic lattice.

[12] They also depend on the equivalence classes since only one representative from each equivalence class need be stored.

[13] It is now possible to have other inconsistencies, like Positive Assertion/Negative Inference

[14] A coding scheme that combines the Assertion and Inference bits is possible.

The algorithm considers graphs in topological sort order [4] by using a combination of a FIFO queue called C and the addition of a depth to each graph node. The depth is one greater than the largest depth of its immediate predecessors, IPs. The function pop(C) returns the next graph from C. If IS(X) is the set of immediate successors of graph X, then IS'(X) returns the members of IS(X) which have depth one greater than X. The procedure push(C, SET) pushes set SET onto the FIFO queue C. The augmentation is to the original Method III algorithm [4, 14] to add Phase 0, the greatest lower bound (GLB) test, negation checking, and Phase IV.

```
Phase 0: Handle negation and assertions.
   1. Set Q = Q minus pairs of leading ¬'s.
   2. IF Q is being asserted AND Q = ¬Q' THEN Q = Q' AND complement the assertion.
   3. IF Q asserted THEN PA = 1 OR IF ¬Q asserted THEN NA = 1.
Phase I: Find IP(Q), the immediate predecessors of Q.
   4. IF Q = −, THEN RETURN IP(−).
   5. INITIALIZE C := ⊤ AND S := {}.
   6. WHILE not empty(C) DO X := pop(C) AND
      IF All parents of X are predecessors of Q AND
        GLB(S ∪ {X}) ≠ − AND X is a predecessor of Q (isomorphism test)
      THEN mark X as a predecessor of Q;[15]  S := S ∪ {X} - IP(X);[16]  push(C,IS'(X)).
   7. RETURN S with negative links marked.
Phase II: Find IS(Q).
   8. INITIALIZE S := {} AND C := {} AND Y := some element of IP(Q).
   9. I := intersection of the successor sets of each element of IP(Q) except Y.
  10. FOR each successor X of Y in topological order DO
      IF X is in I and X is a successor of Q (isomorphism test) THEN S := S ∪ {X}.
      Eliminate successors of X from the rest of the FOR loop.
  11. RETURN S with negative links marked.
Phase III: Update IP & IS Nodes (possibly inferring Q).[17]
  12. For each x in IP(Q) DO IS(x) := IS(x) ∪ {Q} - IS(Q) AND
      IF x negatively asserted or inferred, THEN negatively infer Q.
  13. For each x in IS(Q) DO IP(x) := IP(x) ∪ {Q} - IP(Q) AND
      IF x asserted or inferred, THEN infer Q.
Phase IV: Semantic Update.
  14. IF Q asserted THEN infer all the IPs of Q[18] ANDIF
      Q is A⇒B[19] AND A is true OR Q⇒B is true THEN infer B.
  15. IF ¬Q asserted THEN negatively infer all the ISs of Q ANDIF
      B⇒Q is true THEN negatively infer B.
  16. FOR EACH newly inferred node DO Phase IV as if the node had just been asserted.
```

[15] Based on the isomorphism test, this may be a negative link mark.

[16] Notationally, - stands for set subtraction.

[17] A graph to be inserted, that is not known to be true, will be true if it is more general than a graph that is known to be true or, if it can be derived from a sequence of inferences based on one or more asserted graphs.

[18] Because we are storing CGs in CNF, if a CG is true, then all of its IPs must be true.

[19] By "Q is A⇒B" we mean Q is an implication and, if true, can be used to do inferences.

The algorithm above uses only the syntactic links stored in the lattice. After insertion is completed and the links fixed, semantic update, Phase IV, ensues. It is based on both the new syntactic links formed and the implications of the graph inserted. Phase IV will change inferred bits to be on for graphs that can now be inferred which were not known to be inferred previously. It should be pointed out that Phase IV does not absolutely complete the semantic lattice, but represents the transitive closure of the implications that have been identified. Many inferences remain due to Peirce's inference rules.

The algorithm also needs to be modified slightly for nested contexts. For them the graphs are inserted "inside out" with the deepest nested graphs inserted first, and concept variables obtained, with outer levels referring to inner levels via these variables. When comparing two contexts to each other, the concept variables are compared identically to the way type labels are compared in the type hierarchy.

Just as bitcodes may be given to members of the type hierarchy to allow comparison to take place in constant time, bitcodes may be assigned to members of the graph hierarchy itself. If a graph structure is repeated multiple times in a nested context graph, and this correspondence is known (possibly due to the interface), the query should be organized to ask this graph only once.

7 Peirce Inference Rules

So far we have extended the CG generalization hierarchy/lattice and algorithms to include contexts, other complex referents, and negation. It is now possible to use the CG lattice to perform inference. Peirce defined a graphical system of logic and inference rules upon which CG theory is based. Peirce's inference rules provide a basis for defining a corresponding simple set of inference rules for the CG lattice.

The formation rules defined in Section 2 applied to graphs in the same context. These are augmented by cross context formation rules which consist of join and detach across context domination boundaries.[20]

Whether these cross context formation rules are generalizations or not depends on the even or oddness of the dominant concept(s). For example, it's not enough to detach from even context to even context. That's because it's the even or oddness of the context containing the dominant concept of that line of identity that counts. The basic idea of the Peirce rules is to allow equivalence changes in any context, generalization of even contexts, and specialization of odd contexts. The problem is that, depending on the even or oddness of the contexts involved, the generalization formation rules are not always generalizations. In other words, it is not possible for each formation rule to always fall into only one category, equivalence, generalization, or specialization, when multiply nested, possibly negated contexts are involved.

Every structure in the lattice is of even or odd parity, with its negation taken as the opposite parity. As one traverses to deeper levels of nesting, the parity (and hence which inferences are possible) changes with each negation step. Higher level nested structures may be able to exploit inference links previously formed between lower structures without having to rediscover these inferences.

[20] "Domination" is a key word here since contexts are strictly nested and lines of identity must follow the same nesting.

With the above in mind, here's a restatement/merge of the Peirce rules in John Sowa's 1984 book [21] and in the draft of his new book [23] using our formation rules.

Definition: Erasure. An evenly enclosed context may be generalized by applying the following formation rules to one or more of the graphs contained in the context: erasure, unrestrict, or detach from evenly enclosed dominant concepts.

Definition: Insertion. An oddly enclosed context may be specialized by applying the following formation rules to one or more of the graphs contained in the context: insertion, restrict, or join to oddly enclosed dominant concepts.

Definition: Iteration. A copy of all or part of any graph may be inserted into the same or a dominated context. Note that this is a coreferent copy.[21]

Definition: Deiteration. Any graph or part of a graph which could have been the result of an iteration, may be simplified.

Definition: Double Negation. Two nested negated contexts with nothing between them may be drawn around or removed from around any graph, set of graphs, or context.

8 Conclusions and Summary

The methods and implementation model described in this paper provide the ground work for a full first-order logic theorem prover based on conceptual graphs. The ground work laid includes the handling of negation, nested contexts, complex referents, and consistency checking. While we have clearly defined a distinction between syntactic and semantic implication, with the former corresponding to Sowa's conception of a generalization hierarchy, the most efficient algorithm for discovery of the semantic links remains an important topic for future work. Along similar lines, while we have a mechanism for discovering when syntactically-derived contradictions have occurred, we are not yet guaranteed of finding any inconsistencies that also may require semantic links other then a brute force generation of all implied graphs of a given size. A definition of Peirce's inference rules based directly on the implemented data structure also remains to be completed.

The syntactic lattice provides a mechanism for compilation of powerful macros and proof structures for use in all domains so that a theorem prover can improve with experience. In Lisp-based EG theorem provers developed for a class project, we negate queries and attempt to validate them by deriving an empty context on the sheet. This process is accelerated over time by storing (remembering) graphs from which we have already derived the empty context.

In future work, we also hope to improve (as discussed above) the ability to update the semantic links and, along similar lines, to exploit better the abstractions imposed by the semantic equivalence of two graphs. The ability of the retrieval algorithms to find all syntactic implications of graphs at any contextual level provides exactly the information needed to employ higher-order inference rules. Our other work on heuristic search and machine learning [12] may also be able to improve the efficiency of the inference system.

It is our hope that the implementation model in this paper will extend the scope of discussion in the CG community to include the hard issues that we have ignored up until present. It is also hoped that the beauty and elegance of logical inference based on graphs,

[21]The following is a different wording that does not have quite the same meaning and, consequently, needs to be evaluated: "The result of a copy formation rule may go into the same or a dominated context." (Note that this may not be true since the copy formation rule is not strictly a copy.

first envisioned by Peirce, will be more deeply appreciated and complete implementations of his theory, such as incorporating our approach into the Peirce Conceptual Graphs Workbench [5], will soon come to pass.

References

[1] H. Boley. Pattern associativity and the retrieval of semantic networks. *Computers and Mathematics with Applications*, 23(6-9):601–638, 1992. Part 2 of Special Issue on Semantic Networks in Artificial Intelligence, Fritz Lehmann, editor.

[2] M. Chein and M.L. Mugnier. Conceptual Graphs: Fundamental Notions In *Revue d'Intelligence Artificielle*, n14, 1992, pp365-406.

[3] G. Ellis. Compiled hierarchical retrieval. In E. Way, editor, *Proceedings of Sixth Annual Workshop on Conceptual Structures*, pages 187–208, SUNY-Binghamton, 1991.

[4] G. Ellis. Efficient retrieval from hierarchies of objects using lattice operations. In G. Mineau and B. Moulin, editors, *Proceedings of First International Conference on Conceptual Structures (ICCS-93)*, Montreal, 1993.

[5] G. Ellis and R. Levinson. Proceedings of the Fourth International Workshop on Peirce: A Conceptual Graph Workbench. In association with the *Second International Conference on Conceptual Structures*, ICCS'94, College Park, Maryland, USA, August 19, 1994.

[6] G. Ellis and F. Lehmann. Exploiting the Induced Order on Type-labeled Graphs for Fast Knowledge Retrieval. In W.M. Tepfenhart, J.P. Dick and J.F. Sowa editors *Conceptual Structures: Current Practices*, the proceedings of the Second International Conference on Conceptual Structures, ICCS'94, College Park, Maryland, USA, Springer-Verlag, August, 1994.

[7] J.W. Esch. Contexts as White Box Concepts. In *Supplemental Proceedings of First International Conference on Conceptual Structures: Theory and Application*, Quebec City, Canada, August 4-7, 1993, pages 17-29.

[8] J.W. Esch. Contexts and Concepts: Abstraction Duals. In em Proceedings of Second International Conference on Conceptual Structures, ICCS'94, College Park, Maryland, USA, August 1994, pages 175-184.

[9] J.W. Esch. Contexts, Canons and Coreferent Types. In em Proceedings of Second International Conference on Conceptual Structures, ICCS'94, College Park, Maryland, USA, August 1994, pages 185-195.

[10] B. Goertzel. Chaotic logic : language, thought, and reality from the perspective of complex systems science In *IFSR International Series on Systems Science and Engineering* v.9., Plenum Press, New York, 1994.

[11] J.A. Goguen and F.J. Varela. Systems and Distinctions: Duality and Complementarity. In *International Journal of Systems*, 5(1):31-43, 1979.

[12] J. Gould and R. Levinson. Experience-based adaptive search. In *Machine Learning:A Multi-Strategy Approach*, volume 4. Morgan Kauffman, 1992.

[13] R. Levinson. A self-organizing retrieval system for graphs. In *AAAI-84*, pages 203–206. Morgan Kaufman, 1984.

[14] R. Levinson. Pattern associativity and the retrieval of semantic networks. *Computers and Mathematics with Applications*, 23(6-9):573–600, 1992. Part 2 of Special Issue on Semantic Networks in Artificial Intelligence, Fritz Lehmann, editor. Also reprinted on pages 573–600 of the book, Semantic Networks in Artificial Intelligence, Fritz Lehmann, editor, Pergammon Press, 1992.

[15] R. Levinson and G. Ellis. Multilevel hierarchical retrieval. *Knowledge-Based Systems*, 1992.

[16] R. Levinson and Karplus K. Graph-isomorphism and experience-based planning. In D. Subramaniam, editor, *Proceedings of Workshop on Knowledge Compilation and Speed-Up Learning*, Amherst, MA., June 1993.

[17] R.A. Levinson. Exploiting the physics of state-space search. In *Proceedings of AAAI Symposium on Games:Planning and Learning*, pages 157–165. AAAI Press, 1993.

[18] R. Levinson. UDS: A Universal Data Structure. In *Proceedings of Second International Conference on Conceptual Structures*, ICCS'94, College Park, Maryland, USA, August 1994, pages 230-250.

[19] D.D. Roberts. The existential graphs. In *Semantic Networks in Artificial Intelligence*, pages 639–664. Roberts, 1992.

[20] E. Sciore. A complete axiomatization for join dependencies. *JACM*, 29(2):373–393, April 1982.

[21] J.F. Sowa. *Conceptual Structures.* Addison-Wesley, 1984.

[22] J.F. Sowa. Conceptual Graphs Summary. In *Conceptual Structures: Current Research and Practice*, T. Nagle et. al. Ed., Ellis Horwood, 1992.

[23] J.F. Sowa. Knowledge Representation: Logical, Philosophical, and Computational Foundations. Book in preparation. To be published by PWS Publishing Company, Boston, Massachusetts.

[24] M. Willems. Generalization of Conceptual Graphs. In *Proceedings of Sixth Annual Workshop on Conceptual Graphs*, Binghamton, New York, State Univ. of New York, 1991.