Unification and Constraints over Conceptual Structures

Dan R. Corbett
B.S. Information and Computer Science, U. California, Irvine
M.S. Computer Science, Wright State U.

Department of Computer Science
The University of Adelaide

Thesis submitted for the Degree of Doctor of Philosophy

July, 2000

Contents

List of Figures

Table of Abbreviations and Notation

CG	Conceptual Graph
T_C	Concept Types
T_R	Relation Types
C	The set of concepts
R	The set of relations
$\forall$	Universal quantification
$\in$	Member of (a set)
$\leq$	Subtype or subsumption relation
$\top$	Top
$\bot$	Bottom
$\wedge$	Meet operator
$\vee$	Join operator
$::$	Conformity Relation
$arg_i(r)$	i-th argument of the relation *r*
$arity(r)$	Number of arguments that the relation *r* has
$\geq$	Subsumption relation
glb	Greatest Lower Bound
$\sqcap$	Greatest Lower Bound
lub	Least Upper Bound
$\sqcup$	Least Upper Bound
π	Projection Operator
$a \uparrow b$	maximum(a, b)
$a \downarrow b$	minimum(a, b)
$[x, y]$	The real interval between x and y, where: $[x, y] \subseteq \mathbb{R} \times \mathbb{R} : x \leq y$
q	The head node of a CG, a distinguished member of C (chapter 3)

mgu	Most General Unifier (Chapter 3)
$\lfloor$	Restriction
$\cup$	Set union
$\rightarrow$	Projection
$:=$	Value assignment
$\equiv$	Equivalence

Summary of Thesis

This thesis continues the work done in [Corbett 2001; Corbett 2000; van Zyl and Corbett 2000a; van Zyl and Corbett 2000b; Corbett 1999; Corbett and Woodbury 1999; Corbett and Burrow 1996].

This thesis addresses two important areas in the field of Conceptual Structures. The first is the unification of Conceptual Graphs (CGs), and the consequent work in projection and in type hierarchies.

Other researchers have described methods for combining the knowledge in two Conceptual Graphs. Some of these methods work in a limited sense, but none are general solutions for the fundamental problem of Conceptual Graph unification. The first major contribution of this thesis is the development, analysis, and experimental verification of an algorithm that enables the implementation of a general solution to Conceptual Graph unification.

The second important area of investigation is the definition of constraints, especially real-value constraints on the concept referents, with particular attention to handling constraints during the unification of Conceptual Graphs.

The approach presented in this thesis solves the unification-with-constraints problem by solving several subproblems. First, the definition of a real-value constraint type hierarchy is defined for Conceptual Structures. This hierarchy allows the bounding of real values in concepts, and the expressibility of real numbers. Next, many inconsistent and incomplete definitions in the literature are redefined or formalized. New definitions of projection, unifier, Most General Unifier and real numbers create a consistency with the previous canon of Conceptual Graph formalisms. Finally, the concept of knowledge conjunction is explored. Knowledge conjunction is a system in which two pieces of partial information can be combined into a single unified whole. Unification is defined as an operation that simultaneously determines the consistency of two pieces of partial or incomplete knowledge, and if they are consistent, combines them into a single result.

The experiments with the software implemented from the techniques defined here show these techniques to be useful and efficient. The unification algorithm is shown to be sound, complete, and have a complexity of O(n), where n is the total number of relations in both graphs. This is a significant improvement on previous work in that previous efforts at unification of Conceptual Graphs either restricted unification to a simpler operation, or had a high complexity (in fact, NP-complete in the most elaborate system).

The algorithm defined here produces not only a canonical graph, but one which is valid in the domain. The algorithm guarantees that, for any domain, once the domain hierarchies and canonical formation rules have been properly defined, the only structures that will be allowed will be those that represent reasonable knowledge in the domain.

In the last two technical chapters, the unification model presented here is put into the context of knowledge representation and manipulation by comparing it to other methods, including a tree isomorphism method and a unification with variable-arity relations method, which are also defined in this thesis. The final conclusion is that unification is an important tool for Conceptual Graphs, but must be viewed in relation to other useful tools which already exist for CGs.

List of Results

1. Formal definition of real numbers as concept referents in CGs. (section 3.2)
2. Formal definition of real-valued constraints for CGs. (section 3.3)
3. Redefining standard definitions, so that constraints work within the CG formalism:
    - Conceptual Graph (definitions 2′ and 2′′)
    - Projection (definition 4′)
    - Greatest Lower Bound (definition 5)
    - Subsumption as Projection (definition 6)
4. Eliminated inconsistencies in the definition of Most General Unifier (sections 1.4 and 2.2)
5. Investigation of the relation between Constraint Satisfaction Problems, unification in general and CG unification with constraints (sections 2.4 and 2.5)
6. Investigation of the relationship between unification (in general) and knowledge representation schemes (sections 2.2 and 2.6)
7. Study of unification (in general) as reasoning and theorem resolution (section 2.5)
8. Introduce the concept of knowledge conjunction for CGs (section 2.5, 2.6)
9. Formal definition of CG unification (section 3.5)
10. Investigation of completeness of CG unification (section 3.5, theorem 6)
11. Investigation of soundness of CG unification (section 3.5, theorem 5)
12. Investigation of the complexity of CG unification (section 3.5, theorem 3)
13. Implementation of CG unification algorithm (section 4.1 and 4.2)
14. Study showing usefulness of CG unification for the design domain (section 4.2)

15. Demonstrations of CG unification on simple structures (section 4.3)
16. Demonstrations of the limits of CG unification (section 4.4)
17. Comparisons of CG unification to other standard techniques (section 4.5)
18. Definition of a framework for comparing CG unification to other techniques for merging knowledge in CGs by complexity, flexibility and useful domains (section 5.2)
19. Discussion placing CG unification into the context of other CG tools (section 5.3)
20. A formal definition of an alternative unification method (section 4.5)
21. A formal proposal for a more flexible definition of CG relations which would allow a greater latitude in representation power (section 6.3)

Declaration

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my thesis, when deposited in the University Library, being available for loan and photocopying.

Signed

Dated 7/2/01

Department of Computer Science
University of Adelaide
Adelaide, South Australia
July, 2000

Acknowledgements

First and foremost, I'd like to thank my supervisors, Dr. Rob Woodbury and Dr. Simon Ronald. Rob was willing to step into the supervision void when he was most needed, for which I'm very grateful. His knowledge and assistance in the design domain contributed significantly to my thinking in the area. Simon has been an endless source of knowledge, both in Computer Science and in the process of completing a Ph.D. He has also generally acted as a mentor and advocate throughout this entire process.

I would also like to express my sincere gratitude to Dr. John Maraist, recently of the University of South Australia, now of DePaul University, for some very useful discussions about the nature of completeness and soundness with regard to unification.

I am also indebted to colleagues and associates at the University of South Australia (in particular Prof. Martin Odersky) and the University of Adelaide (in particular Dr. Michael Oudshoorn). Many people at both of these institutions engaged in discussions on the work and made helpful comments.

I couldn't have finished the thesis without the assistance from my soul mate and lifelong companion, Wendy Mayer. Her careful suggestions and endless patience were invaluable.

I also received much help and encouragement from the Laggers Coffee Bunch: Terry, Felicity, Silke, Gail, Guy, Susan and Wendy. Thanks for all the help, wisdom and encouragement, and good luck to you all. Thanks in particular to Gail Higginbottom, for some insightful discussions on the nature and structure of a thesis.

I would like to dedicate this work to Wendy and all of the Laggers.

Chapter 1

Conceptual Structures

1.1 Overview of the Thesis

Chapter One introduces some of the background of this work, including the basics of Conceptual Graphs, and explores previous work in constraints and unification over CGs. All of the concepts that are relevant to subsequent chapters are presented here. The purpose of this chapter is to discuss past research that is relevant to the area of constraints and unification over Conceptual Graphs in order to put this work into context. The formal definitions of some of the terms used in CGs and in constraints are also set out here. The importance (and the significance to CGs) of having constraints implemented by unification in CGs is described in this chapter. Much of this chapter is based on previous observations and literature reviews from my earlier works [Corbett 2001; Corbett 2000; Corbett 1999; Corbett and Woodbury 1999; Corbett and Burrow 1996].

Chapter Two includes a discussion of the related areas of Constraint Satisfaction, Feature Structures and Unification, in order to set the work in this thesis in the context of these more general areas.

Chapter Three introduces and formally defines the methodology used to solve some of the problems of Conceptual Graph constraints and unification introduced in the first chapter. This chapter introduces some techniques and methods for carrying constraints in the concept of a Conceptual Graph, and describes a concept type for expressing constraints on the value of a concept. An efficient algorithm for the unification of this type of Conceptual Graph is introduced.

Chapter Four draws together all these ideas, and demonstrates this method of constraints and unification over Conceptual Graphs in several domains. A number of test sets are presented that are relevant to conceptual relation domains. These test sets are used to test the methodology described in Chapter Three, and the results and properties of this method are compared with the properties of other existing unification techniques for Conceptual Graphs.

Chapter Five places the unification model described in the previous chapters into the context of related work. Chapter Six summarizes the results and presents a discussion of the significance of this work and also presents some thoughts about future research directions. The discussion includes possible applications for CG unification.

1.2 Introduction

1.2.1 Graph-based Knowledge Representation Schemes

It has been demonstrated many times that graphs are a powerful and efficient knowledge representation technique. Mugnier and Chein [Mugnier and Chein 1996] illustrate quite effectively why labeled graphs are useful for knowledge representation in general. Among the main advantages that they list are a solid grounding when it comes to combinatorial algorithms, and that a graph (as a mathematical object) allows a natural representation (and therefore permits the construction of effective algorithms). Until recently, the major technique used in Computer Science for representing the semantic relationships between objects in a data structure was to use a graph technique known as Semantic Networks [Lehmann 1992]. Semantic nets have a long history of use in fields such as the semantics of medical diagnosis [Reggia et al. 1983], language understanding [Corbett 1991], and abductive reasoning [Dasigi 1991; Dasigi 1988].

There have been many attempts to formalize and standardize these graphical knowledge representation schemes, but probably none has been as extensive and comprehensive in recent times as Conceptual Graphs. The major use of Conceptual Graphs is in representing the relationships between objects in a system. Two of the drawbacks of Conceptual Graphs are that they do not easily represent constraints on real numbers, and that there are no efficient unification techniques, especially unification over constrained values.

1.2.2 Constraints on Values

The research discussed here concerns the development of the semantics of a constraint tool which operates over a subset of conceptual graph terms extended to express constraints on the referent values. The idea behind the matching mechanism is to combine order sorted unification with the action of a constraint solver. This combination leverages type representation technology from the Conceptual Graph community [Ellis 1995] and constraint technology [Mackworth 1992].

While the main focus of the research is the definition of the semantics of unification and constraints over conceptual structures, a major application of this work is to aid design projects in the elaboration and specialization of design space representations. So, the significance of this extended Conceptual Graph representation is two-fold. First, this effort will increase the power and utility of Conceptual Graphs by extending them beyond first-order equivalence into real constraints and unification. Second, this thesis demonstrates that the method defined here can be applied to real-world problems and domains.

The constraint techniques used in the Conceptual Graphs literature only cover the most basic kinds of constraints. They are based on using either a basic form of subsumption, or producing invalid graph segments [Mineau 1999; Mineau and Missaoui 1997; Müller 1997; Kocura 1996; Cogis and Guinaldo 1995; Willems 1995]. Real intervals are not represented at all, and there is still no method for validating a set of real constraints in a concept [Mineau and Missaoui 1997].

1.2.3 Unification of Graphs

In the Conceptual Graphs literature, the authors who have attempted to define unification of CGs have either restricted the allowable CGs in some way [Müller 1997], or have restricted the unification to a simpler operation [Kocura 1996]. A join on a Conceptual Graph is a simpler operation, and is usually used to merge two graphs.

Unfortunately, these attempts at merging graphs often lose some of the knowledge contained in the graphs being merged. In many domains, it is essential that the combining of knowledge be represented as a continual refining and specifying of the knowledge, so that none of the knowledge is lost in the process of combining with other knowledge [Corbett and Woodbury 1999; Woodbury et al. 1999; Chang and Woodbury 1996; Flemming and Woodbury 1995; Heisserman 1995; Carpenter 1992; Heisserman 1991; Colmerauer 1990; Jaffar and Lassez 1987].

Previously, when the knowledge of two graphs was combined there was no standard method for checking the validity of the values of the concepts, only for checking the structure or canonicity [Wermelinger 1997] of the graph. The further significance of this thesis is that a unification method is defined that leads to a useful and efficient implementation of constraints over CGs. The major significance of this work is that it improves on previous work in allowing constraints to be placed on real values in the concepts. The constraints are defined as a concept type, and therefore can be used as a type in the normal way with Conceptual Graphs. The constraints are enforced in the unification and join operations, as defined in this chapter. If a join operation violates the constraints on one of the concepts, the join fails. The unification algorithm used in this system is a standard algorithm, and is guaranteed to terminate since we restrict the CGs to a special subset, as defined in Chapter Three.

1.3 Conceptual Graphs: What They Are and How They Work

1.3.1 Overview of Conceptual Graphs

Conceptual Structures (or Conceptual Graphs, or "CGs") are a knowledge representation scheme, inspired by the existential graphs of Charles Sanders Peirce and further extended and defined by John Sowa [Sowa 1999; Sowa 1992; Sowa 1984]. Informally, CGs can be thought of as a formalization and extension of Semantic Networks, although the origins are different. They are labeled graphs with two types of nodes: concepts (which represent objects, entities or ideas) and relation nodes, which represent relations between the concepts. As an example, Figure 1 shows a Conceptual Graph which represents the knowledge that "The cat Felix is sitting on the mat which is known as mat 47."

Every concept or relation has an associated type. A concept may also have a specific referent or individual. A concept in a CG may represent a specific instance of that type (e.g., *Felix* is a specific instance, or individual, of type *cat*) or we may choose only to specify the type of the concept. That is to say that a concept may simply represent a generic concept for a type, such as *mammal* or *room*, or a concept may represent a specific object or idea, such as *my cat* or *the kitchen at the Smith's house*. In the former case, the concepts in Figure 1 would be shown as "cat: * " and "mat: * " indicating non-specified entities of types *cat* and *mat*. In the standard canonical formation rules for Conceptual Graphs, unbound concepts are existentially quantified.

A relation may have zero or one incoming arcs, and one or more outgoing

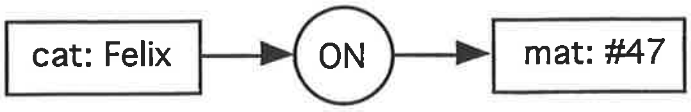


Figure 1. A Simple CG.

arcs. The type of the relation determines the number of arcs allowed on the relation. The arcs always connect a concept to a relation. Arcs cannot exist between concepts, or between relations.

A canon in the sense discussed here is the set of all CGs which are well-formed, and meaningful in their domain. Canonical formation rules specify how CGs can be legally built and guarantee that the resulting CGs satisfy "sensibility constraints." The sensibility constraints are rules in the domain which specify how a CG can be built, for example that the concept *eats* must have a theme which is *food*. Note that canonicity does not guarantee validity. A CG may be well-formed in the canononical formation rules for the domain, but still be false.

A type hierarchy is established for both the concepts and the relations within a canon. A type hierarchy is based on the intuition that some types subsume other types, for example, every instance of *cat* would also have all the properties of *mammal*. This hierarchy is expressed by a subsumption or generalization order on types. Many of these concepts are formalized later in this thesis.

Although the idea of a graphical representation of knowledge is quite old, Conceptual Graphs are still developing. Many tools and applications of CGs have only recently been described for natural language understanding [Chibout and Vilnat 1998; Moulin 1998; Nicolov et al. 1995], business processes [Gerbé et al. 1998; Wing et al. 1998; Gerbé 1997], ontology and epistemology [van Zyl and Corbett 2000a; Mann 1998; Tepfenhart 1998], the semantics of Conceptual Graphs [Wermelinger 1997; Wermelinger 1995], and knowledge engineering [Dibie et al. 1998; Ribière 1998; Corbett and Burrow 1996].

Conceptual Graphs are a useful and efficient knowledge representation tool. They can be used to represent the relations between (complex) objects in systems, and can represent multiple relations (unlike feature structures, as will be discussed in section 2.2).

1.3.2 Conceptual Graphs Formally

1.3.2.1 Fundamental Concepts

This section draws on previous work in defining Conceptual Graphs formally. Sowa discusses his original definitions in [Sowa 1984] but this thesis follows the further formalized and refined versions of Sowa's original ideas presented by Willems [Willems 1995] and by Chein and Mugnier [Mugnier and Chein 1996; Chein and Mugnier 1992]. The following two definitions are adapted from their work.

Definition 1. Canon. A canon is a tuple $(T, I, \leq, ::)$ where

T is the set of types. We will further assume that T contains two disjunctive subsets T_C and T_R containing types for concepts and relations.

I is the set of individuals.

$\leq \subseteq T \times T$ is the subtype relation. It is assumed to be a lattice (so there are types $\top$ and $\bot$ and operations $\vee$ and $\wedge$).

$:: \subset I \times T$ is the conformity relation. The conformity relation relates type labels to individual markers. This is essentially the relation which ensures that the typing of the concepts makes sense in the domain.

Some Conceptual Graph authors include B (also called σ) which associates each relation type with the concept types that may be used with that relation. This helps to guarantee well-formed graphs.

Definition 2. Conceptual Graph. A Conceptual Graph with respect to a canon is a tuple $G = (C, R, type, referent, arg_1, \ldots, arg_m)$ where

C is the set of concepts, $type : C \rightarrow T$ indicates the type of a concept, and $referent : C \rightarrow I$ indicates the referent marker of a concept.

R is the set of conceptual relations, $type : R \rightarrow T$ indicates the type of a relation, and each $arg_i : R \rightarrow C$ is a partial function where $arg_i(r)$ indicates the i-th argument of the relation r. The argument functions are partial as they are undefined for arguments higher than the relation's arity. In this thesis, we adopt the convention that arg_0 indicates the (at most) one incoming arc. If there is no incoming arc to the relation, then arg_0 is undefined. We also define the function *arity(r)* which returns an integer value representing the number of arguments that the relation r has.

It is common in the literature to write $c \in G$ instead of $c \in C$ when it is clear that c is a concept (similarly for relations $r \in G$).

1.3.2.2 Canonical Formation Rules

The following definitions are standard, classical definitions of CG formation, which date back to Sowa's original 1984 work on Conceptual Graphs [Sowa 1984], but which were formalized much more recently [Müller 1997; Wermelinger and Lopes 1994]. We present here rules based on the work of Müller [Müller 1997].

Definition 3. Canonical Graph. A canonical graph is a Conceptual Graph which is in the closure of the Conceptual Graphs in its canonical basis under the following operations, called the canonical formation rules.

1. External join. Given two CGs $G = (C, R, type, referent, arg_1, \ldots, arg_m)$ and $G' = (C', R', type', referent', arg'_1, \ldots, arg'_m)$. (without loss of generality we assume C and C' to be disjoint) $\forall c \in C$, and $\forall c' \in C'$ where $c = c'$ (that is, they have identical types and referents) the external join of C and C' is the CG $G'' = (C \cup (C' - \{c'\}), R \cup (R'_{c':=c}), type'', referent'', arg''_1, \ldots, arg''_m)$. The subscript $c':=c$ denotes the replacement of every occurance of c' by c. The functions *type* and *referent* are such that: $f'' \lfloor c \equiv f, f'' \lfloor c' \equiv f'$.

2. Internal join. Given a CG $G = (C, R, type, referent, arg_1, \ldots, arg_m)$ and two nodes $c, d \in C$ with identical types and referents the internal join is the CG $G' = (C - \{d\}), (R_{d:=c}), type\lfloor_{C - \{d\}}, referent\lfloor_{C - \{d\}})$. The subscript $d{:=}c$ denotes the replacement of every occurance of d by c.

3. Restrict type. Given a CG $G = (C, R, type, referent, arg_1, \ldots, arg_m)$ and a node $c \in C$ with type t which has a subtype $s \neq \perp$ the restrict type is the CG $G' = (C, R, type', referent, arg_1, \ldots, arg_m)$ such that $type'(c) = s$, $\forall d \neq c$: $type'(d) = type(d)$.

4. Restrict referent. Given a CG $G = (C, R, type, referent, arg_1, \ldots, arg_m)$ and a node $c \in C$ with $referent(c) = *$ and an individual marker $i \in I$ the restrict referent is the CG $G' = (C, R, type, referent', arg_1, \ldots, arg_m)$ with $referent'(c) = i$, $\forall d \neq c$: $referent'(d) = referent(d)$, and $type(c)::i$.

Sowa (and others) also define a *copy* rule, which allows a new graph G′ to be created as an exact duplicate of a graph G, and a *simplify* rule which allows the deletion of duplicate (and presumably redundant) relations. The *simplify* rule is just the equivalent of the *internal join* rule, but for relations.

1.3.2.3 Types and Inheritance

Types and inheritance for Feature Structures are discussed in section 2.2, and in more detail in [Carpenter 1992]. The discussion of type hierarchies presented here is adapted for Conceptual Graphs from Carpenter [Carpenter 1992]. The set of types discussed in Definition 1 is arranged into a type hierarchy, ordered according to the specificity of each type. A type t is said to be more specific than a type s if t inherits information from s. We write $s \geq t$, and say that s *subsumes* t or is more general than t (or inversely, that t is *subsumed* by s, or is more specifc than s). We may also call s a *supertype* of t, or t a *subtype* of s. Equivalently to the above, one can write $t \leq s$.

A standard restriction on inheritance hierarchy specifications is that they do not contain inheritance loops [Carpenter 1992]. It would simply be inconsistent (and even nonsensical) to be able to follow a chain of subtype links from a type back to itself.

In early pioneering work on the unification of first-order terms, Reynolds [Reynolds 1970] used the natural lattice structure of first-order terms, which was a partial ordering based on subsumption of terms [Davey and Priestley 1990]. Many terms (or types in our case) are not in any subsumption relation, for example *cat* and *dog*, or *wood* and *mammal*. Unification corresponds to finding the greatest lower bound of two terms in the lattice [Knight 1989]. The bottom of any lattice, which is represented with the symbol $\perp$, is the type to which all types can unify, and represents inconsistency. The top of the lattice, represented by $\top$, is the type to which all pairs of types can generalize, and is called the *universal* type. Every type is a subtype of $\top$. Inheritance hierarchies can be seen as lattices that admit unification and generalization [Knight 1989].

1.3.2.4 Specialization, Projection and Subsumption

The common specialization of two Conceptual Graphs, *s* and *t*, is known as a join, and is represented as $s \vee t$. The common generalization of the two graphs is known as a meet, and is represented as $s \wedge t$.

In this thesis, we will explain that the process of unifying Conceptual Graphs includes the process of finding the most general subtypes for pairs of types of concepts, which depends on the two types in question being consistent.

The definitions of unification, consistency and type subsumption in this thesis are based on formal concepts of projection and lower bounds. Carpenter [Carpenter 1992] defines each of these operators (for Feature Structures) as a morphism. This thesis follows Carpenter's definitions, modifying them to work with the properties of Conceptual Graphs. A morphism is then a mapping from the set of nodes of one Conceptual Graph to the set of nodes of another that preserves the order of relation arguments and the values of those arguments. In a morphism, all of the connections and arguments are preserved. The following definition of projection is the standard definition used in recent Conceptual Graph literature [Corbett 2001; Leclère 1997; Müller 1997; Mugnier and Chein 1996; Cogis and Guinaldo 1995; Willems 1995; Carpenter 1992].

Definition 4. Projection.

$G = (C, R, type, referent, arg_1, \ldots, arg_m)$ subsumes $G' = (C', R', type', referent', arg'_1, \ldots, arg'_m)$, $G \geq G'$, if and only if there is a pair of functions h_C: $C \rightarrow C'$ and h_R: $R \rightarrow R'$, called morphisms, such that:

$\forall c \in C$ and $\forall c' \in C'$, $h_C(c) = c'$ only if $type(c) \geq type'(c')$, and

$referent(c) = *$ or $referent(c) = referent(c')$

$\forall r \in R$ and $\forall r' \in R'$, $h_R(r) = r'$ only if $type(r) \geq type'(r')$

$\forall r \in R$, $arg'_i(h_R(r)) = h_C(arg_i(r))$,

Willems also includes the following non-emptiness condition in his definition of projection in [Willems 1995]:

$\forall c \in C$ there is a concept $c' \in C'$, such that $h_C(c) = c'$

This non-emptiness condition guarantees that all the concepts present in the more general graph are also present in the more specific graph, although they may be in a more specific state. Willems' definition allows for the more specific graph to have concepts of a more specific type, or for a generic referent to be replaced by a specific individual. The definition used in this thesis also admits the non-emptiness condition.

Leclère discusses the definition on not only type hierarchies, but also the functionality of types. Leclère also uses the projection operator to define the specialization relation [Leclère 1997; Leclère 1996]. He does this, however, by defining an atomic formula for Conceptual Graphs [Chein and Leclère 1994].

Essentially Leclère defines a transformation on Conceptual Graphs which computes projection. In Chapter Three of this thesis, we demonstrate an implementation of projection as a morphism, in a manner similar to Leclère and Chein.

Regarding the join and meet definitions, it is sometimes essential to obtain the most general common specialization for a given pair of Conceptual Graphs. In this case it is important to prove that not only is the graph obtained a consistent specialization of the two graphs being considered, but also that it is the unique graph which is the most general of all possible common specializations. Such a graph is known as the Greatest Lower Bound, as it represents the highest join which falls under the two graphs in the specialization hierarchy.

Definition 5. Greatest Lower Bound. The greatest lower bound (glb) of two CGs is the most general common specialization of the two Conceptual Graphs. Let G'' be a specialization of G and G'. G'' is the glb of G and G' if, for any Conceptual Graph U where $G \vee G' = U$, either $G'' \geq U$ or $G'' = U$.

The glb of two graphs s and t is written as $s \sqcap t$. Conversely, the most specific common generalization, known as the least upper bound (lub), of two graphs is written $s \sqcup t$.

Definition 6. Subsumption. We say that a Conceptual Graph G subsumes another Conceptual Graph G', or $G \geq_{CG} G'$, iff G' can be obtained by applying a finite number of canonical formation rules to G.

Note here that Definition 6 is actually redundant to these definitions, as subsumption is simply another form of projection. Since any application of the canonical formation rules to a graph s will always produce a graph t which is more specific than the original, s will necessarily have a projection into the new graph t. Chein and Mugnier formalize this idea, and demonstrate that $s \geq t$ iff there exists a projection from s to t [Chein and Mugnier 1992].

While Definition 6 is presented here for completeness, and to give formal substance to any discussion of subsumption, the rest of this thesis will concern itself strictly with the use of projection. Any mention of subsumption from this point can be construed as meaning projection.

To complete the formal discussion and definition of Conceptual Graphs, we present another result from Müller's work [Müller 1997].

Theorem 1. The subsumption order for Conceptual Graphs is a pre-order.

Proof. It is necessary only to show that this relation is both reflexive and transitive. First, any Conceptual Graph can be derived from itself by means of zero applications of canonical formation rules. Thus, CG-subsumption is reflexive. It is easily verified that the composition of two projections is also a projection. Therefore, CG-subsumption is also transitive. □

Although Sowa tries to show in [Sowa 1984] that subsumption is a partial order, this is in fact not the case, since the antisymmetry condition does not hold. Consider the example from Müller [Müller 1997] of the two CGs shown in Fig. 2. There are projections both ways even though the CGs obviously are not identical.

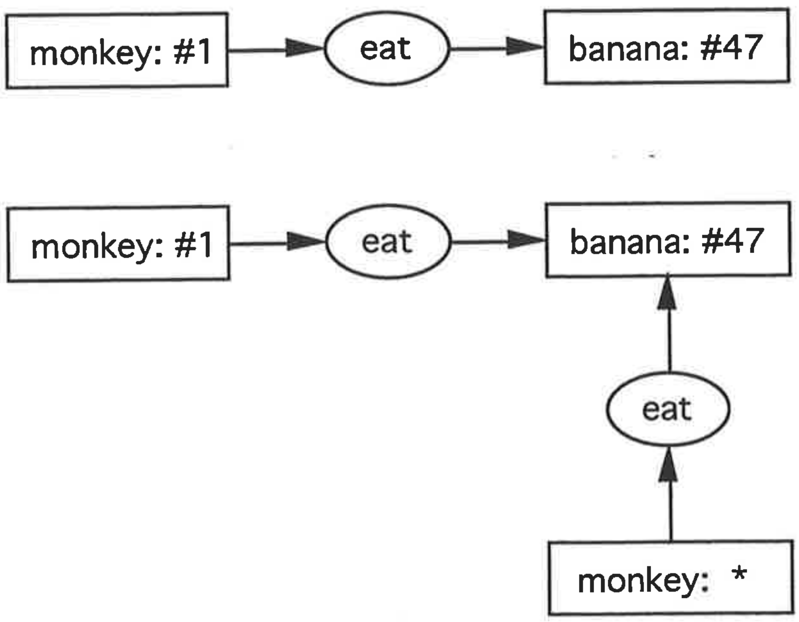


Figure 2. Müller's example of antisymmetry.

Also note that these examples are not isomorphic.

1.4 Previous Work in Constraints and Unification over Conceptual Graphs

1.4.1 Structural Constraints and Value Constraints

Until very recently, CGs have had no formalism for constraining real values in the referent of a concept. The standard method for representing and validating constraints has been to use type subsumption to specify which concept types (or subsumed subtypes) are valid in a system. One could constrain values in a knowledge representation system by forcing the concepts to conform to a specified type, or else to be subsumed by that type. A similar method applies to relations. To extend a previous example, the concept *eats* is specified to occur only between an agent which is an *animal* and a theme which is a *food*. Any individual used in the animal concept must conform to the *animal* type, which means that it must either be *animal*, or be subsumed by *animal*, such as *cat* or *reptile*.

However, there are some severe limitations on this type of constraint. First, reasoning with real numbers by subsumption is not only inefficient and difficult, but violates the basic definitions of Conceptual Graphs. Representing real constraints only with the standard definitions of Conceptual Graphs as presented in the previous section has led to various attempts to alter the structure of the graphs, or to implement procedures to validate the graph after making all the constraining calculations.

By defining smaller integers as "more general" than larger integers, one can define "greater than" and "less than" over the integers. While this method (though quite trivial) does informally define these functions for CGs, it is really more an artifact of the structure, rather than actually representing the relationship between numbers. For example, what intention or properties are inherited by the type *two* from the type *one*? How does it even make sense to discuss the number one as a type?

Furthermore, this technique fails in a more significant way when representing constraints on real numbers. A strict subsumption ordering cannot be defined when numbers can be sandwiched in between already defined numbers. That is to say, if properties are inherited by the type *two* from the type *one*, then what additional properties are inherited from *1.5* by *two*? What properties are inherited by *1.5* from the type *1.3*, or from *1.295* which are not part of the properties of the type *one*? How can it be said that *two* or *1.295* is more specific than *one*? The use of subsumption ordering on a lattice to produce constraints on real values is simply an artifact of the appearance of the lattice structure, and carries no formal, semantic meaning in Conceptual Graphs.

Further, by adhering to a strict subsumption order on other types of concepts, it becomes difficult to represent certain domain knowledge. In Chapter Four, examples of domains which rely on a continual construction and refining of the structures to find a structure which represents the solution to a problem in the domain are discussed. If a least upper bound type of unifier is used to unify structures in these domains, it is sometimes possible (depending on the unification algorithm used) to lose some of the knowledge gained in the refinement process. These example domains are presented in detail in Chapter Four.

There have been some interesting recent attempts to create a constraint system for Conceptual Graphs including the introduction of structural constraints on the graphs [Mineau and Missaoui 1997; Kocura 1996], the use of fuzzy logic in the definition of concepts [Cao et al. 1997; Wuwongse and Cao 1996], enforcing domain semantics on the relations [Dibie et al. 1998], and constraining processes which can change a Conceptual Graph [Mineau 1999; Mineau 1998; Mineau and Missaoui 1997].

Work on structural constraint systems has included Mineau's system of representing topological constraints and domain constraints [Mineau and Missaoui 1997]. Mineau's topological constraints are produced by specifying a section of the canon of graphs which are invalid in the specified domain, and then attempting to join them to graphs which are produced from the knowledge in the domain. A non-validity interval is a specification of two graphs *u* and *v*, such that $v \leq u$. Any graph *g* which falls into this interval is considered to be invalid. So *g* is invalid if $v \leq g \leq u$. The scheme relies on subsumption to find invalid graphs, but also relies on the development of efficient lattice operators [Champesme 1996; Ellis 1995; Godin et al. 1995; Wille 1992; Davey and Priestley 1990; Aït-Kaci et al. 1989].

Mineau specifies domain constraints by defining procedural attachments to graphs which are activated when their concepts are instantiated. These procedural attachments (or actors) should always have a projection into the graphs where they are used. If a projection does not exist, then the graph violates the constraints, and is invalid.

Figure 3, taken directly from [Mineau and Missaoui 1997] illustrates Mineau's concept of an implementation of an actor. The actors are used to help define a type, in this case a type called *employee-age*, which defines the legitimate age for a person to be hired into the profession under consideration in this domain. This type will be matched with the *age* concept in a graph which represents an employee (or potential employee), and the age of the employee will be unified with the generic referent labeled *x in the type definition. For the *employee-age* type to have a projection into the employee graph, the employee must be at least 18 years old, but no older than 70. The actors in this definition are

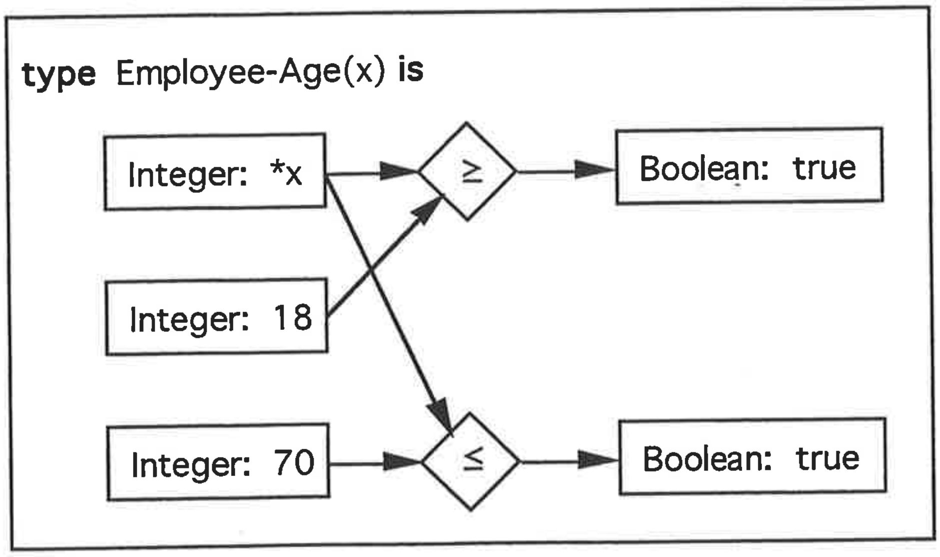


Figure 3. Mineau's actors.

shown as the ≥ and ≤ operators in diamond boxes. When the employee-age is unified with the generic referent, the actor ≥ verifies that the age is greater than 18, and sets the boolean concept appropriately. Similarly, the ≤ actor verifies that the employee age is less than 70, and sets its boolean concept. If both of the boolean-type concepts are true, then the employee is of an employable age.

Mineau's work really is more of an attempt to define valid structures in a canon, while the work presented in this thesis constrains the values in concepts. A major difference in these approaches is that the work presented in this thesis relies only on projection and join, while Mineau has implemented actors to check the graphs after an attempt at a join.

Kocura's approach [Kocura 1996] is another structural method, which essentially is to block certain parts of the graph away from the join; in order to disallow unification with certain parts of a graph. An illustrative example is one where he specifies that a man can marry a woman, but that a celibate man does not marry. He specifies which parts of a type hierarchy a graph can combine with, but excludes a certain concept, and then all of its specializations, including any join with other concepts or concept types. For example, *celibate man* could be joined with (and specialized by) *professional man*. It basically is the idea of not allowing a match too far down a type hierarchy.

This example is illustrated in Figure 4 which is taken directly from [Kocura 1996]. Kocura here demonstrates the concept of the negative canonical model. The relation *spouse* could be represented as the relation shown as R in the diagram. Two type hierarchies are shown as trees of subsumption, with the black nodes representing, for example, *mature-man* and *mature-woman*. The graph CM+ represents the positive canonical model. Any specialization of CM+ is a canonical graph, unless that graph is also a specialization of one of the negative canonical models, CM1- or CM2-. The gray nodes represent *male-member-of-celibate-religious-order*, and *female-member-of-celibate-religious-order*, respectively. The graph CCG is a canonical graph, since it is a specialization of CM+, but not a specialization of either CM1- or CM2-.

Many of the projects discussed in this section are attempts either to constrain the structure of the CGs, or to specify what types of concepts are valid. One notable exception is the work of Cao et al in the implementation of fuzzy values in the concepts [Cao et al. 1997] and in the definition of fuzzy concept types [Cao and Creasy 1998]. Cao's work is an implementation of fuzzy words, concepts and types, such as "fairly ripe" or "young." These implementations are useful for specifying fuzzy boundaries for the values in concepts, but Cao has not pursued constraint processing technology per se. Cao's type constraints are used to partially match concept types which are near to an expression, such as *not expensive* or *not low*. The purpose of these constraints is to allow types to subsume concepts which are near enough (in a fuzzy sense) to the type specified.

One of Cao's examples of a fuzzy program using Conceptual Graphs is

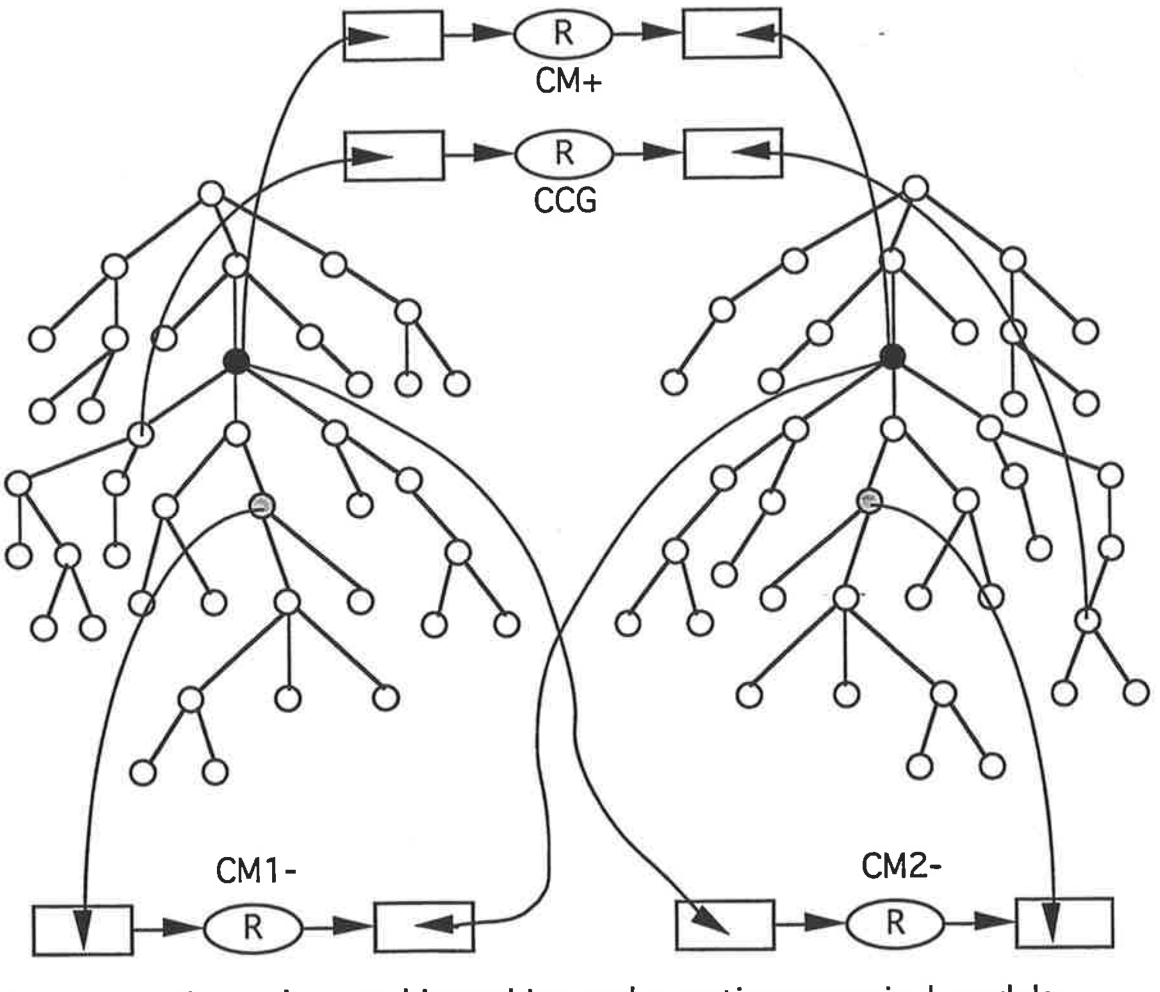


Figure 4. Kocura's type hierarchies and negative canonical models, from [Kocura 1996].

illustrated in Figure 5, which is taken directly from [Cao et al. 1997]. This figure shows a Fuzzy Conceptual Graph (FCG) program P which consists of an assertion that Apple #1 is fairly red, and a rule stating that an apple which is described as (being within a fuzzy tolerance of) red can be called ripe. There is also a query G, asking whether there is any fruit which is fairly ripe.

Cao defines a modus ponens, among other FCG operators and functions, which is used in this example to satisfy the query G. The fuzzy tolerance degree [Cao 1995] is used to match the program to the query. The response to query G is that Apple #1 is fairly ripe, which is a projection of G, and consequently implies G.

Based on the recent literature, the main techniques that exist for constraining the value of Conceptual Graphs mostly affect the structure of the graph. The constraint techniques used in the Conceptual Graphs literature only cover the most basic kinds of constraints. They are based on using either a basic form of subsumption, or producing invalid graph segments. Real intervals are not

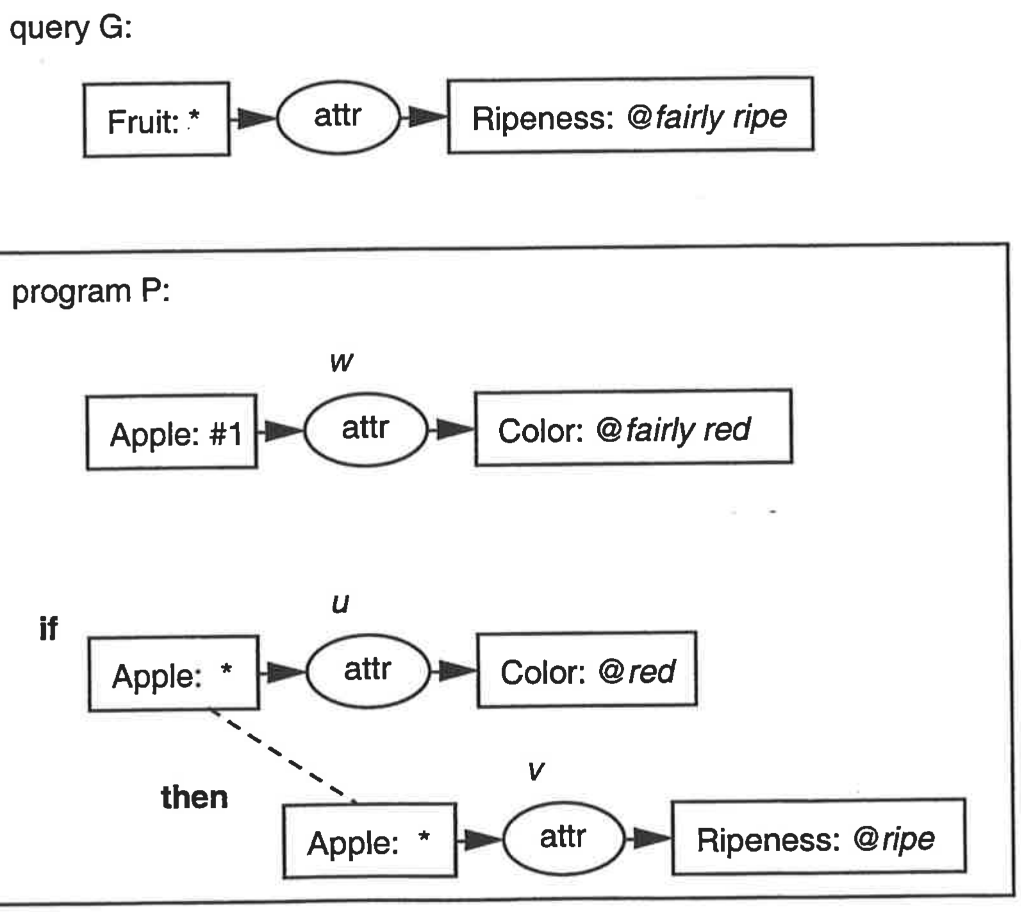


Figure 5. An example of Cao's Fuzzy CG programming.

represented at all, and there is still no method for validating a set of real constraints in a concept [Mineau and Missaoui 1997]. When the knowledge of two graphs is combined (unified, merged, or otherwise combined, as discussed in the next section) there is no standard method for checking the values, only for checking the structure of the graph.

1.4.2 Unification of Conceptual Graphs

Unification is related to constraint processing in that constraints are now used to play the role in theorem proving and logic programming that unification of terms once played in Constraint Satisfaction Problems [Baader and Siekmann 1994]. The unification-based approach computes projections and general unifiers and applies them to the terms under consideration. The constraint-based approach uses constraints to determine which instances are valid. Constraints can be seen as a filter that prohibits instantiations of the variables not satisfying this constraint [Baader and Siekmann 1994]. Further concepts in Constraint Satisfaction, Constraint Logic Programming and related areas are discussed in detail in Chapter Two.

This section discusses the current methods for unification and knowledge combination of Conceptual Graphs. In the methods described in the current CG literature, constraints are sometimes handled during the unification process, but again not as a standard CG technique. In order to make a CG programming language feasible and usable, it is essential for the user to be able to validate a set of constraints over a system. It is also essential to be able to combine knowledge in a way which is sensitive to the domain, and the knowledge being represented. Unification is the likely method for doing this.

The usual abstract definition given for the unification problem is to find an object z that fits both of the descriptions of two objects x and y [Knight 1989]. More specifically, the unification problem for logic can be described as, given two terms of logic built up from function symbols, variables, and constants, is there a substitution of terms for variables that will make the two terms identical [Knight 1989]?

We discuss unification of Conceptual Graphs in terms of combining the knowledge contained in two different graphs. While this may involve term substitution and constraint solving, this thesis is more concerned with knowledge combination as discussed in [Carpenter 1992]. Carpenter defines unification as a system in which two pieces of partial information can be combined into a single unified whole. In our case, these pieces of partial information are represented by Conceptual Graphs. Carpenter refers to this idea as information conjunction, but in this thesis, it is *knowledge conjunction* that is more important to us. Unification here is the combining of pieces of knowledge, represented as Conceptual Graphs, in a domain. Where information is simply a gathering and processing of data, knowledge is the intelligent application of information in a domain. Unification is then the combining of partial or incomplete knowledge into a single result.

Some researchers in the Conceptual Graphs community claim that CGs are equivalent to First Order Logic [Esch and Levinson 1995; Sowa 1992; Sowa 1984], and therefore that built-in unification already exists in the form of deduction and theorem resolution (or even Horn clause-equivalent terms). Sowa and Levinson's position is that Conceptual Graphs are equivalent to First Order Logic [Sowa 1992]. Their claim is that a system of CGs can easily be translated into First Order Logic expressions, have automated reasoning, deduction and theorem resolution applied, and then be translated back into the equivalent CG expression.

Esch and Levinson [Esch and Levinson 1995] discuss their concept of the phi operator, originally defined by Sowa [Sowa 1984]. The phi operator maps a CG into "an equivalent logic expression." Esch and Levinson claim that two CGs which are mapped to logically equivalent expressions by the phi operator have the same meaning. Unfortunately, neither a complete semantics nor an implementation of the phi operator has ever been described.

Wermelinger demonstrates that these assumptions of Esch and Levinson, and the original definitions of Sowa are flawed, by proving that Sowa's definition is incorrect and inconsistent [Wermelinger 1995]. He then constructs a new translation algorithm to solve the problem, stopping short of a complete translator which works in both directions. While he discusses the semantics of the phi operator for translating Conceptual Graphs into closed formulas of First Order Logic[1], he does not demonstrate an operator for translating FOL formulas back into CGs, failing to demonstrate equivalence between the two systems [Müller 1997; Wermelinger 1995]. Wermelinger asserts that it is a more important goal to have inference rules which operate directly on Conceptual Graphs [Wermelinger 1995; Wermelinger and Lopes 1994].

In the Conceptual Graphs literature, the authors who have attempted to define unification of CGs have either restricted the allowable CGs in some way, or have restricted the unification to a simpler operation. Fundamentally, the CG community has reconized that while First Order Logic subsumption is undecidable [Müller 1997], CG subsumption is at least well-defined, and well-behaved, since it involves at worst a finite number of mappings in a finite number of nodes [Corbett and Woodbury 1999; Müller 1997]. Unification work in Conceptual Graphs includes Willems' work on finding compatible nodes [Willems 1995], and Müller's restricted subset of CGs [Müller 1997].

Willems' approach to unification is to use type subsumption to compare compatible nodes [Willems 1995]. Two CGs unify in Willems' sense if they each have some common sub-graph, which can be joined under the usual type subsumption rules for CGs. The unified graph is then the joined graph, plus all the other relations and concepts in the two original graphs.

Willems [Willems 1995] discusses creating "unifiers" for Conceptual Structures which, in his definition, are closely related to the meet operation and the concept of finding a common generalization. His concept is to first find a "unifier," which is a segment which is similar in both graphs. The idea is to then match the two graphs together along the common segment by finding the join of

[1]In Wermelinger's definition of a closed formula, a formula with free variables can be regarded as equivalent to its universal closure.

all the nodes in the unifier, and then just attaching all other nodes (i.e. the nodes that are not in common) in the same relations that they were in before the unification. Willems refers to the process as "gluing together" the two graphs, but he does not present an algorithm for accomplishing this gluing process.

Willems' unification algorithm [Willems 1995] finds a segment of the CG which is present in both of the CGs being unified. This segment is a sub-graph of each graph, but Willems allows each individual concept c in the sub-graph to subsume or be subsumed by its counterpart in the other sub-graph, c'. Similarly for relations. In this approach, Willems then defines a graph which is at least as (or possibly more) general than both of the graphs being unified, and has a projection into both graphs, based on Sowa's π operator [Sowa 1984]. In Willems definition, the unifier is a graph in which every concept can subsume its corresponding concept in both graphs, and every relation can subsume its corresponding relation. This makes the unification algorithm efficient, as it takes advantage of the built-in CG attributes of subsumption and type hierarchies.

Willems' process of unification is illustrated in Figure 6, which is taken directly from [Willems 1995]. The two graphs G and G' are to be unified. The unifier U is found by taking the meet of G' and the segment of G which corresponds to "a man with a name, which is some word." The unifier is then "a person with a name, which is some word," shown as graph U in Fig. 6. Projections for $U \rightarrow G \rightarrow G''$ and $U \rightarrow G' \rightarrow G''$ are created to find the unification G''. The two major problems with this approach are that there is clearly more than one possible projection (why is 'Smith' the girl's name, and not the man's?) and that Willems does not actually describe how to find the projection into G''. He only describes an algorithm for finding the unifier U (essentially a graph isomorphism approach).

Willems' approach makes no attempt to implement a method for producing the unified graph, however. His effort is mainly to find an upper bound, which may be the least upper bound of the two graphs under consideration, which he terms the "most general unifier." It is unclear whether creating a generalization of

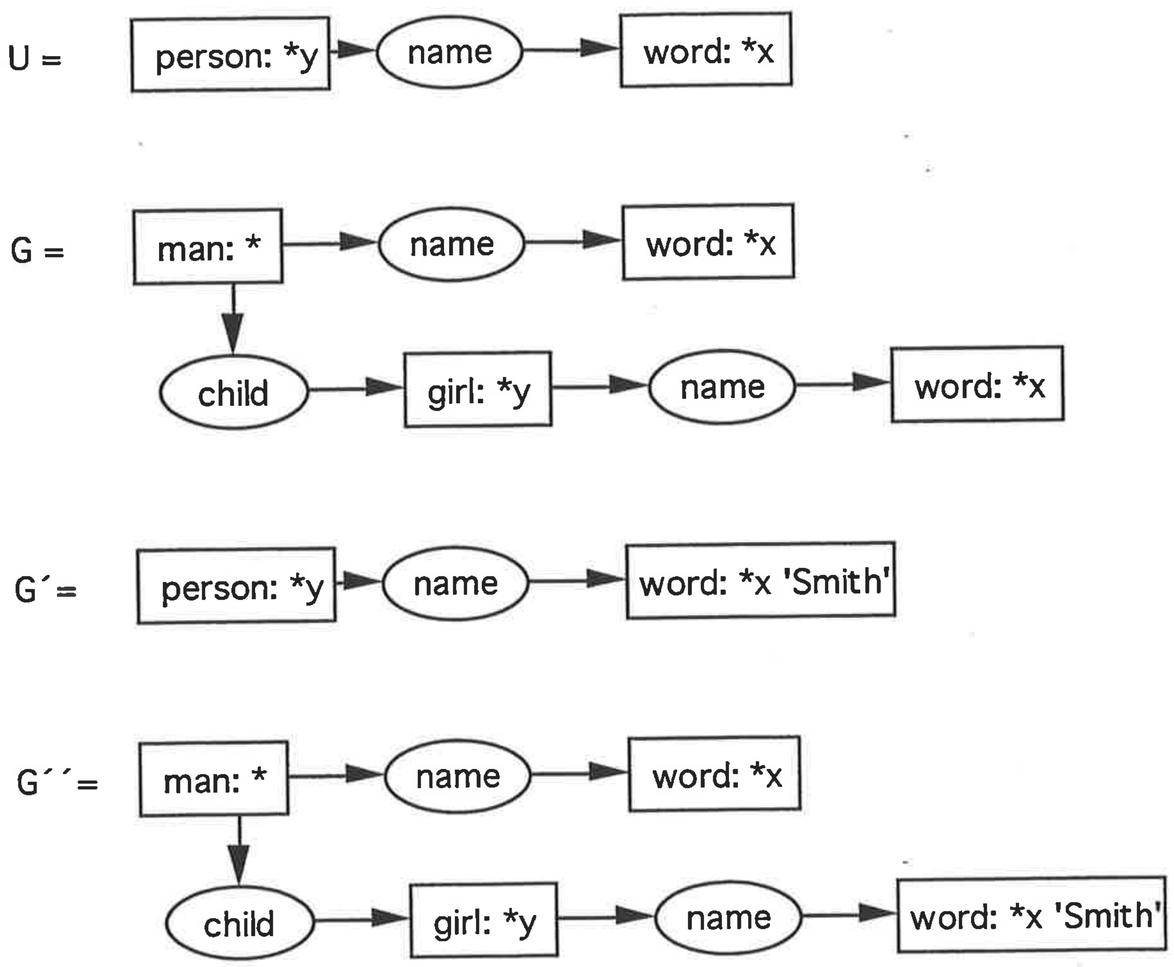


Figure 6. Unification in the style of Willems [Willems 1995].

a concept using his method loses the essential knowledge that the user wants to retain, through the loss of specifics in the generalization process. In the domain which was used for testing the algorithm presented in this thesis, the constructive nature of architectural design is a continual *refining* and *specifying* of the structure [Corbett and Woodbury 1999; Chang and Woodbury 1996].

Also, Willems' concept of a polyprojection allows for a certain amount of confusion in the implementation of joining concepts. A type of "crossover" of the relations attached to concepts could result from the generalization which is only useful in certain applications. While polyprojection does preserve the concept and relation pointers, the difference is that it is not clear which concept should be pointed to, if there is more than one possibility in the two graphs being unified. He compares this with the Prolog unification, where it is only important to find a

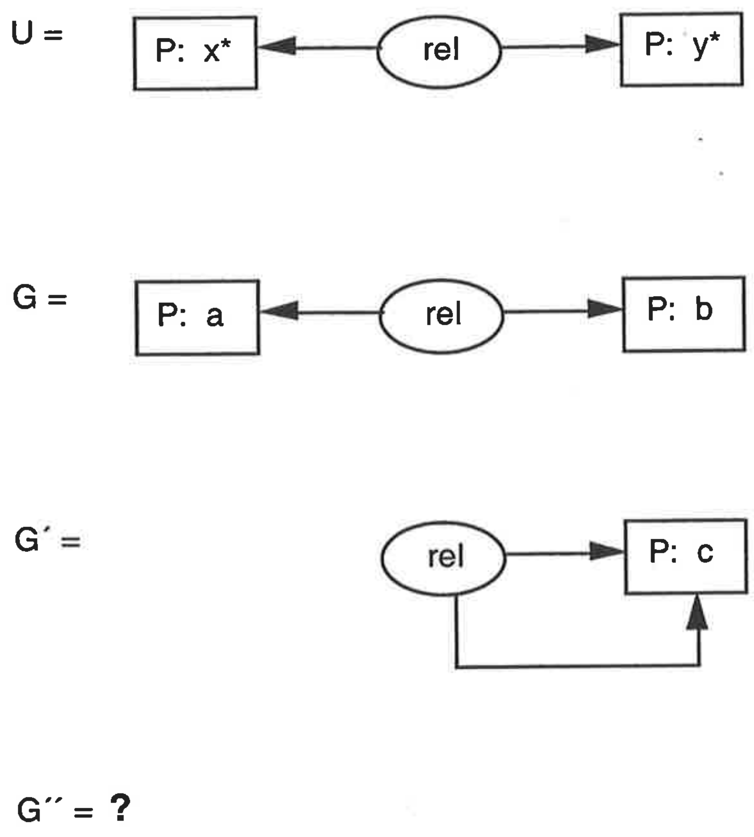
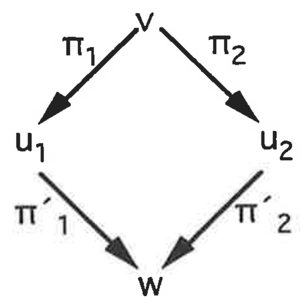


Figure 7. A counter example to Willems' unification algorithm.

concept that could potentially match a rule, given unbound variables. Essentially, Willems is defining a way to have no bindings on the variables, and still have a valid graph.

For example, in Figure 7, we wish to unify the two graphs G and G'. Under Willems' algorithm, it would be possible to derive graph U as the "most general unifier" in Willems' terms, but U is actually merely a common generalization of the two graphs, $G \wedge G'$. Willems only describes the method for finding the mgu, and does not describe how to derive the unified graph, G''. As the example shown in Figure 7 shows, there are cases where it is not possible to define an appropriate G'' from Willems' most general unifier[2]. Willems' unification algorithm, then, is *not* a unification of the information contained in the two original graphs. Rather, the resulting graph is something like a *merge* of the two graphs, which makes Willems' term, "gluing together," quite accurate.

[2] I am indebted to the anonymous reviewer from *Revue d'Intelligence Artificielle* for suggesting this counter example.



$$\begin{tikzcd} & v \arrow[dl, "\pi_1"'] \arrow[dr, "\pi_2"] & \\ u_1 \arrow[dr, "\pi'_1"'] & & u_2 \arrow[dl, "\pi'_2"] \\ & w & \end{tikzcd}$$

Figure 8. Compatible Projections.

Müller [Müller 1997] describes unification over a subset of CGs, and discusses the problems associated with attempting to find a general algorithm which will unify CGs. The problem that Müller is attempting to solve is finding a finite "unifying set" of common specializations which spans all possible common specializations of two CGs. In other words, if it becomes computationally difficult to specify the one Greatest Lower Bound of two CGs, would it be possible to specify a finite set of common specializations which together define all possible greatest lower bounds?

Müller borrows from the work of Sowa [Sowa 1984] and Willems [Willems 1995] in discussing a join on compatible projections. As shown in Figure 8, two projections π_1 from v to u_1, and π_2 from v to u_2 are defined to be compatible if for every concept of v, its image under π_1 and its image under π_2 have compatible types and referents. Müller then defines the maximum common specialization of two CGs, u_1 and u_2 with common generalization v, and compatible projections π_1 from v to u_1, and π_2 from v to u_2. The join on compatible projections is defined as the maximum element of all common specializations w, such that projections π'_1 from u_1 to w and π'_2 from u_2 to w exist. Müller shows that such a maximum element exists in general for any common generalization and two compatible projections.

The problem is to show whether this type of maximal join exists for graphs without the restrictions of showing a common generalization, or compatible projections. Müller proves that there is no such maximal element in these circumstances, and further that there is not even a finite unifying set for two such CGs. However, he is able to demonstrate that using headed CGs will always allow a Greatest Lower Bound to be found. Müller's conclusion is that CG unification is only guaranteed for graphs which have a labeled head node [Müller 1997]. Unifiers do not always exist for other CGs.

The recent studies of the unification of CGs have concentrated on small areas of applications. These techniques either simply merge the two graphs, in a straight-forward mechanical sense, or they restrict the structure of the graphs being unified. It seems as though the real intention of unification has been lost, however, in that the Conceptual Graph unification techniques found in the literature, while fixing small, domain-specific problems, do not perform a true knowledge combination of the graphs. While some restrictions, such as those proposed by Müller, can aid the efficiency of the unification algorithm, a true sense of knowledge combination is still not present.

The approach to unification of CG structures taken in this thesis is first to adopt the restrictions imposed by Müller, and deal only with finite, headed CGs, and then show that headed CGs with constraints can also be unified. In effect, the head concept node gives the algorithm a "starting point" for joining the two CGs. In the following chapters, the generality of the algorithm is demonstrated, before giving an example in the target domain.

1.5 Summary

This chapter has explored the fundamentals of Conceptual Graphs. Conceptual Graphs are a highly flexible and useful graph-based knowledge representation scheme. CGs can be used to represent the relations between (complex) objects in a system, and can represent multiple relations. CG concepts are designated with a type drawn from a lattice-structured type hierarchy. A canon can enforce "sensibility constraints" on the concepts and relations, and there is a very natural specialization order on CGs, induced by the canonical formation rules. The canonical formation rules correspond to "adding information."

The main techniques that exist for constraining values in a Conceptual Graph mostly affect the structure of the graph. The constraint techniques used in the Conceptual Graphs literature only cover the most basic kinds of constraints. They are based on using either a basic form of subsumption, or producing invalid graph segments. When the knowledge of two graphs is combined, there is no standard method for checking constraints on the values in the graphs, only for checking the structure of the graph.

Unification is defined here as the combining of pieces of knowledge, represented as Conceptual Graphs, in a given domain. This thesis defines unification as an operation that simultaneously determines the consistency of two pieces of partial or incomplete knowledge, and if they are consistent, combines them into a single result. The unification of two graphs contains neither more nor less information than the two graphs being unified. While some claim that CGs can be easily translated into other forms, such as First Order Logic, it is clear from recent studies that it is more important to have inference rules which operate directly on Conceptual Graphs.

While CGs are a highly flexible and efficient, they still lack a complete reasoning capability. Chapter One has demonstrated not only the potential utility and power of the unification of knowledge structures, but also that the unification techniques currently used for Conceptual Graphs are inadequate to fully capture this potential. These techniques do not combine knowledge in a way which maintains the integrity of the domain knowledge, and they do not handle constraints in the unification process.

Techniques for Unification, Constraint Processing, and knowledge structures are examined in Chapter Two. These techniques are found to be inadequate for use with CGs in their current state. After discussing some of the basic concepts and recent research in some areas of related work, the remainder of this thesis will then define a formal method for representing constraints on the value of a concept.

Then an efficient unification algorithm is formally defined which carries these constraints through the unification process, so that unification fails if a constraint is violated. These techniques are demonstrated in various domains, and with various types of knowledge. Finally, a discussion of their use and application is presented.

Chapter 2

Unification, Knowledge Structures and Constraints

2.1 Introduction

This chapter discusses the relationships between unification, knowledge structures, and Constraint Logic Programming. First, each of these fields is introduced, and the relevant literature reviewed. Then the recent research in each area is compared to each of the others. We find that there is a great deal of overlap in these fields of research, and that each field already contributes to the others. Finally, the interaction among these three fields is brought into the arena of Conceptual Graph research, and we examine how these fields contribute there.

2.2 Unification

2.2.1 Overview of Unification

The usual abstract definition given for the unification problem is to find an object z that fits both of the descriptions of two objects x and y [Knight 1989]. More specifically, the unification problem for logic can be described as, given two

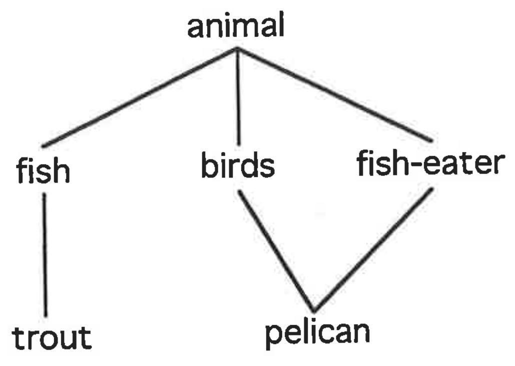


Figure 9. An example type hierarchy from Aït-Kaci's work [Aït-Kaci and Nasr 1986].

terms of logic built up from function symbols, variables, and constants, is there a substitution of terms for variables that will make the two terms identical [Knight 1989]? The modern definition of unification comes from Robinson [Robinson 1965], who introduced a method of theorem proving based on resolution.

The ψ-terms of Aït-Kaci [Aït-Kaci 1986; Aït-Kaci and Nasr 1986] are very similar to Feature Structures, in that subterms are labeled symbolically, rather than by argument position, and there is no fixed arity. The novel contribution of ψ-terms is in the use of type inheritance information. Aït-Kaci's view of unification was as a filter for matching partial structures, using functions and variables as the "filters." Aït-Kaci disagreed with the philosophy of Feature Structure unification, where two structures with different functors can never unify [Knight 1989; Aït-Kaci 1986]. Aït-Kaci relaxed this requirement for his ψ-terms by allowing type information to be attached to functions and variables. Then, his unification technique uses information from a taxonomic hierarchy to achieve a more gradual filtering.

An example from Knight [Knight 1989] illustrates this gradual filtering technique. Assume that we have the following inheritance information, as illustrated in Figure 9: Birds and fish are animals; a fish-eater is an animal; a trout is a fish; and a pelican is both a bird and a fish-eater. Then unifying the following ψ-terms:

fish-eater (*likes* $\rightarrow$ *trout*)

bird (*color* $\rightarrow$ *brown*; *likes* $\rightarrow$ *fish*)

will yield the new ψ-term:

pelican (*color* $\rightarrow$ *brown*; *likes* $\rightarrow$ *trout*)

Unification does not fail on comparing *fish-eater* to *bird*, or *trout* to *fish*. Instead, the conflict is resolved by finding the greatest lower bound on each of the two pairs of items in the taxonomic hierarchy, in this case *pelican* and *trout*, respectively. In this manner, Aït-Kaci's system naturally extends the information-merging (or *knowledge conjunction*) nature of unification.

2.2.2 Generalization and the Most General Unifier

Generalization is the dual function to unification, and can be used in many of the same domains as unification [Knight 1989]. Generalization is also known as anti-unification [Lassez et al. 1988]. The usual definition of generalization is, given two objects x and y, is there some third object z, of which both x and y are instances? In parallel to the greatest lower bound, the object which is the most specific of all common generalizations of two objects is known as the *least upper bound*. The most specific generalization of two terms retains information that is common to both terms, introducing new variables (essentially "unbinding" variables) when information conflicts [Knight 1989]. Knight gives the example that the most specific generalization of the two terms $f(a, g(b, c))$ and $f(b, g(x, c))$ is $f(z, g(x, c))$. Since the first argument to f can be a or b, generalization abstracts the terms to the variable z. Unification of these terms, however, would simply fail.

The various (and sometimes conflicting) definitions of unifier, most specific generalization, least upper bound, and most general unifier have led to some confusion over the intended application of a unifier. While we have set out our definitions of "most specific generalization" and "least upper bound" in the previous chapter, a discussion of "unifier" and "most general unifier" is still warranted, as there is still some confusion in the Conceptual Graphs community regarding these ideas. Some of this confusion may stem from the fact that many of the ideas used by the Conceptual Graphs community are borrowed from similar work in the Feature Structures community. In the standard definitions for Feature Structures however, a type hierarchy is represented as having its most general (or universal) type at the bottom of the lattice. Thus, a Least Upper Bound would represent a *specialization* of the types, while a lower bound would be a generalization [Carpenter 1992].

Lassez, Maher and Marriott [Lassez et al. 1988] cite many examples of confusion in the definition of Most General Unifier. They finally settle on a formal definition of "Most General Solution" to represent the idea of a Most General Unifier via a partial order on terms as the substitution that maps two terms to their most general common instance.

As mentioned in the previous chapter, Willems' approach makes no attempt to implement a method for producing the unified graph, but rather to find an upper (more general) bound [Willems 1995]. If this is the least upper bound of the two graphs under consideration, he terms it the "most general unifier". This phrasing is in conflict with other definitions of "most general unifier" [Knight 1989].

The definition of Most General Unifier offered by Lassez and by Knight is the definition most accepted in the unification literature [Knight 1989; Lassez et al. 1988]. We formally define "Most General Unifier" for Conceptual Graphs in the next chapter, and informally state here that our definition will be the unique most general concept which is more specific than the two graphs being unified, in accord with Lassez's and Knight's work.

2.2.3 Properties of Unification

Knight gives a list of several properties which are essential to unification, and which are useful in our study. Knight asserts that unification is monotonic, in that it adds information, but never subtracts [Knight 1989]. In our domain, we want to construct new knowledge structures from old, using unification defined over Conceptual Graphs. This continual refining and specifying process is essential in our domain. This is also in accord with Knight's second assertion of unification, as dealing with partially defined structures. As discussed later in this chapter, it is essential to be able to create partial descriptions of objects, and then specialize them later. The basic idea behind unification is to accept inputs that contain uninstantiated variables [Knight 1989]. The output may also contain uninstantiated variables.

Knight's next property of unification is very important in our domain. That is to view unification as a constraint-merging process. Structures which can not only represent information or knowledge, but can also encode constraints can use unification to merge the constraints. Unification can then be used to detect when combinations of certain constraint sets are inconsistent.

2.2.4 Tree Isomorphism as Unification

The standard graph technique of tree isomorphism is normally used to check whether two trees have the same structure under a mapping algorithm [Aho et al. 1974], but this technique can also be used as an efficient implementation of the unification of graphs, if the further restriction of only allowing structures which are finite trees is allowed. The definition of tree here is quite strict: The graph must have a root (or head) node, and be finite, directed, and acyclic with no node having multiple parents. Under the formal definition of Tree Conceptual Graph, all relations must be binary.

The formal definition of a Tree Conceptual Graph will be discussed in Chapter Four along with an algorithm for unification of Tree Conceptual Graphs. The essential point here is that graph merging techniques do exist which are very efficient, but they can place restrictions on the type, structure and flexibility of the graphs for which they will work.

2.3 Feature Structures

2.3.1 Overview of Feature Structures

Feature Structures are another graph-based knowledge representation scheme which also provides a formalism for representing partial information about the objects in the domain under consideration. Feature structures are therefore very similar to Conceptual Graphs, except that Feature Structures are basically simpler structures, and therefore generally more efficient in terms of computation and complexity. They also have a well-defined and very efficient unification algorithm.

One of the advantages of Feature Structures is that the graphs are specified functionally, so that unlike First Order Terms, the position of the argument is not important. Instead, Feature Structures use functors, called "features," which have explicit names. These functors can be arranged into a type hierarchy, so that if we know that functor *f* subsumes functor *g*, then *f*(a) subsumes *g*(a), for any object a in the domain.

The usual way to conceptualize a Feature Structure is as a labeled, rooted, directed graph [Carpenter 1992]. A root (or "head") node is specified for each Feature Structure, and all other features follow on from the root. Since the order or position of the arguments is not relevant to the definition of the knowledge structure, the structure essentially has an extensible arity, in that another functor can always be added to the structure to further specify the structure.

Feature structures are depicted graphically in several different forms. Two of these forms are shown here in Figure 10 and Figure 11, which are taken directly from Carpenter [Carpenter 1992]. Figure 10 illustrates the standard graphical notation for feature structures, where its automata-like and graph-like character are most apparent. The nodes are enclosed in small, numbered circles, and a small arrow indicates the root node. The type of each node appears in bold face next to their nodes, and the features label the arcs. This notation is generally considered by the Feature Structures community to be highly illustrative of the knowledge, but is unfortunately complicated and difficult to understand when the structure is complex [Carpenter 1992].

Figure 10 illustrates the knowledge of a grammar. The knowledge depicted can be understood as "a sentence requires both a **subject**, which must be of type **noun**, and a **predicate**, which must be of type **verb**. The **noun** and the **verb** must be in **agreement**, with the **syntax** of the sentence. In this case, the **person** must be **third**, and the **number** must be **singular**.

Figure 11 illustrates the more standard method, which is the frame-like attribute-value matrix notation. In this notation, each bracketed entry represents a node, and the type of the node is written at the top left of the frame. The tags, such as the boxed 4 in the diagram, indicate recursive (or just structure sharing) re-entrancy. The slots are the features, and their values are written next to them.

2.3.2 Feature Structures Formally

This formal definition of Feature Structures is due to Carpenter [Carpenter 1992] and follows the standard definitions in the Feature Structure literature. In this definition, we assume that a finite set of features, **Feat**, and an inheritance hierarchy, **Type** (and a subsumption function over the hierarchy), have already been defined.

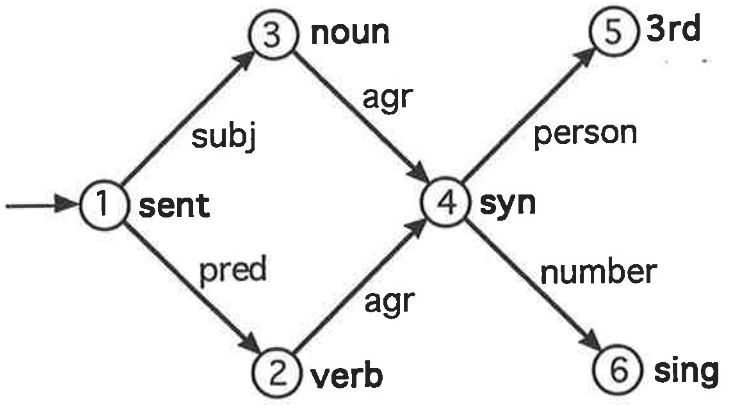


Figure 10. A Feature Structure in Graph Notation.

Definition 7. Feature Structure. A feature structure over Type and Feat is a tuple $F = (Q, q, \theta, \delta)$ where:

Q is a finite set of nodes, rooted at q,

$q \in Q$ is the root node,

$\theta : Q \rightarrow \text{Type}$ is a total node typing function, and

$\delta : \text{Feat} \times Q \rightarrow Q$ is a partial feature value function.

The usual way to visualize these definitions is that Q is the set of nodes, θ determines the labels on the nodes and, where there is an arc from q to some q' labeled by f, then $\delta(f, q) = q'$.

2.3.3 Strengths and Limitations of Feature Structures

In Feature Structures, the substructures are labeled symbolically, rather than inferred by argument position. This means that there is no need for a fixed arity of arguments in a structure, since all arguments can be referred to functionally [Carpenter 1992; Knight 1989]. This is a major difference with Conceptual Graphs, where argument position is significant, although symbolic names can still be used

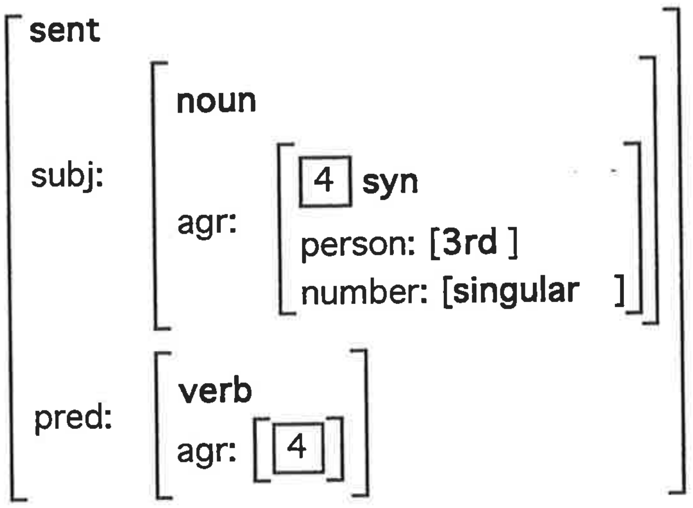


Figure 11. The Feature Structure of Fig. 10 in matrix notation.

to label substructures. Note, however, that Chapter Six of this thesis will propose a method of variable-arity relations, which will use the symbolic naming convention, and not rely on argument position.

Feature Structures also remove the distinction between function and argument, while Conceptual Graphs have some difficulty with functions (although note some recent work in bringing agents and processes into the formalism of Conceptual Graphs in order to express functional constraints [Mineau 1999; Mineau 1998; Mineau and Missaoui 1997]).

When dealing with Feature Structures, however, the distinction between the intension and the extension of the structure must be adhered to. In Conceptual Graphs, concepts which use the generic marker are existentially quantified, but there is no real distinction made between a Conceptual Graph as a framework of the knowledge in a domain and the knowledge contained in that framework. A graph made up only of generic markers can be easily unified with other graphs, and in fact that is the standard method for gathering, specifying, and validating the knowledge of a system. Both systems have a method for expressing partialness of the knowledge.

2.4 Logic Programming, Constraint Logic Programming, and Constraint Satisfaction Problems

2.4.1 Logic Programming

Logic programming is a language paradigm based on logic, and specifically on resolution theorem proving as proposed by Robinson in [Robinson 1965]. Robinson distinguished between the two components in automatic theorem proving: the inference rule, called resolution, and the testing for the equality of trees, which he called unification [Cohen 1996; Robinson 1965]. In Robinson's view of resolution theorem proving, resolution is an inference step used to prove the validity of predicate calculus formulas expressed as clauses, and unification is the matching of terms used in a resolution step.

Prolog, the first logic programming language, was at first a tightly-constrained resolution theorem prover, which was later turned into a useful programming language [Colmerauer 1990; Knight 1989]. It essentially inherited unification as a central operation through its use of resolution. Prolog consists of a sequence of Horn clauses, which is a clause containing at most one positive term. Prolog programs can be viewed as a set of definite clauses in which the positive literal is the head of the rule and the negative literals constitute the body or tail of the rule [Cohen 1996]. Functionally, the head corresponds to the definition of a Boolean function, whose body consists of a conjunction of calls to the Boolean functions representing the tail [Kowalski 1979].

2.4.2 Constraint Logic Programming

Constraint Logic Programming (CLP) began as a natural merger of two declarative paradigms of constraint solving and logic programming [Jaffar and Maher 1994]. Jaffar claims that it is this combination of schemes that helps to make CLP programs both expressive and flexible. A large amount of work in constraint programming languages actually preceded logic programming and CLP [Jaffar and Maher 1994]. Taken broadly, CLP can be said to incorporate constraints and constraint solving methods in a logic-based language [Jaffar and Maher 1994; Jaffar and Lassez 1987].

Jaffar makes the point that despite this characterization of CLP as suggesting the possibility of many languages based on different logics and constraints, the fact is that CLP has almost exclusively been devoted to languages based on Horn clauses (but [Bürckert 1991], discussed later, presents an exception). Many languages based on definite clauses have very similar semantics. In fact, the semantics of a logic-based programming language could be parameterized by a choice of the domain of computation and constraints. The resulting scheme defines the class of languages *CLP(X)*, which is obtained by instantiating the parameter X [Jaffar and Maher 1994].

Constraint Logic Programming languages are Logic Programming languages in which unification is replaced by constraint solving in various domains. Constraints are expressed as special predicates whose satisfiability can be established using efficient algorithms [Cohen 1996]. Unification can then be viewed as a particular type of constraint that tests equality in the domain of graphs. There exist some CLP languages, such as Prolog IV™ [PrologIA 1997] and Numerica [Van Hentenryck et al. 1997], which use intervals to bound their constraints, similar to our solution for Conceptual Graphs (see Chapter Three).

Aït-Kaci [Aït-Kaci et al. 1992] describes a constraint system over Feature Structures. The system (called FT) depends on having finitely many possibilities (or possible sub-trees, or branching) in the feature trees. This corresponds to having finitely many features in a Feature Structure. Aït-Kaci defines a model for feature trees, and claims that the model defines the semantics of constraint processing on Feature Structures.

Constraints in FT are discrete and finite. The entire domain is specified with the structure (ie as part of the tree), so that there is no distinction between the framework of the knowledge of domain, and the knowledge contained in that framework (just as we define later for CGs). There's no room for real-value constraints in FT but Aït-Kaci discusses the possibility of the infiniteness of his structures, implying that a continuous, real-valued domain can be represented by continually inserting new features between existing features [Aït-Kaci et al. 1992].

2.4.3 Constraint Satisfaction

Constraint satisfaction, sometimes known as constrained deduction, or the Constraint Satisfaction Problem (CSP), has been formalized in many ways [Van Hentenryck et al. 1997; Baader and Siekmann 1994; Gini and Rogialli 1994; Mackworth 1992; Bürckert 1991; Kirchner et al. 1990; Van Hentenryck 1989]. The underlying concept in all of these formalizations is that it is not necessary to compute a complete set of solutions to the constraints, and that merely deciding satisfiability of the constraints is usually sufficient.

$$\begin{tikzcd}
\text{CLP} \arrow[d, dash] & \text{(general)} \\
\text{CSP} \arrow[d, dash] & \\
\text{FCSP} & \text{(specific)}
\end{tikzcd}$$

Figure 12. Mackworth's hierarchy of constraint satisfaction techniques.

A formalization of CSP usually takes this form. Given a set of variables $V = \{v_1, v_2, \dots v_n\}$ and a set of constraints on those variables $C = \{c_1, c_2, \dots c_k\}$, a solution for the problem is an ordered assignment of a set of values $S = \{s_1, s_2, \dots s_n\}$ to the variables such that all the constraints are satisfied. Some of the work in CSP relies on efficient backtracking methods [Nadel 1990], while others are concerned with representing CSP as a highly restricted logical calculus with associated properties and algorithms [Mackworth 1992].

Mackworth's approach to constraint satisfaction is to exploit the logical relationship between Constraint Logic Programming and constraint satisfaction. He first defines Finite Constraint Satisfaction Problems (FCSP), which are basically CSPs with finite domains, and then describes a hierarchy of CLP, CSP, and FCSP (see Figure 12).

Mackworth's idea is that by describing logical methods of approaching CSP, he attempts to uncover the semantics of common approaches which are more easily solvable in CLP. Mackworth's main hypothesis is that all CSP must have an underlying logical calculus, which can then be used to lift the CSP problem to a solvable CLP problem.

Finally, Mugnier and Chein convincingly demonstrate that any CSP problem can be represented as a mathematical morphism [Mugnier and Chein 1996]. Their proof of the strong correspondence between CSP and the general problem of morphism (or projection) also demonstrates that a type hierarchy, such as used in Conceptual Graphs, can be effective in representing and solving a CSP problem. They develop, and prove the soundness of, algorithms for transferring CSP problems to a projection problem, and for transferring projections back to a CSP representation.

Mugnier and Chein demonstrate that the algorithmic techniques that they develop for resolving the problem of the existence of a solution to a Constraint Satisfaction Problem also can enumerate the solutions. Further, these are transferable from one domain to another [Mugnier and Chein 1996].

2.5 Unification and Constraints

2.5.1 The Relationship Between Unification and Constraints

Unification is related to constraint processing in that constraints are now used to play the role in theorem proving and logic programming that unification of terms once played in Constraint Satisfaction Problems [Baader and Siekmann 1994; Bürckert 1991; Kirchner et al. 1990; Jaffar and Lassez 1987]. The unification-based approach computes projections and general unifiers and applies them to the terms under consideration. The constraint-based approach uses constraints to determine which instances are valid. Constraints can be seen as a filter that prohibits instantiations of the variables not satisfying the constraint [Baader and Siekmann 1994].

Unification (by projection) is the mechanism used in this thesis to find the solution of the constraints. The difference with this thesis is that here, the structures are also carrying a complex, powerful knowledge representation scheme along with the constraints. As discussed previously, unification starts off as the identifying of two logical formulae by variable substitution. In this thesis, unification is a tool which performs the work of identifying two structures using subsumption, where the elements of the structure can be constrained.

2.5.2 Unification-based Reasoning and Constraint-based Reasoning

Automated deduction approaches differ in that a unification-based approach computes a complete set of unifiers (or all terms which have projections [Willems 1995]) which are more general than the terms under consideration, and immediately applies these unifiers to the terms [Baader and Siekmann 1994]. Simply finding the least common generalization of two terms is sometimes called anti-unification [Lassez et al. 1988], or generalization [Knight 1989].

In a constraint-based approach, one does not actually instantiate the terms by solutions of the constraints, but the constraints determine which instances of the terms under consideration are admissible. In order to know whether there are still admissible instances one must determine solvability of the constraints. The main advantage of this approach is that it is not necessary to compute a complete set of solutions (e.g., [Müller 1997; Baader and Siekmann 1994]). Merely deciding whether a solution exists while satisfying all of the constraints is sufficient. In this case, it is not necessary to compute a general unifier or projection of the terms under consideration, but only to test whether the unification problem is solvable. In the case that the unification is solvable, one only need generate a new constrained expression of the problem. The constraint can then be seen as a filter that prohibits instantiations of the variables not satisfying this constraint [Baader and Siekmann 1994].

In principle, it is enough to decide solvability of the constraints, however in a realized system, it is necessary to define an algorithm which is also incremental. That is, the algorithm should not simply state whether the constraints are unifiable, but then should also transform the constraints into a simpler, "solved" form [Baader and Siekmann 1994]. In this form of constraint solving, "solved" can mean either finding the specific instance of a concept which simultaneously matches all constraints, or it can mean narrowing the constraints into a simpler form, which still matches all constraints, but represents them in a simpler, easier to manage form. This simpler form can still be converted into an instance later. Then, when combining constraints, it is not necessary to re-do all the work from previous constraints, but only to combine the new constraint with the solved form of the previous terms, which corresponds to their most general unifier (or greatest lower bound, in CG terms).

2.5.3 Unification as Reasoning

Another motivation for this approach to constraint solving is the ability to enhance CSP by considering more general constraints than the usual equality constraints of normal unification [Baader and Siekmann 1994]. One generalization of unification constraints is the use of ordering constraints, i.e., constraints of the form $s \leq t$ where s and t are terms. Depending on the application, the ordering $\leq$ may have different interpretations. Recall from Chapter One that in conceptual graphs, $\leq$ has the meaning of subsumption on a partially-ordered lattice of concepts (in the case of the present discussion, this means projection of one Conceptual Graph onto another). A concept may unify by subsumption with another concept if one of the concepts is a more general expression of the other, as defined in the partial order.

There are also constraint approaches in logic programming where constraints are not interpreted over a single structure. An example for such an approach is H. Aït-Kaci's Login [Aït-Kaci et al. 1992], where first-order terms are replaced by feature terms, as discussed earlier in this chapter.

H.-J. Bürckert describes a framework for general constraint resolution theorem proving [Bürckert 1991]. Bürckert's work followed the work in automated deduction, where the goal was to integrate semantic information about the problem domain, in order to intelligently explore the search space (by intelligent pruning, directed search, etc.). His work with constraints proposed a method to handle clauses whose variables are bound by restricted quantifiers.

In Eisinger and Ohlbach's discussion of intelligent behavior in deduction systems based on resolution [Eisinger and Ohlbach 1993], the importance of defining a resolution technique is made clear. They describe the *logic* of a system as the syntax and semantics of a deduction system, which includes the ideas of entailment and the formalization of the intuitive relationship between statements. The logic is a precise representation of the system, which defines the permissible structure and meaning of statements. Most deduction systems define a notion of semantic entailment which does not provide any means to determine algorithmically whether a given statement entails another [Eisinger and Ohlbach 1993]. However, Eisinger and Ohlbach define a *calculus* of the system, which extends the syntax by syntactic rules of inference, and is treatable in an algorithmic way.

For example, Conceptual Graphs as they stand have a logic in this sense, in that they can represent knowledge formally and use subsumption, join, etc. but, in the terms of Eisinger and Ohlbach, they do not have a calculus yet. A calculus would allow Conceptual Graphs to have reasoning in the system. Unification (and thereby a system of inference and resolution) is a good first step toward giving Conceptual Graphs that reasoning ability. This thesis will give Conceptual Graphs their first steps toward being a usable reasoning system, and thereby a usable programming language.

Essentially, while Conceptual Structures do not contain efficient constraint satisfaction and reasoning methods, the standard Constraint Satisfaction techniques do not contain methods for efficient storage and manipulation of domain knowledge. This thesis will bring constraint processing into the Conceptual Graphs formalism. Chapter Six contains some thoughts about bringing together the fields of CSP and CGs more closely.

2.6 Knowledge Structures, Partialness and Unification

2.6.1 The Relationship Between Unification and Knowledge Structures

Unification is related to Knowledge Structures through the concept of partialness. Structures which are not completely specified can be merged together through unification. Feature Structures are limited in this sense, because pure features or properties cannot be unified under the Feature Structures formalism.

In this section, we discuss the idea of knowledge conjunction in terms of partially-specified structures being unified. The purpose, need and intent of unification of Conceptual Graphs is clarified.

2.6.2 Partialness

In this thesis, *partialness* means that a structure need not contain all information that is implied about it by its structure and types. A partial representation is used here as a generalized, or higher-level description of an object in the domain. Whether a structure is partial or not depends on the context of the knowledge, and the domain. In domain terms, a model might be partial against one set of knowledge but complete with respect to a subset of the knowledge. For example, if our current domain knowledge of a building is limited to its spatial organization, a complete model of it would assign functions to physical spaces. Such a model would be partial with respect to a larger set of knowledge, containing for example, knowledge of how to construct the building.

The main thrust of the research described in this thesis is the unification of Conceptual Graphs in terms of conjoining the knowledge contained in two different graphs. While this may involve term substitution (or the Conceptual Graphs equivalent - instantiation, subsumption, variable binding, etc.) and constraint solving, this thesis is more concerned with knowledge conjunction as discussed in [Carpenter 1992]. Carpenter defines unification as a system in which two pieces of partial information can be combined into a single unified whole. In our case, these pieces of partial information are represented by Conceptual Graphs. Carpenter refers to this idea as information conjunction, but in this thesis, it is *knowledge conjunction* that is more important to us. We want to be able to combine the expert knowledge of a system, or even combine knowledge from different sources, not merely gather additional information. Unification here is the combining of pieces of knowledge in a domain, represented as Conceptual Graphs. In this thesis, unification will be defined as an operation that simultaneously determines the consistency of two pieces of partial or incomplete knowledge, and if they are consistent, combines them into a single result.

In Feature Structures theory, it is important to know whether one is attempting to unify the *intensions* or the *extensions* of two Feature Structures (FS). Essentially, the intension of a Feature Structure is all of the attributes (or properties, or *features*) of a construct. The extension of a Feature Structure is the actual object being represented, with the attributes specified, even if only partially [Wille 1996a; Wille 1996b]. In Feature Structures theory, one must decide whether the Feature Structures being unified are of the same *intensional* type, or the same *extensional* type, and then seek to identify the two FSs under that type. The unification of two FSs under their extensional type is simply the identification of all their values for their features (similar to type labels and individual markers for the concepts in CGs). There is no way to derive identities of intensional types of two Feature Structures, as there are no values to be compared.

The significance of intensionality in a representation scheme is the simple fact that two structures can be identical in all aspects yet remain distinct objects. In an intensional representation scheme, such as Feature Structures, two structures which represent the same structure must be explicitly identified as being the same.

It is this property of intensionality which does not allow Feature Structure unification over existentially quantified features. One major advantage that Conceptual Graphs have over Feature Structures is that Conceptual Graphs which contain existentially quantified concepts can still be unified.

2.6.2 Intensionality, Join and Unify

The formal definition of unification for Conceptual Graphs is set out in Chapter Three of this thesis, however, it is essential to clarify the difference between the "join" operator, introduced in the previous chapter, and the general concept of unification. The difference between these two operators can be illustrated in the following way. In the standard canonical formation rules for Conceptual Graphs, unbound concepts are existentially quantified. We take for

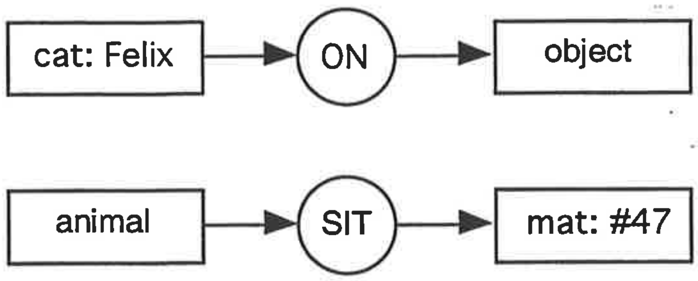


Figure 13. Is Felix on the mat?

our example the two graphs in Figure 13, which can be interpreted as "Felix is on some object," and "There is some animal sitting on that particular mat." Joining these two graphs is not possible under the standard canonical formation rule for external join because there's no projection from one graph to the other. However, there are individual concepts which can be joined, such as the concept that "Felix is a cat" and "animal." However, as discussed in previous sections of this chapter, true unification is the knowledge conjunction of the two graphs. The unification of these two Conceptual Graphs would be similar to the unification of ψ-terms presented by Aït-Kaci. The unification is therefore "Felix sat on mat number 47," as shown in Figure 14. Here, the more general concepts of "animal," "on," and "object" have been replaced by their more specific instances. This illustrates that unification is more than an external join, and is composed of several operations, including join.

The external join rule can be used to "glue together" two graphs in Willems' sense, in that a few compatible concepts and relations can be joined together from two graphs to make a larger, joined graph. Willems then attempts to create a truly unified graph by finding the least upper bound of the two graphs that will validate this newly joined graph [Willems 1995]. As discussed in Chapter One, in

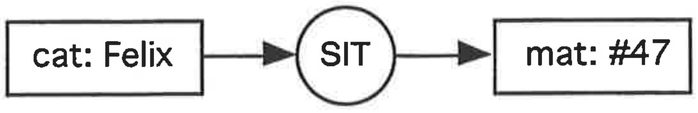


Figure 14. Felix is on the mat.

the true sense of unification simply joining a few concepts and relations does not guarantee the conjunction of the knowledge contained in the graphs.

Unification, however, is somewhat more complicated, and also more interesting and useful. The unification of two graphs contains neither more nor less information than the two graphs being unified. Figure 14 shows that the unification of the two graphs in Figure 13 still retains all the information of the original two graphs. This is the idea behind knowledge conjunction.

As was discussed in Chapter One, attempts to define unification of CGs have either restricted the allowable CGs in some way [Müller 1997], or have restricted the unification to a simpler operation [Kocura 1996]. A join on a Conceptual Graph is a simpler operation than unification, and is usually used to merge two graphs. As was illustrated in the examples of Chapter One, unfortunately these attempts at merging graphs often lose some of the knowledge contained in the graphs being merged, thus violating the concept of knowledge conjunction.

2.7 Summary

Unification is a collection of techniques for finding an object (or structure, graph, etc.) which fits the description of two objects simultaneously. While unification has been studied for a long time, many divergent techniques exist for accomplishing unification in different domains. This thesis will formally present a general definition of unification of Conceptual Graphs.

Generalization, or anti-unification, is the dual function to unification. The Most General Unifier of two Conceptual Graphs is the most general graph which is more specific than the two graphs under consideration. In Chapter Five, it will be shown that a complete set of tools for Conceptual Graphs will include not only unification and Most General Unifier, but also constraint satisfaction, generalization and all of the canonical formation rules.

One contribution of this thesis is in contributing a definition of knowledge conjunction to the Conceptual Graphs community. Knowledge conjunction is a system in which two pieces of partial information can be combined into a single unified whole, as defined by Carpenter. We want to be able to combine the expert knowledge of a system, or even combine knowledge from different sources, not merely gather additional information.

Feature Structures are another graph-based knowledge representation scheme which is simpler in concept and manipulation than Conceptual Graphs. Their main advantages over Conceptual Structures are in the use of symbolic labels for their arguments, and in their freedom from fixed arity of arguments. While the Feature Structure formalism includes a well-defined and efficient unification technique for Feature Structure extensions, they lack a general solution to the unification problem, because there is no method for defining the unification of the intension of two Feature Structures. For example, in attempting to unify the intension of one Feature Structure ("some cat") with the extension of another ("the cat Felix"), there is no defined solution. This problem does not exist in Conceptual Graphs, because any binding is as good as any other under an existentially quantified concept. Intensionality precludes this type of subsumption as the solution to the unification problem in Feature Structures.

Constraint Logic Programming is a method which is used to decide the solvability of constraints in Constraint Satisfaction Problems. While there has been a great deal of research in CLP, most of this research has concentrated on making CSP algorithms and techniques more efficient. The common CLP languages and methods are not able to manipulate complex knowledge structures, as can Conceptual Graphs.

The unification-based approach to constraint processing computes projections and general unifiers. The constraint-based approach uses constraints to determine which instances are valid. Mugnier and Chein discuss how CSPs can be transformed into projection (or morphism) problems in order to find a solution. This type of technique can be exploited with CGs.

Conceptual Graphs have a well-defined formal representation scheme and can use techniques such as subsumption and lattice operations. However, CGs still lack a good reasoning capability. Techniques which have well-defined unification processes on knowledge structures, such as Feature Structures, only handle simple versions of the structures, and do not have the representation power that CGs have. Techniques for constraint solving have concentrated on efficient constraint algorithms, and have not produced efficient knowledge structure manipulation techniques. Standard unification techniques have never been applied to CGs, and do not take the structure of CGs into account.

A sound and complete unification technique for Conceptual Graphs would be the first step in creating a reasoning ability for CGs. This thesis will contribute to Conceptual Graphs by formally defining a unification model for CGs which can use and manipulate constraints over the graphs. The constraints will not only apply to the structure of the graph, but more importantly to the referent values in the concepts. The model defined in Chapter Three will use unification of Conceptual Graphs to propagate constraints on concepts through the reasoning process of a domain.

Chapter 3

An Algorithm for the Unification of Conceptual Graphs with Constraints

3.1 Introduction

Constraint Satisfaction Problems (CSP) are a well-understood and well-researched area, and many tools exist to handle CSP [Van Hentenryck et al. 1997; Frost and Dechter 1994; Gini and Rogialli 1994; Cohen 1990]. These languages and tools are excellent at handling real numbers, and simultaneous equations. However, most still fall short in representation power, as they lack sufficiently complex knowledge structures to represent the knowledge of any interesting domain, as will be discussed in Chapter Four.

This chapter discusses research in bringing constraint technology to formal structured knowledge representation. While the fields of CGs and CSP are still some distance apart, this project has made a start on bringing formal, structured knowledge representation techniques into contact with constraint technology. The significance of this work is that we demonstrate that the merging of these techniques is both efficient and very useful.

The basic objective of this work is to extend Conceptual Graphs not only by adding constraints, but then also by being able to guarantee that the unification of two such constrained graphs generates a new graph which is canonical, that is, structurally valid in the domain. It is very important that the generated graph is valid not only in terms of being a valid CG structure, but also in terms of representing valid knowledge in the domain under consideration. While we cannot guarantee that the fact represented by the graph is currently true (that's the arena of Truth Maintenance), we can guarantee that when two constrained Conceptual Graphs are unified, not only is the canonicity of the graph guaranteed, but we also produce a graphical representation of knowledge which is meaningful in the domain under consideration. We discuss this issue in more depth in section 4.4.4.

The method described here uses intervals to bound the value of an attribute, thus capturing the idea of a real number. Subsumption of an interval (defined on an interval lattice) is used to decide whether two concepts of the same type are still unifiable. The lattice operator "join", again defined for intervals, is used to decide subsumption. We are then able to define real-valued constraints in concepts, and use the standard Conceptual Structures operations of join, projection and subsumption to decide whether a concept is valid according to the domain knowledge. Part of the work presented in this chapter is taken from my earlier work in this area [Corbett 2001; Corbett 1999; Corbett and Woodbury 1999].

3.2 Interval constraints

3.2.1 Previous Use of Interval Constraints

The concept of using intervals to represent constraints on real variables is certainly not new. The work done by Cleary [Cleary 1997] concentrates on defining arithmetic operators for intervals in an implementation of Prolog. His method of constructive negation deals with quantifier elimination problems in constraints by reducing the answer to a disjunction of conjunctions of primitive constraints. The primitive constraints are defined as either $X = Y$ or $\forall Z X \neq Y$, where Z is the set of variables in the domain.

Cleary is mainly concerned with the problem of finding efficient methods for eliminating a negated existential quantifier. The problem is in finding efficient solutions to a (possibly infinite) assertion that there does not exist a solution to a formula. Cleary also formally defines the arithmetic operators over the constraints in this domain [Cleary 1997].

Older [Older 1997] discusses the lattice operations and gives a formal definition of an arithmetic over interval constraints on the reals. Older's definition of the partial order on intervals states that:

$[x_1, x_2] \geq [y_1, y_2]$ iff $x_1 \leq y_1$ and $x_2 \geq y_2$.

Older's lattice operations are defined by:

(join) $[x_1, x_2] \vee [y_1, y_2] = [x_1 \uparrow y_1, x_2 \downarrow y_2]$ and
(meet) $[x_1, x_2] \wedge [y_1, y_2] = [x_1 \downarrow y_1, x_2 \uparrow y_2]$,

where $a \uparrow b$ denotes maximum(a,b) and $a \downarrow b$ denotes minimum(a,b). Older calls these "extervals," and puts no restriction on the relation between x_1 and x_2. For example, the interval [3.1, -5.7] is allowed in these definitions. In order to avoid confusion in the system described here on interval constraints, we disallow the interval [x, y] where $x > y$. Older avoids defining the top and bottom of the lattice, in order to avoid "failure interpretations" in his algorithms. Failure interpretations in Older's system result from defining boundaries on the reals. Older chooses not to represent preset boundaries, but rather allows the lattice of the real intervals to be infinite [Older 1997].

3.2.2 An Interval Concept Type Hierarchy

It is possible to apply the idea of interval constraints to Conceptual Graphs. First, it is necessary to formally define what we mean by an interval type over concepts. We then describe how this concept type hierarchy can be used within the Conceptual Graph formalism to achieve a representation of constraints over the real numbers. Finally, we can define unification over these constraints.

We start by defining a concept type lattice of real-value intervals, as shown in Figure 15.

Definition 8. Interval. An interval is defined as: Intr: $[x, y] \subseteq \mathbb{R} \times \mathbb{R}: x \leq y$.

This definition gives us $\top = [-\infty, \infty]$ and $\perp$ is any absurd interval. Informally, the lattice is ordered on interval inclusion, such that two intervals have a join if there is some other interval which is in the "overlap" area of the two intervals. It is clear how such an operation on the lattice would be defined. A maximal join is the largest such overlap interval. Formally, we adopt the definitions discussed for

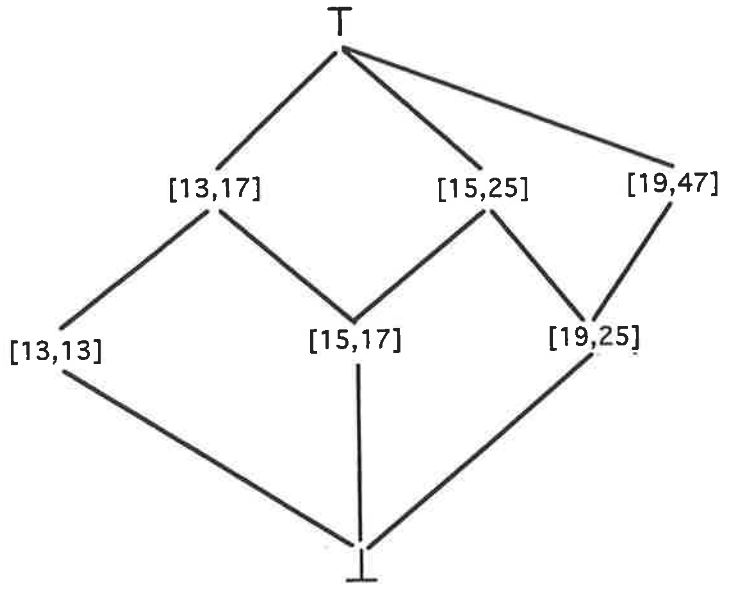


Figure 15. An example of an interval lattice, ordered by interval inclusion.

interval meet and join in [Older 1997].

These definitions give us the following desirable results:

$$[13, 17] \vee [15, 25] \rightarrow [15, 17]$$

$$[15, 17] \vee [19, 47] \rightarrow \perp$$

$$[15, 11] \rightarrow \perp$$

This lattice then defines the concept type hierarchy for an interval type. Two interval-type concepts can be joined if there is a join on the interval lattice for them. Constraints over real variables can then be expressed by specifying the interval into which the real value must be constrained. Note that a specific value can still be specified by using an interval: 13.7 can be expressed as the interval type [13.7, 13.7]. Unifying two real-value concepts then becomes finding the join of the concepts on the interval lattice. The set of individual markers for concepts can now be expanded to include intervals as defined here.

3.3 Projection and constraints using intervals

3.3.1 The Conformity Relation

The designs of the relations and algorithms presented in this thesis are based on the assumption that relations "know" what types they can validly point to. The Conformity Relation was defined formally in Chapter One, but the discussion of its use and function has been reserved until now, when we can put it into its proper context. The conformity relation relates type labels to individual markers. This is essentially the relation that ensures that the typing of the concepts makes sense in the domain.

For example, we can derive the graph in Figure 1 (reproduced again on the next page) from the more general graph shown in Figure 16 by simply applying the Canonical Formation Rules. Once the general concept has been specified, as shown in Fig. 16, the concepts can be specialized (or even unified). It is clear from earlier discussions how this is done. The conformity relation enters into the operations by specifying which individuals can be called "cat" or "object". That is, it is clear from the type hierarchies that a cat is a kind of animal, but the fact that Felix is a cat must be specified by the Conformity Relation. Once this type of declaration has been made, then the generic marker can be appropriately replaced by a specific individual.

In the case of the interval type, the Conformity Relation can be used to ensure that the real numbers being represented by the intervals are being used in an appropriate manner. For example, the conceptual relation "area" must relate a measure to a floorspace, room or similar. This can be enforced by the Canonical Formation Rules, but replacing the generic marker with a specific interval requires that the Conformity Relation validate the individual marker. In this manner, the intervals can be used to represent prices, lengths, measures or any other number concept.

3.3.2 The Headed Conceptual Graph

Müller demonstrates in his work [Müller 1997] that it is not always possible to find a common generalization for two arbitrary graphs in the same canon, and therefore it is not possible to guarantee that two graphs can be unified. If a subset of CGs is defined as all graphs that have a designated head node, and the head nodes are of compatible types, then these graphs will necessarily have a most general unifier. We define q, the user-selected head node of a graph, in order to be

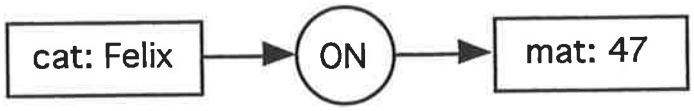


Figure 1. A Simple CG.

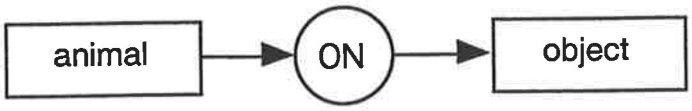


Figure 16. A General CG.

able to guarantee unification of the graphs under Müller's algorithms. This method allows us to combine the graphs while preserving the knowledge in both graphs, and still be able to use efficient unification methods as defined by Willems, Müller, and others. The specification of a head node gives us a starting point for a standard depth-first graph search algorithm modified for use with CGs. As will be discussed in a later section of this chapter, these algorithms have a complexity of O(n) rather than the O(n^2) that might be presumed for a general graph match algorithm. We therefore present a definition for a Headed Conceptual Graph as a slight modification (or perhaps restriction) of that for Conceptual Graph:

Definition 2′. Headed Conceptual Graph. A conceptual graph with respect to a canon is a tuple $G = (C, q, R, type, referent, arg_1, \ldots, arg_m)$ where

$C, R, type, referent,$ and $arg_1, \ldots, arg_m$ are as defined previously.

q is a distinguished member of C, the head or root node of the graph.

It is important to note here that both Sowa [Sowa 1984] and Leclère [Leclère 1996] (among others) define (or at least discuss the possibility of) a head node for a Conceptual Graph. In these definitions, the "head node" is really more of a category of the Conceptual Graph, in a kind of X′, government binding sense [Chomsky 1980]. This head node states the type of the graph and enforces appropriate specialization and generalization rules.

Müller's use of the head node is as an aid in resolution theorem proving. In fact, Müller's entire definition and use of unification is as a method for logical resolution [Müller 1997]. While the application of Müller's head node was originally for a different purpose, the sense used by Müller can still be used for general unification for knowledge conjunction.

3.3.3 The Projection Operator

The next step in defining constraints on concept markers is to define a projection operator, similar to Sowa's π operator [Sowa 1984], but which takes intervals into consideration. Sowa's π operator is used to identify that portion of a graph which is derived (by a join) from another graph. In other words, if two graphs, u and v, are joined to form graph w, then there will be in w a projection of graph v and a projection of graph u.

We again follow the definitions presented by Willems [Willems 1995] and by Chein and Mugnier [Mugnier and Chein 1996; Chein and Mugnier 1992]. In these works, projection is defined as a mapping between two graphs, rather than a sequence of canonical derivations as Sowa defined it in [Sowa 1984]. This allows the definitions of projection and specialization to become identical; the existence of a specialization implies the existence of a projection, and vice versa. Similarly, generalization can be defined as the inverse of specialization, using the basic definition of projection.

Note that the following definition is a re-wording of Definition 4 from Chapter One. The two major differences from the Chapter One definition are in including the head node and in allowing individual referents to subsume other individual referents. We use this notion of subsumption of individuals for some of the notions to be presented in Chapter Five.

Definition 4´. Projection on Headed CGs. $G = (C, q, R, type, referent, arg_1, \ldots, arg_m)$ subsumes $G' = (C', q', R', type', referent', arg'_1, \ldots, arg'_m)$, $G \geq G'$, if and only if there is a pair of functions π_C: $C \to C'$ and π_R: $R \to R'$, called morphisms, such that:

$\forall c \in C$ and $\forall c' \in C'$, $\pi_C(c) = c'$ only if $type(c) \geq type'(c')$, and
$referent(c) = *$ or $referent(c) \geq referent(c')$, and by extrapolation,
$\pi_C(q) = q'$,
$\forall r \in R$ and $\forall r' \in R'$, $\pi_R(r) = r'$ only if $type(r) \geq type'(r')$
$\forall r \in R$, $arg'_i(\pi_R(r)) = \pi_C(arg_i(r))$,
$\forall c \in C$ there is a unique concept $c' \in C'$, such that $\pi_C(c) = c'$

This definition of projection is very similar to previous definitions [Corbett 2001; Leclère 1997; Müller 1997; Mugnier and Chein 1996; Cogis and Guinaldo 1995; Willems 1995; Carpenter 1992], except that here we define a head node, and we specifically allow a referent marker to subsume another marker in the projection. The notion of an individual subsuming another individual (a property often restricted to the generic marker) is not novel. The idea has wide acceptance in, for example, the use of conceptual graphs in Formal Concept Analysis (FCA) [Wille 1997] . In FCA, an individual is lower on the concept lattice if it is more specified, or simply has more properties than another individual. In some domains in which FCA is used, it makes sense to derive an ordering on this hierarchy. This sense of an individual subsuming another which is more specified or more specialized is the sense that is also employed in this work.

3.3.4 Projection and Interval Constraints

In our method, when two individuals of type interval are joined, their join may be another interval which is contained in both of the originals. This leads to a situation where, after joining two concepts, it may not be possible to find either of the original intervals exactly as they were before. It may be a joined interval, or similar concept type. The new type would be a specialization of the previous types. In the research discussed here, an interval must be able to have a more specific type, in order to be useful in the domains that we work in. This is the significance behind defining a type lattice for intervals.

The other properties of the standard π operator remain the same as in previous work. There is still the guarantee that for each relation r in the graph in question, the old type must subsume the new type, and that the i-th arc of the old relation must point to the same concept as the i-th arc of the new relation.

The standard definition of the π operator from [Sowa 1984] applies to two graphs, one of which subsumes the other. In our research, combining the knowledge of two graphs (even if there is no subsumption relationship between them) into a more refined representation is more important. The π operator as used and further refined here helps define the unifier, so that unification can proceed with combining the two graphs into one.

3.4 Unification over constraints

The unification of two conceptual graphs with constraints now becomes the combination of two graphs which are compatible in corresponding concepts and relations, as defined by our definition of the projection operator and join. We again employ a definition from Willems [Willems 1995] to complete the discussion of unification:

Definition 9. Most General Unifier. The most general unifier (mgu) of two CGs is the most general common specialization of the two Conceptual Graphs. Let G'' be a glb of G and G'. G'' is the most general unifier of G and G' if, for any Conceptual Graph U where $G \geq U$ and $G' \geq U$, either $G'' \geq U$ or $G'' = U$. By this definition, the following conditions must hold true:

1. projection: projections $G \rightarrow G''$ and $G' \rightarrow G''$exist, and
2. compatibility: for any concept in G'', the images $c \in G$ and $c' \in G'$ must be compatible, i.e. $type\,(c) \vee type(c') \neq \perp$.

The compatibility condition simply states that it is useless to discuss a unifier, if no possibility of unification exists. In the Unify algorithm, we will discuss that completeness dictates that an answer must always be found to the unification process, even if the answer is $\perp$.

Also note that this definition of Most General Unifier is nearly identical to the definition of Greatest Lower Bound. In fact, we take Most General Unifier to be exactly the Greatest Lower Bound of two Conceptual Graphs. If we can find the Greatest Lower Bound, then we have accomplished unification of the two Conceptual Graphs.

This leads to our formal definition of Unification:

Definition 10. Unification. The unification of G and G' is the graph G'' such that:

1. projection: there exist morphisms $G \rightarrow G''$ and $G' \rightarrow G''$, and
2. mgu: G'' is the mgu of G and G'.

3.5 Unification Algorithm

3.5.1 Unification Algorithm Informally

We can now employ standard graph spanning and unification algorithms which recursively move down through the relations and concepts from the head node [Aho et al. 1974]. Informally, our algorithm starts by comparing the head nodes, q and q' for compatibility, that is, that they have the same type or that the type of one subsumes the type of the other, and that the referents are either equal or that one of them subsumes the other. Given compatible head nodes, the algorithm selects a relation r from all of the relations that lead from the head node q, and seeks its projection r' in the second graph from the relations that lead from q'. If none is found, then the relation r becomes part of the unified graph trivially. If the projection is present in the second graph, then the unification algorithm is called recursively with one of the concepts c pointed to by the relation r used as the head node. The concept c is marked "old" to aid in cycle detection on the graph. If c is already marked "old", then this branch of the graph is not examined any further, as it has already been examined. If all these concepts c prove to be compatible, then the two subgraphs are joined, and attached to the unified head, q''. If one relation in graph G is incompatible with its projection in G', then the unification fails. The algorithm proceeds depth-first through the graph, to the next relation. When all relations attached to the head concept in both graphs have been successfully processed in this way, the algorithm terminates successfully.

Note that this algorithm makes no attempt to find other relations r' in G' which might "match" the relation r. The algorithm matches with the first compatible r' in G'. The issues of possible backtracking problems which this approach might entail are discussed in detail in section 4.4.1. As for finding the corresponding c' for each c, the algorithm is able to exploit the semantics of relations and the orderedness of the relation arguments to look in the one place where c would have a projection. If there is no c' there, then there will not be a projection for c.

3.5.2 Unification Algorithm Formally

In the previous section, the algorithm for unification was discussed informally. Here, the algorithm is laid out in a more formal manner, and the complexity issues are discussed. The algorithm is displayed in Figure 17, and issues of soundness and completeness are discussed in the sections that follow.

The Unify algorithm is guaranteed to terminate for finite CGs, since the algorithm is based upon examining and performing operations on the relations of each graph. The algorithm imitates a depth-first graph search which detects cycles. At each level of the graph, one node is selected, and its children nodes are explored, also in a depth-first manner. The algorithm performs a finite number of steps on a finite number of relations in each graph. To formalize the idea of termination of the Unify algorithm, the following theorem is presented.

Theorem 2. Termination. Algorithm 1, Unify will terminate for finite Canonical Conceptual Graphs.

Proof. The algorithm is based on creating a finite set, M_R, of the relations that are adjacent to the concept under consideration ($arg_0(r)$). There are a finite number of relations $r \in M_R$ for each recursion, and there are a finite number of recursions (based on the number of concepts $c \in M_C$ in the graph, and the fact that cycle detection is performed on the set of concepts), and each recursion has a finite number of steps. $\square$

```
Algorithm 1. Unify $(G, G')$.
• If $q \sqcap q' = \bot$ then terminate with failure.
• Mark all $c \in C, c' \in C'$ as "new."
• Construct the set $M_G$ of all $r \in R$ and $r' \in R'$.
• Search $(G, G')$.
• While there is an $r \in M_G$ :
        • Construct a new $G$ and $G'$ by selecting an $r$ from $M_G$ .
        • Search $((C, arg_0(r), R, type, referent, arg_1, \ldots, arg_m)$ ,
                  $(C', arg'_0(\pi_R(r)), R', type', referent', arg'_1, \ldots, arg'_m))$
        • Remove $r$ from $M_G$ .
• Return $G'' = (C'', q'', R'', type'', referent'', arg''_1, \ldots, arg''_m)$ .
Function Search $(G, G')$.
• Construct the set $M_R$ of all $r \in R$ and $r' \in R'$ where $arg_0(r) = q$ and $arg'_0(r') = q'$.
• While there is an $r \in M_R$:
        • If $\pi_R(r) \notin M_R$ then set $q'' = arg_0(r)$.
        • Construct the set $M_C$ of all $c, c'$ such that $arg_i(r) = c$ and $arg'_i(\pi_R(r)) = c'$
              for all $i, 0 < i \leq arity(r)$.
        • While there is a $c \in M_C$:
            • If c is marked "old" then remove $c$ from $M_C$. Else:
                • Mark $c, \pi_C(c)$ "old", and remove $c, \pi_C(c)$ from $M_C$ .
                • If $c \sqcap c' \neq \bot$ then set $arg''_i(r) = c \sqcap c'$, else terminate with failure.
                • If there is an $r \in M_G$, such that $arg_0(r) = c$, or $arg_0(\pi_R(r)) = c'$, then:
                    • Search $((C, c, R, type, referent, arg_1, \ldots, arg_m)$ ,
                              $(C', c', R', type', referent', arg'_1, \ldots, arg'_m)$ ).
        • Set $arg''_0(r \sqcap r') = q''$.
        • Remove $r, \pi_R(r)$ from $M_R$ and $M_G$ .
• Return $G'' = (C'', q'', R'', type'', referent'', arg''_1, \ldots, arg''_m)$ .
```

Figure 17. The Unification Algorithm.

3.5.3 Complexity of the Unification Algorithm

Chein and Mugnier discuss the complexity of projection, equivalence, irredundance and related operations for a formalization of Conceptual Graphs, which they call S-graphs [Chein and Mugnier 1992]. Their S-graphs are bipartite graphs "of any structure," with no bound on the size or structure of the graph. The structure of these possibly infinite graphs are what leads to the nondeterminism in Chein and Mugnier's algorithms. Projection under these properties becomes a generalized graph isomorphism problem on possibly infinite, possibly connected graphs, with no general solution. They demonstrate that projection (in its various forms) is NP-complete in their system.

Chein and Mugnier discuss, however, the development of the S-tree, which is a subset of the S-graphs, with no cycles and a strict tree structure. They then demonstrate polynomial solutions of projection of S-trees.

The addition of the head node, and the restriction that the Conceptual Graphs be finite are the two attributes that make the algorithms in this thesis tractable. We do not adopt the additional restrictions of Chein and Mugnier of a strict tree structure and no cycles. The Unify algorithm described in this section deals with general graph structures which have a head, and which can have cycles. The unique head node guarantees that the graphs are not connected. It also guarantees that they are not the same graph. The requirement for finite numbers of concept and relation nodes keeps the algorithms from imitating the halting problem.

The Unify algorithm proceeds by starting with the head node q, chooses one of the relation nodes being pointed to by q, and then identifies all of the concepts that it points to, and searches for a corresponding and compatible relation/concept combination in the other graph. The Most General Unifier of two headed Conceptual Graphs can be found by restricting the types of both head concepts to their join (and restricting their referents if necessary), and then proceeding to repeat the procedure with the concepts pointed to by the relations from the head nodes.

From the description of the algorithm in the last few sections, it can be seen that the unification of two CGs amounts to visiting each relation node of each graph once, deciding the join of the nodes (if necessary) and then copying the join of the two nodes into a new Conceptual Graph. This is the method used in a general depth-first search of a standard graph [Aho et al. 1974]. Our lattice operations are implemented by simple comparison operations of linear complexity. The G'' can be found in a finite time because a join on a lattice of intervals can be accomplished in constant time, even if the join is bottom. There are implementations of lattices that represent their elements by bit strings and have a join operation of constant complexity [Ellis 1995; Aït-Kaci et al. 1989]. The copying operation can safely be assumed to be of a linear complexity, and when combined with the lattice operations as described, a linear complexity of the algorithm is assured.

Theorem 3. Unify has complexity O(n). Algorithm 1 requires O(n) steps on two graphs with a sum of n relations in the two graphs.

Proof. The initialization of marking all concepts "new" and the selecting of "new" concepts require O(n) steps if a list of relations is made and scanned once. The time spent in the M_C loop is proportional to the number of relations adjacent to a concept *c*, which is the number of links incident on concept *c*. Therefore, summing over each concept, the algorithm has a complexity proportional to the number of relations in the graphs. This loop is only called once for a given *c*, since *c* is marked "old" the first time through the loop. Thus the total time spent in this loop is O(n). □

3.5.4 Soundness of the Unification Algorithm

Our discussion of the soundness of the Unification Algorithm centers around two concepts: that the algorithm produces a valid Conceptual Graph; and that the algorithm produces the Greatest Lower Bound of the two input Conceptual Graphs. We first show that the result of unifying two Conceptual Graphs is a Conceptual Graph.

Theorem 4. Unification produces a CG. If G'' is the unification of G and G', then G'' is a Conceptual Graph.

Proof. $G'' = (C'', q'', R'', type'', referent'', arg''_1, \ldots, arg''_m)$ is finite (i.e., C'' and R'' are finite) because G and G' are finite.

The function *type* still works as a valid function, since for any $c \in C$, $type(c) \geq type(c'')$, by the definition of projection.

Each arg_i is still a valid partial function, since all $arg_i(\pi_R(r)) = \pi_C(arg_i(r))$, by the definition of projection. □

Recall from earilier discussions that this algorithm makes no attempt to find other relations r' in G' which might "match" the relation r in G which is currently under consideration. The algorithm matches with the first compatible r' in G'. The issues of possible backtracking problems which this approach might entail are discussed in detail in section 4.4.1. So, the Greatest Lower Bound in this work is the *first* glb which matches the graph under consideration.

Theorem 5. Unification produces a Greatest Lower Bound. The unification of G and G' is a greatest lower bound of G and G' in a canon, if G and G' have a lower bound.

Proof. Suppose that $G'' = (C'', q'', R'', type'', referent'', arg''_1, \ldots, arg''_m)$ is a lower bound of G and G'.

Define a morphism h: $h \in C \times C'$

$$h(c'') = \begin{cases} c & \text{if } c' \geq c \\ c' & \text{if } c \geq c' \\ c \vee c' & \text{if incomparable} \\ \perp & \text{otherwise} \end{cases}$$

and another morphism h: $h \in R \times R'$

$$h(r'') = \begin{cases} r & \text{if } r' \geq r \\ r' & \text{if } r \geq r' \\ r \vee r' & \text{if incomparable} \\ \perp & \text{otherwise} \end{cases}$$

Also, the types would be joined on their type hierarchies:

$type(c'') = type(c) \vee type(c')$

$type(r'') = type(r) \vee type(r')$

so that $\pi_C : G \rightarrow G''$ and $\pi_R : G \rightarrow G''$ and $\pi_C : G' \rightarrow G''$ and $\pi_R : G' \rightarrow G''$

By the definitions of projection and subsumption, the unification of G and G' is a lower bound of G and G'.

It now only remains to show that the unification of G and G' is a greatest lower bound of G and G'. Consider that $G \sqcap G'$ is defined as $G'' = (C \cup (C' - \{c'\}),$ $R \cup (R'_{c'':=c \vee c'}),$ $type'',$ $referent'')$. Again, the subscript $c'':=c \vee c'$ denotes the replacement of every occurance of c'' by $c \vee c'$. The functions *type* and *referent* are such that: $f'\upharpoonright c \equiv f$, $f'\upharpoonright c' \equiv f'$. Recall from previous discussions that type hierarchies are restricted to lattices in this work, so that the join operator is well-defined and produces a unique result.

The unification of G and G' will then be a greatest lower bound, since the lattice operators as used in the Unify algorithm produce unique results. The first graph encountered into which the two graphs G and G' both have projections will be used as the greatest lower bound.

Because of the subsumptive nature of the type hierarchies, if there is another lower bound U which satisfies the above conditions, then either G'' subsumes U, or U is an alphabetic variant of G''. That is, either $G'' \geq U$, or $G'' = U$. (It is possible, however, that even if U is an alphabetic variant of G'' they may be unique, separate Conceptual Graphs.) □

3.5.5 Completeness of the Unification Algorithm

Here, we will argue that the join and unify operators are complete, even given the constraints. The proof given here will demonstrate that all possible graphs are produced by unify, or by a simple join.

The argument is that finding a G'' which contains an interval which is the glb of some G and some G' is accomplishable in finite time, given that a backtracking algorithm can be used when needed. If the algorithm can guarantee that the answer is always decidable in finite time, then the function is total and there is completeness of the expressibility of the knowledge in the domain.

Theorem 6. Completeness of the Unification Algorithm. Unification of two Conceptual Graphs is a total function.

Proof. Theorems 4 and 5 state that the Unify algorithm applied to two graphs, G and G', will produce their Greatest Lower Bound (which may be $\perp$). Theorem 2 guarantees that this result will be found in finite time. Therefore, if G and G' have a Greatest Lower Bound, then the Unify algorithm will find it in a finite amount of time, which is to state that all possible correct results will be returned by the Unify algorithm. □

3.6 Summary

This chapter presents several significant results in formally defining the unification of Conceptual Graphs over constraints. First, the definition of a real-value constraint type hierarchy was defined for Conceptual Structures, and the formal definitions were presented. The system described here allows real numbers to be represented in concepts, and also allows those values to be constrained by specifying valid intervals. In the software as implemented, inequality relations and variables in the constraint specifications are also allowed. As will be demonstrated in the next chapter, the experiments with this software have shown these techniques to be useful and efficient.

In the methods discussed in this chapter, the intent was to attempt to improve on previous work in several ways. The major step is in formally specifying constraints for Conceptual Graphs, but this work also points the way toward work in defining the major constraint processing techniques from the Constraint Satisfaction Processing (CSP) community for Conceptual Graphs. Further work in this area would allow the entire semantics of CSP to be included in the CG formalism. This would be a major benefit to the researchers and users of both communities, as it would bring the entire knowledge representation formalism and structures of CGs (concepts, canonical formation rules, join, type subsumption, etc.) into the CSP work. The CG community would benefit by having formal definitions and algorithms for implementing standard constraint methods within the CG formalism.

The algorithm presented in this chapter has a linear complexity. This compares favorably with the complexity of other graph operations, such as simple sorts or tree isomorphism. This is a significant improvement on previous work in that previous efforts at unification of Conceptual Graphs either restricted unification to a simpler operation, or had a high complexity (in fact, NP-complete in the most elaborate system).

At the close of Chapter Four, a tree isomorphism unification procedure will be defined, which has an extremely efficient algorithm, but which places severe restrictions on the structure of the Conceptual Graphs used in the procedure. It will be shown that a tree isomorphism unification technique for a restricted set of CGs is a useful and practical device, but can only be used in certain domains.

The techniques and methods presented in this chapter are sound and complete, as proven earlier in this chapter. This improves on previous work, where projection and unification were found to be NP-complete. Completeness was acheived in our algorithm by restricting the Conceptual Graphs to a specific subset, Finite Headed Conceptual Graphs. While this does literally restrict the types of knowlege that can be represented in the graphs, this subset still contains an extremely useful set of structures, which can be used for many domains. In the next two chapters, we explore some of these domains, and demonstrate the utility of these structures.

Further, we assume extremely efficient lattice operators to be available for the operations we perform here. Our own lattice operators use simple techniques to determine subsumption of real intervals. These operators have a simple, efficient implementation, but we look forward to using the type of efficient lattice operators that are being defined in other parts of the Conceptual Graphs community.

Representing intervals as types will also have its pitfalls. The projection mechanism will guarantee the compatibility of types, as it would for any type. There is a possibility, however, in some applications where there is a large number of such intervals that it would become computationally expensive to perform the lattice operations, notwithstanding the assumption of efficient lattice operators discussed above. The two main method which can be used to circumvent this problem are to precompute the closure of the type lattice over all possible intervals, or to dynamically compute what could be a very large lattice. The cluttering of the type lattice will depend on the number of interval joins that can be inferred in the lattice. In the domains that we explore in this thesis, this is not a problem. While it is theoretically possible for a lattice to be designed which will overwhelm the system, nevertheless, this lattice design solves a problem identified in [Mineau and Missaoui 1997]. In that work, actors are used as a patch over a conceptual system in order to constrain the referents of concepts in Conceptual Graphs.

There has been previous work in representing interval constraints for various implementations. One implementation for Prolog concentrated on the negation of existentially quantified varaibles, while another defined lattice operators for interval constraints on real variables. While the algebras of the former project have little to do with Conceptual Graphs, they do represent an interesting step toward bounding real values in an Artificial Intelligence domain. The latter project defines only algebras for interval constraints, but does not discuss these constraints in the context of any implementation or domain. We borrow some of the ideas presented in these works, and apply them in our work to constraining real values in the concepts of Conceptual Graphs.

The formal definitions and the descriptions in this chapter lead to a useful and efficient implementation of constraints over Conceptual Graphs. This work improves on previous work in allowing real numbers to be expressed in the concepts, and in allowing constraints to be placed on those values in the concepts. The constraints are defined as a concept type, and therefore can be used as a type in the normal way with conceptual graphs. The constraints are enforced in the unification and projection operations, as defined in this chapter. If a projection operation violates the constraints on one of the concepts, the join fails. The unification algorithm used in this system is based on a standard algorithm, and is guaranteed to terminate since we restrict the CGs to a special subset, finite headed Conceptual Graphs.

Chapter 4

Results, Demonstrations and Comparisons

4.1 Introduction

4.1.1 Domains

The main focus of this thesis so far has been the formal definition of a unification and constraint tool over conceptual structures. We now demonstrate the significance of this extended Conceptual Graph model defined in the first three chapters. This chapter will now present the results of applying that tool to real-world domains.

As will be detailed in the following sections, much of the testing has been done in the domain of architectural design. This is the main demonstration of the major theoretical work in this thesis, and will concern the definition, retrieval, and unification of pieces of design knowledge.

4.1.2 Implementation and Testing

The theory, concepts and algorithm discussed in previous chapters were implemented in Allegro Common Lisp on a Sun Workstation. All of the relations were implemented as lisp functions, and all data structures were lisp lists. Many different types of designs were detailed, in order to cover a wide range of generic design problems, all type hierarchies and subsumption problems, and various relations. These designs were unified in various combinations, in order to test the functionality of unification with and without constraints of various kinds, and to demonstrate the usefulness of the software. The combinations represented the types of unification problems that can be encountered. These types of problems are detailed and demonstrated in Sections 4.3 and 4.4.

In the software as implemented, we included a simple scheme for variable binding, variable arity relations (to be discussed in Chapter Six) and various arithmetic relations, such as inequality operators. These deviations from and additions to the theoretical work of the previous three chapters will be detailed in Chapters Five.

4.2 Design

4.2.1 Computational Design Tools

In this section, we will introduce one domain which has been used for testing the concepts and algorithms presented in the preceding chapters. The design domain has been chosen for several reasons; among them are an access to the data from an automated design project and ease of translation of design concepts into Conceptual Graphs.

There have recently been many research forays by the design community into computational design tools which will give the designer useful structures which can be combined and constrained in useful ways [Burrow and Woodbury 1999; Chang and Woodbury 1996; Heisserman 1995]. There have also been attempts in the CG community to assist in defining methods and techniques which will be useful in computational design [Woodbury et al. 2000; Mineau and Miranda 1998; Corbett and Burrow 1996].

The results discussed in this chapter are those recorded from the application of the Conceptual Graphs constraint tool operating over a design domain. The domain knowledge is represented in Conceptual Graphs with constraints. Here, we demonstrate the idea behind the unification mechanism by employing order sorted unification and constraints within the domain of architectural design. This mechanism leverages type representation technology from the Conceptual Graph community [Ellis 1995] and constraint technology [Mackworth 1992].

4.2.2 Search as a Metaphor of Design

Designers have borrowed Artificial Intelligence techniques to explore design possibilities and models. While AI researchers may know these techniques as discovery or search through a space of designs, designers see them as guided movement through design possibilites [Woodbury et al. 2000]. The point of automated search for the designer is to use computer media that engage designers in exploring design modifications. The design user may want to create new designs, or index, compare or adapt existing designs. This type of user requires efficient representations for the designs and states (of designs) in a symbol system. The designer needs to be able to represent spaces of possibilities which are both relevant to this exploration metaphor and lend themselves to tractable computations. It is necessary for the design process that the information in the system can be ordered by specificity, since design exploration usually means starting from an under-specified design and proceeding to a more specialized state.

This constraint has led us to consider state spaces structured by information specificity. Under this assumption, it turns out that two crucial properties in a design space exploration system are intensionality and partialness, as discussed previously. Type hierarchies, subsumption and conceptual relations are used to realize these concepts.

4.2.3 SEED

These design concepts are made concrete by considering a recent project in design decision space exploration. This section is an introduction to a system which will benefit from an expanded unification definition on Conceptual Graphs. Here, we detail the SEED project. The intent of the SEED project is to create software which will support preliminary design of buildings [Woodbury et al. 2000; Woodbury et al. 1999; Chang and Woodbury 1996; Corbett and Burrow 1996; Heisserman 1995; Heisserman 1991]. This includes using the computer as an active tool which helps to generate designs [Flemming and Woodbury 1995].

SEED is an acronym for Software Environment to support the Early phases in building Design. Specifically, SEED will help with recurring building types (designs which are used again frequently). The SEED system is intended to be an aid to architects in creating building designs by reusing design knowledge. In order to store and reuse the design cases in an efficient manner, it is necessary to use a representation scheme which can handle real-value constraints and unification.

The philosophy behind SEED is that much design work is based on reusing previous designs, and that design practice may be captured in the collections of operations used to construct classes of designs. Annotated solid models have allowed the formal representation of designs, and shape grammars operating over these annotated solid models have been interpreted as formalizations of the constructive operators [Heisserman 1995]. A similar system of reusing previous designs has been described by Guy Mineau [Mineau and Miranda 1998], as was discussed previously.

The SEED system is built around the idea of a design space. The design space is a set of partial or complete solutions to an architectural design problem. In this sense, it is roughly equivalent to the AI term "search space", in that the design space is defined by starting states and operators which allow the derivation of one state from another, including some acceptable goal states. The current state is the current focus of interest of the system. Since SEED is always restricted to the domain of building-design problems, it is sufficient to call the search space a design space.

The basic knowledge structures of SEED are described in [Chang and Woodbury 1996] [Woodbury et al. 1999] and [Corbett and Burrow 1996]. They include the *functional unit* (FU), which is a specification of the functional properties that a design must satisfy, the *design unit* (DU), which is the design solution to the problem presented by the FU, and the *specification unit* (SU) which describes the constraints on the FU which must be met by the DU. The SU contains not only static constraints, such as color or material, but also real-value constraints, such as a range of values which would satisfy a constraint on the area of a room. The SUs are therefore the fundamental building blocks in SEED, and can be represented as conceptual graphs arranged in a tree structure, with the fundamental concept of the SU at the root and the attributes of the SU as the branches from the root.

Figure 18 shows an example SU for a house, where the relation "has-parts" (of arity 4) is used to divide the house into its *private, public,* and *outdoor* sections, and then to consider each section.

SEED works by exploring the design space during the elaboration of a design. To achieve the goal of design experience reuse, SEED allows for the storage of "interesting" design states, where "interesting" is decided either by the user, or by the interaction of the user's search path and heuristics in SEED. The difficulty, then, is in identifying and retrieving stored design states containing design decisions most applicable to the current state, that is, retrieving useful information corresponding to design experience captured in the historical pattern of explored and stored design states.

As SEED explores the design space, each of the retrieved designs must be compared to the requirements to find whether the design meets each of the specifications and constraints. The problem then, is to find a previous design which will unify with the SU specification currently being worked on. This unification process must attempt to identify each attribute of the specification with the same attribute in the retrieved design. If all the attributes can be unified while

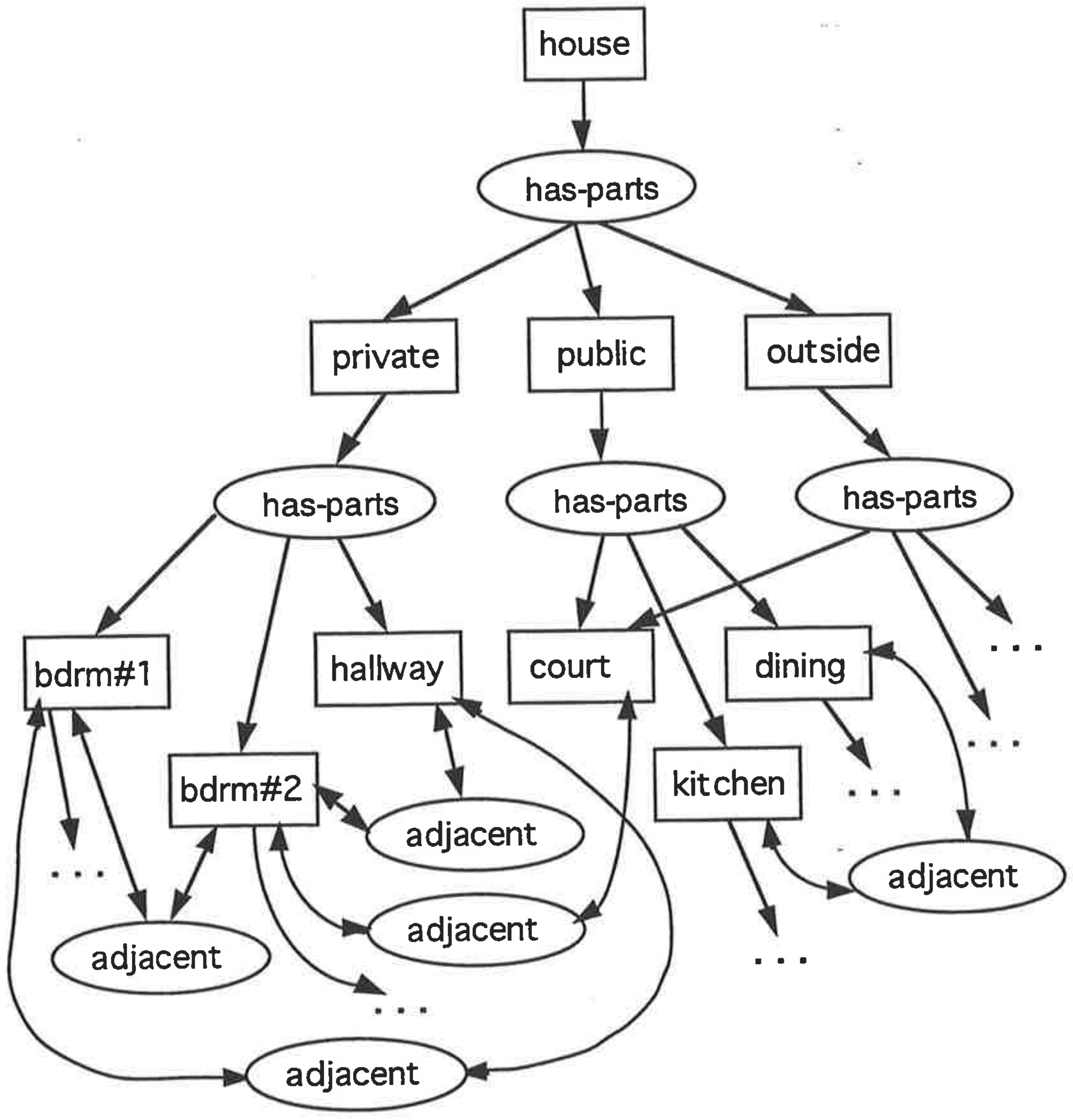


Figure 18. A SEED partial design represented as a Conceptual Graph.

satisfying all of the constraints, then the two structures are said to unify, producing a new structure which is a combination of the knowledge in the two previous structures. The new structure is a new design, which can be used to satisfy the current design requirements.

4.2.4 Specification Units as Conceptual Graphs

This section presents an example of some of the types of concepts and relations which can be used in the SEED domain. We introduce here only a subset of the possibilities as an illustration of the SEED domain. A complete description

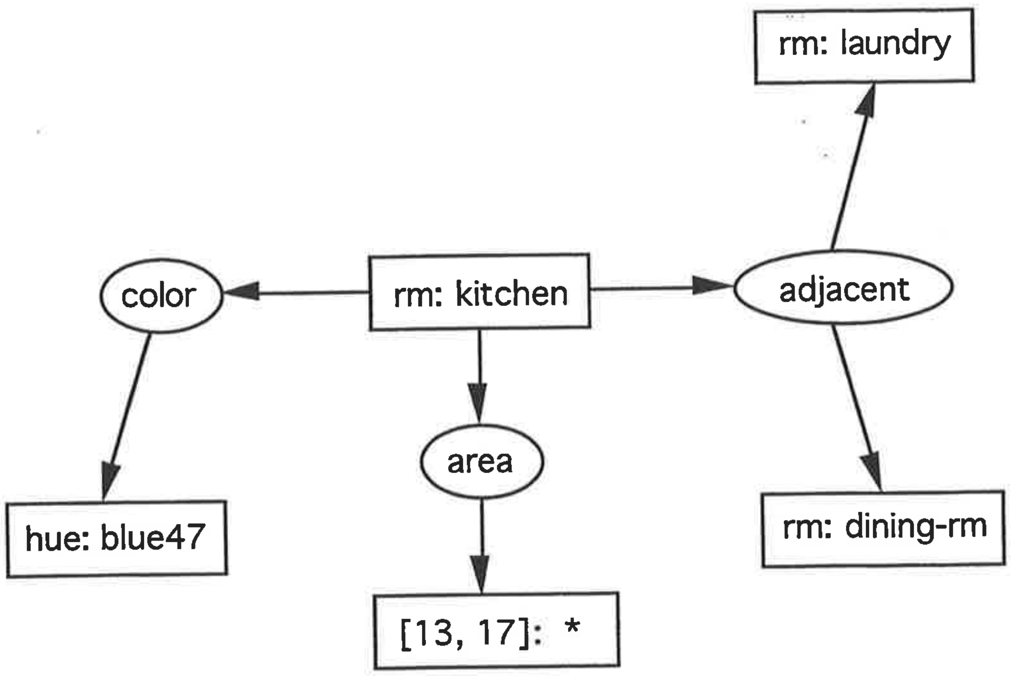


Figure 19. A Conceptual Graph representation of a kitchen SU.

and type hierarchies of these concepts and relations will be detailed in the next section.

In Figure 18, the SU for a house is shown with its *has-parts* relations, among others. In this diagram, an ellipsis indicates that other parts of the graph continue, but that they have been omitted from this figure to save space. Also, the types of the concepts have been omitted, in order to simplify the presentation of the diagram. The concept type applicable for most of the concepts in this figure would be "room."

Also note in Figure 18 that the relation *adjacent* is shown as a binary relation. In the diagram, each *adjacent* relation is shown with double-headed arrows. This is a diagram shorthand, which indicates that, for example, not only is the hallway adjacent to bedroom #2, but bedroom #2 is also adjacent to the hallway. These are two separate relations in the graph, but shown as a single relation with double headed arrows here.

Since SEED works on the principle of constructive design, it is important to be able to create small units in the design, and then link them together. The mechanism we use to link these units is unification. For example, the partial design illustrated in Figure 18 contains the concept *kitchen*. This may initially be used as a single concept, or as a link to the standard or template attributes of a kitchen. These generic concepts would be specialized later. Figure 19 shows a design for a kitchen, with some of the usual relations. (Figure 19 returns to the standard of specifying the concept type.) This can be thought of as an extension of the kitchen concept in Figure 18 (in other words, Figure 19 could be the diagram referred to by the ellipsis near the *kitchen* concept in Figure 18).

The *area* relation indicates that the value of the area is an interval, which constrains the values that the area may have. Other attributes may include insulation, illumination, support structure and other factors which concern the design of a building structure. We define some of these relations as follows:

color : $c \rightarrow$ *hue* (for any concept node $c \in C$)

has-parts : $c \rightarrow Rm \times Rm \times Rm$, (of arity 4)

area : $c \rightarrow$ *Interval*

adjacent : $c \rightarrow Rm \times Rm$, where

hue is a concept type, which refers to specific color hues,

Interval is the interval concept type, defined previously, and

Rm is a concept type, which is the room labels.

We assume that these types and their respective type hierarchies have already been defined, with their obvious meanings. We detail some of these types and type hierarchies in the following section.

Note however, that we have changed the arity of the *adjacent* relation. In Figure 18, we specified adjacent as a binary relation, but in Figure 19 it has an arity of three. It is consistent with the Conceptual Graphs literature to define an *adjacent*\2 relation and an *adjacent*\3 relation. These would then be treated as separately defined relations that could not be joined (unless the user wanted to find some way of defining these relations on the same relation type hierarchy, and defining what a join would mean). For now, we will keep similar relations of different arity separate, in order to avoid confusion. Figure 19 will now be considered separately from Figure 18, with a separate definition of *adjacent*.

In Chapter Six, we explore the concept of defining relations with a *variable arity*. Such a variable arity would allow us to join the relations in Figure 18 with the relations in Figure 19. Variable arity raises other problems in the syntax of Conceptual Graph joins and projection, as well as the fundamental meaning of some relations. These problems will be considered in Chapter Six.

4.3 Experiments: Simple Structures

This section of the thesis will describe some of the simpler experiments and structures that were used to test the unification algorithm. Essentially, this section will demonstrate the basics of the unification theory and algorithm, while the following section will explore some of the more difficult structures, special cases and problems.

4.3.1 Unification of Specification Units

The unification of two SUs is another SU representing neither more nor less information than is contained in the two SUs being unified. Unification only fails when it is applied to SUs that, when taken together, provide inconsistent information. In the case of SUs, inconsistency can only arise from attempting to unify two structures that assign incompatible values (in either literal information, or in the lattices defined for that type of information) to the same attribute.

Informally, the process of unification starts by identifying the head nodes of two SUs, *S* and *S´*. We then proceed to examine each attribute (relation) in *S*, and search for the same attribute type in *S´*. If that attribute type is not found in *S´* then that attribute unifies trivially. If that attribute type is found, then either the values must be equal (if literal values, such as color) or they must have a join, or the values must not be incompatible (eg, if one SU is adjacent to "dining" and the other is adjacent to "laundry" these SUs are still compatible). If the values do not meet one of these criteria, then they are incompatible, and unification fails. Continue until all attributes of each SU are either found to be compatible, or until one fails.

In the SEED domain, there is more of a sense of "constructing" the solution to a design problem, rather than resolution. SEED users will want to find specifications for partial designs which help to solve the current design problem. However, while the join operator will sometimes suffice, unification will be necessary when it is important to retain all the information being offered in two graphs. SEED will need to retain all the constraints and restrictions on the design, without resolving away specifications.

4.3.2 Example of Specification Unit Unification

We now demonstrate the software on sections of the SEED knowledge base. This example assumes that some partial designs have been entered into the SEED knowledge base as Conceptual Graphs, including the CG shown in Figure 19, which is reproduced on the next page, for easy comparison. Figure 20 is an example of a kitchen SU specified by the user to have one adjacent room and a floor area constrained to be in the interval [15, 20]. When we try to find a match for the specified partial design among the previous designs, we retrieve the Conceptual Graph shown in Figure 19 as a previously-designed kitchen from the knowledge base. This is unified with the SU shown in Figure 20, and the result is as shown in Figure 21.

The adjacent relations unify by taking on the values of both "dining-rm" and "laundry", since these two values are compatible (ie there is nothing to exclude the kitchen from being adjacent to both the laundry and the dining room).

The area relation unifies, because the intervals specified in the two original SUs have a join on the interval lattice. The join becomes the value of the unified area relation. The color relation unifies trivially, as there is nothing specified which could be incompatible with it.

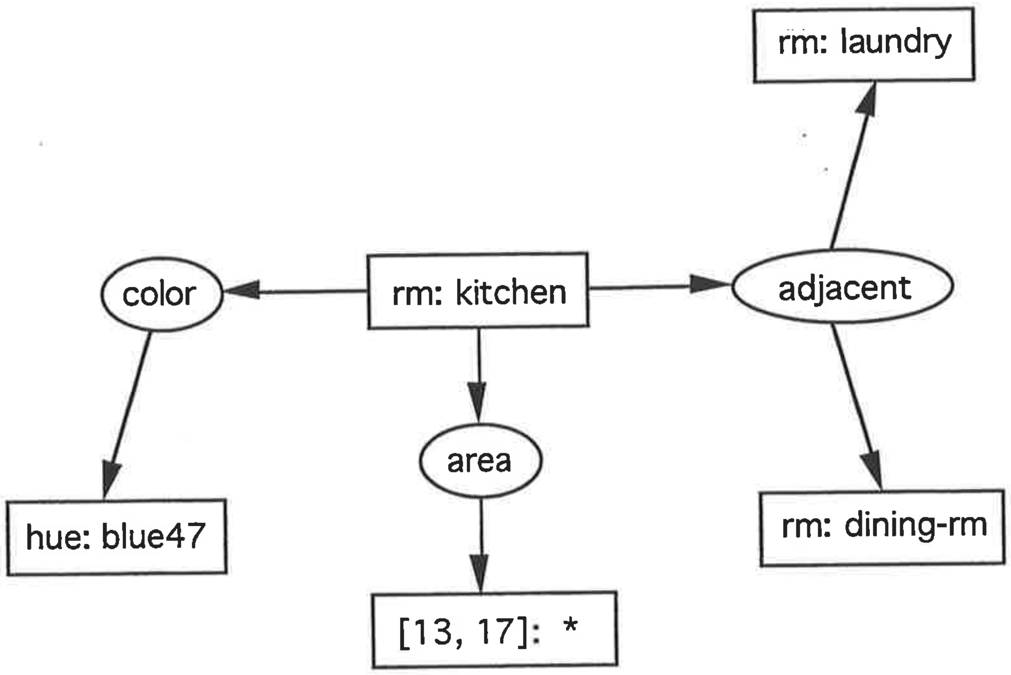


Figure 19. A Conceptual Graph representation of a kitchen SU.

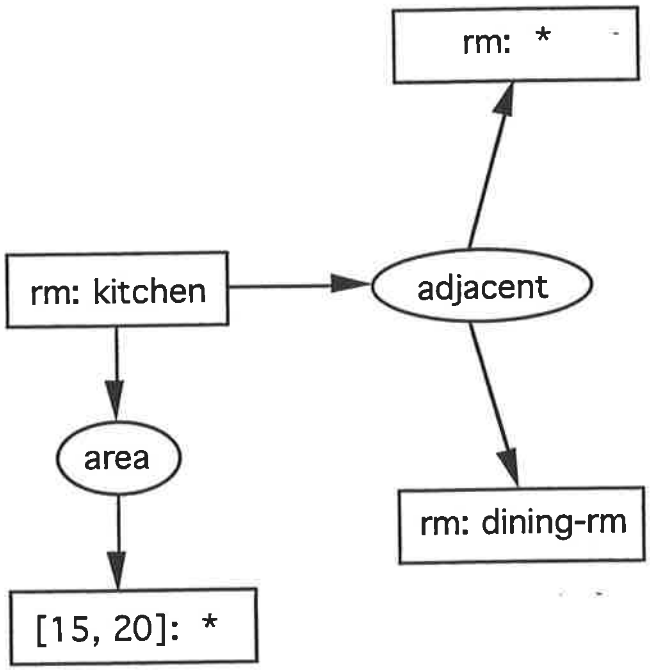


Figure 20. A conceptual graph of another kitchen SU.

4.3.3 Example of Unification with Constraints

We present here some of the examples which have been tested in the software which implements these ideas. We continue with the knowledge domain of the building architect. The architect will often try to reuse previous designs, not

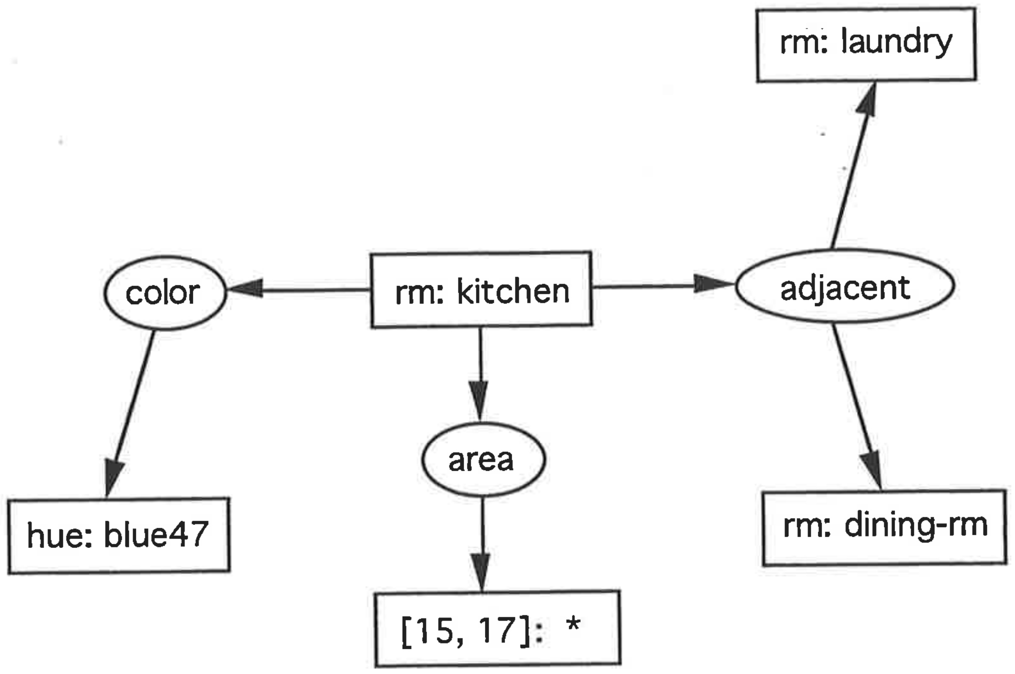


Figure 21. The unified design.

only to save time and resources, but also to identify solutions to problems previously encountered. Once a given problem has been solved, the design which represents the solution can be stored, and then retrieved when needed.

Consider in the domain of architectural design, a design for the kitchen of a custom-made house. In this design, the architect has specified some of the lighting design and that the floor area must be greater than 20 square meters. The architect has also retrieved an old design, which specifies the remainder of the lighting design. The graphs specifying the partial design and the retrieved design are shown in Figure 22. We assume that the portions of the graphs not shown in the diagram are compatible.

The unification algorithm defined above combines these two graphs, with the result shown in Figure 23. In this graph, all the original knowledge of the first two graphs has been preserved, and the values in the concepts have been joined as specified.

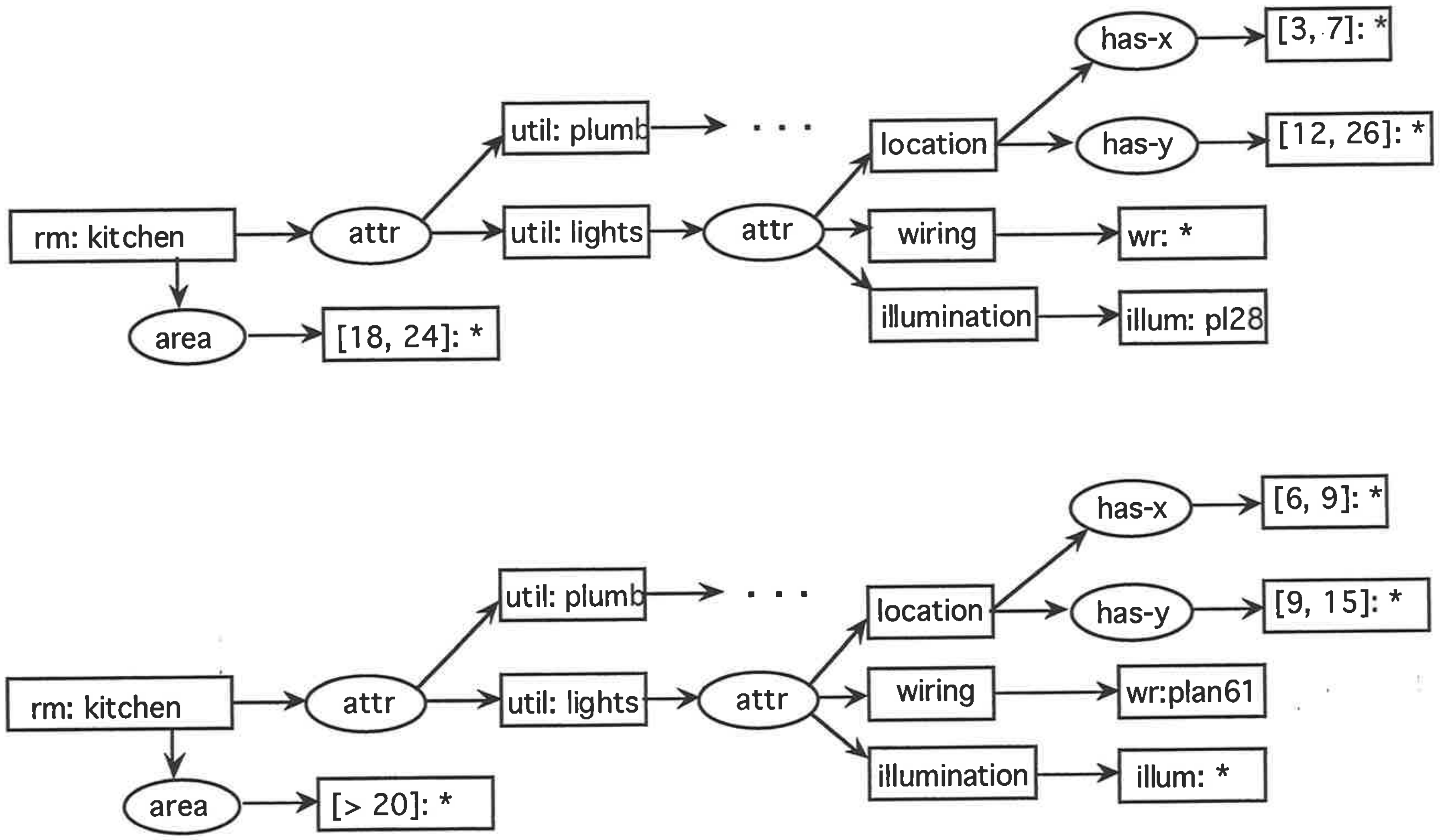


Figure 22. Two constrained partial designs to be unified.

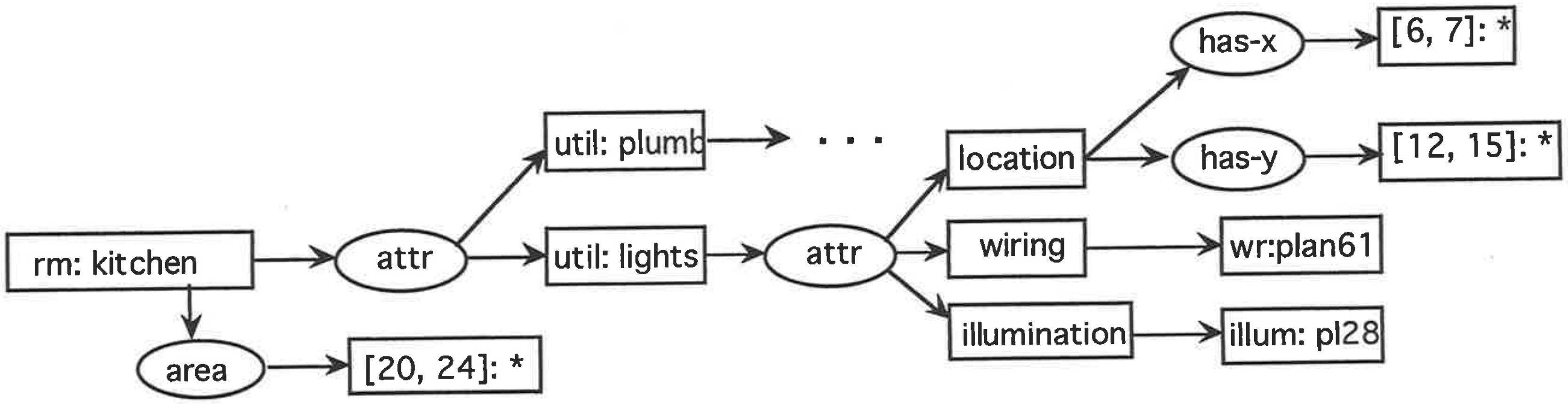


Figure 23. The unified design.

Another example of the utility of unifying partial designs is if the designer has a design similar to the second in Figure 22, specifying most of the lighting design. The second graph could represent a kitchen design where only the plumbing design is specified, but no lighting. These two would unify since the two heads are compatible, and the remainder of the graphs would be included in the unified graph. All of the knowledge is represented in the unified graph, which would specify the design for the lighting and the plumbing.

These examples also illustrate how the interval type allows real numbers to be represented in Conceptual Graphs. Any real number could be bounded inside an interval, similar to the concept of using floating point numbers to approximate real numbers in digital computers. Further, any concept containing a real value can be constrained with an interval. This allows the representation in Conceptual Graphs of constraint satisfaction problems. This use of interval constraints to represent real constraints has been used for some time in the Constraint Satisfaction Problem community. The work by van Hentenryck [Van Hentenryck et al. 1997; Van Hentenryck 1989] is a good example of intervals in CSP.

4.3.4 Graphs with Cycles

The Unify algorithm is based on a standard graph search algorithm which detects cycles in the graph. The algorithm proceeds in a depth-first manner until a leaf concept is encountered, or a concept is encountered which has been visited before. The depth-first traversal then continues from the last relation encountered, by following its next argument.

For example, in Figure 24 we see a generic house with three rooms, which the user wants to unify with the more specified design shown in Figure 25. (Recall that the double-headed arrows indicate a shorthand for a binary, reflexive pair of relations.) When these two graphs are unified, the Unify algorithm proceeds down along the *has-parts* relation to the first argument, in this case the *kitchen* concept in Figure 25.

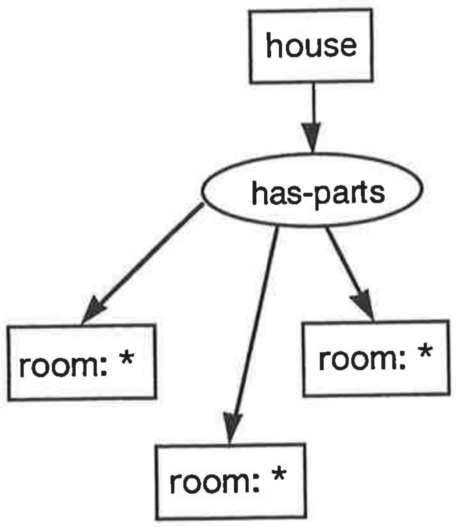


Figure 24. A generic house with three rooms.

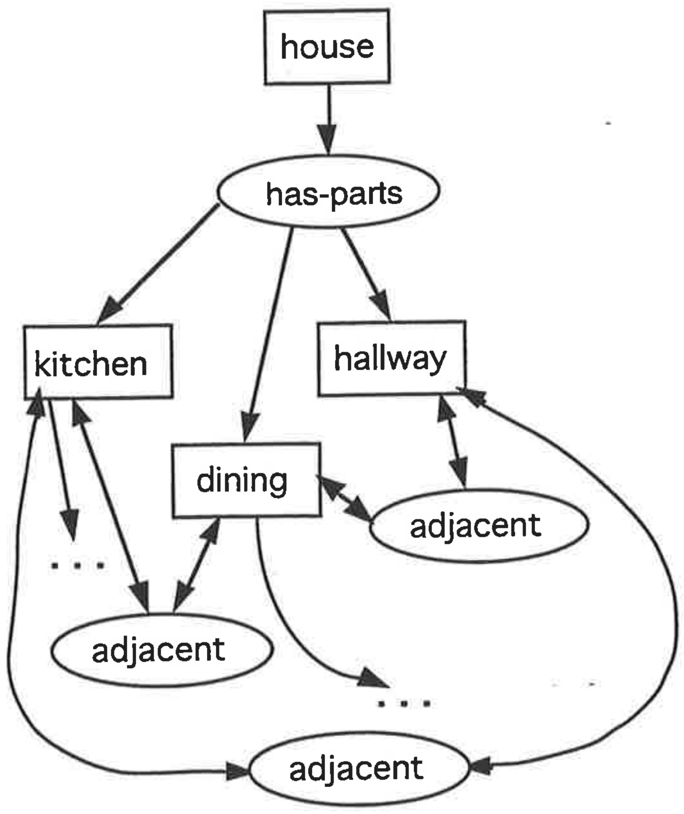


Figure 25. A house design with cycles.

The graph traversal will proceed by following the first argument in each relation, from *kitchen* to *hallway,* to *dining,* then back to *kitchen.* When the *kitchen* concept is encountered for the second time, the algorithm finds the concept marked "old" and stops the depth-first search. The corresponding *adjacent* relations are then traversed (ie from *kitchen* to *dining*, from *dining* to *hallway*, and from *hallway* to *kitchen*) but in each case the traversal stops immediately when encountering an "old" concept. Since the rooms all unify, the *adjacent* relations are all included in the new graph, and the graphs unify successfully.

4.3.5 Graphs with Alternate Heads

For unification of two Conceptual Graphs to succeed, the heads of each graph must be compatible, as defined previously. However, we have also discussed the fact that unification can be used as the technique to specialize and refine a generic graph, or a partially specified design. In this case, the user may want to start with a partially specified design for a house (for example) and extend it with a Conceptual Graph which only represents a bathroom. In this case, the user will want to specify the *rm: bathroom* concept as the head node in both graphs. In other words, the natural head of the graph (in this case *house*) may not be the concept under consideration by the user when specializing a graph.

In these cases, it is necessary not only to check that the sub-graphs under consideration are compatible, but also that the rest of the graph is consistent with the knowledge being added. The unification algorithm does this automatically, since it creates a set of all relations and checks every node in both graphs.

For example, it was mentioned in an earlier section that the graph shown in Figure 18 could be extended (and specialized) by unifying its kitchen concept node with the more complete kitchen Conceptual Graph shown in Figure 20. If the kitchen concept in Figure 18 is specified as the head node, and the corresponding kitchen node was specified as the head node in Figure 20, then the two graphs would unify. The resulting graph would consist of all of the graph shown in Figure 18, plus a unification of the kitchen elements in Figure 20.

The Unify algorithm starts from the head node, but always checks for relations that point to the $c \in C$ currently under consideration. If any are found, those relations are treated like any of the other relations in the graph, i.e. they make up part of the set M_G. Each arg_i is traced and attached in the appropriate place for the entire graph. In this way, a graph can be expanded and specialized by using unification to "attach" a small part to it.

By the same argument, a very deep branch in any graph under consideration will be followed to the leaf node, or until a cycle is detected. If a graph contains a very deep branch, it will be appended to the resulting unified graph. This is the same as a very detailed specialization of the graph. So, no matter how one of the graphs being unified is specialized (whether by a deep branch, or by relations from a larger graph), the specialization will still be included in the resulting graph.

4.4 Experiments: Difficult Structures and Unification Failures

This section of the thesis will describe the more difficult structures and experiments which were used to test the Unify algorithm. We discuss here special cases and problems with the algorithm, and cases where the basic nature of Conceptual Graphs leads the algorithm into unwanted (or counterintuitive) answers. At the close of Chapter Four, and in Chapter Six, we discuss some proposals for altering the definition unification and the nature of Conceptual Graphs to solve some of these problems.

4.4.1 Backtracking

Figure 26 shows two partial designs for a two-bedroom house. At first thought, it might appear that these two graphs should unify. In fact, the Unify algorithm produces a failure when presented with these two Conceptual Graphs as arguments. While there is an intuitive sense that these two graphs should unify, in fact there is no basis for unification under the standard rules of projection for Conceptual Graphs.

Recall the definition of Projection from Definition Four in Chapter One. The third condition of the definition states that projection must be structure preserving over the relations. That is to say that each arg_i in graph G must unify with its corresponding arg_i in G'. In the case presented in Fig. 26, in graph G, arg_2(*has-rooms*) is a bedroom which requires an area of at most 20 square meters. In graph

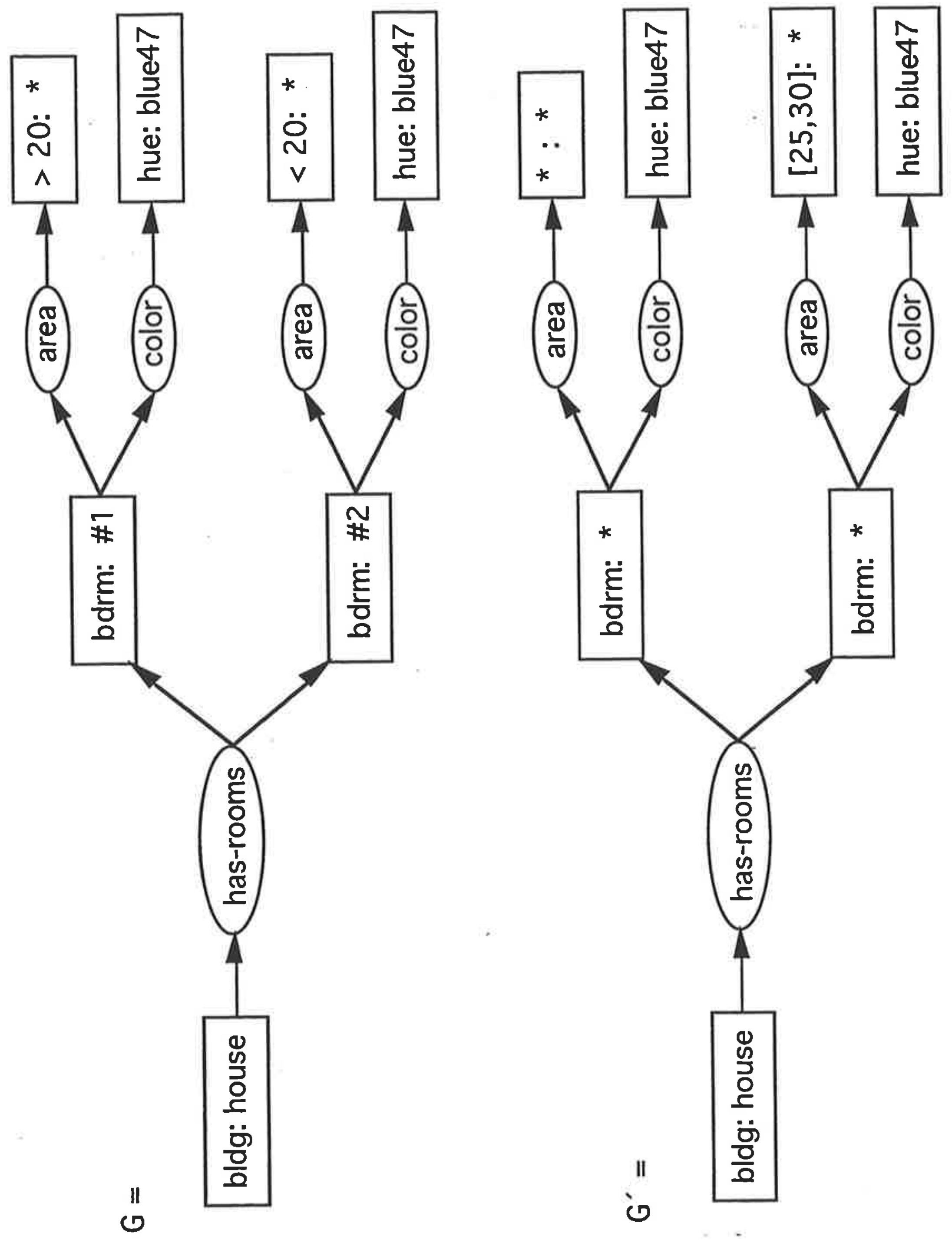


Figure 26. Graphs which fail to unify.

G´, arg'_2*(has-rooms)* indicates a bedroom which requires between 25 and 30 square meters. Since in this case, $arg_1(area) \sqcap arg'_1(area) = \bot$, there can be no unification.

From the point-of-view of the designer or knowledge engineer, this may not be desirable, even if correct according to the rules of Conceptual Graphs. The user might want to try all permutations, but that violates the concept behind structure-preserving functions. However, if we allow the user to permute the arguments to the *has-rooms* relation in Fig. 26, then we can obviously find a combination which will unify. This may be the case if, for example, the user specified each graph by using two binary *has-rooms* relations. If this method were used, Unify could fail on the backtraking issue. It is unclear from the CG literature, however, whether this would violate the semantics of structure preservation.

If a user wants the flexibility of permuting the arguments to the relations, then there are two solutions. The first is to build in a backtracking mechanism, which unbinds the first join, and then tries to bind other arguments, much in the way that Prolog processes unification arguments. There are many standard backtracking algorithms which will satisfy this first solution, and we will not pursue them here.

The second solution is to use an algorithm which restricts our structures to only headed graphs which have a very strict tree structure. We can then use Tree Isomorphism algorithms to find a match quickly and efficiently while avoiding the backtracking problem. This solution avoids backtracking problems by working from the leaf concepts, finding appropriate matches by permuting the arguments, and then proceeding up the graph to the head node. Restricting our structures to a subset of Conceptual Graphs can yield quite an efficient solution. The Tree Isomorphism solution to this problem will be discussed at the close of this chapter.

4.4.2 Constraint Satisfaction

The Constraint Satisfaction Processing literature contains many algorithms for solving such constraint problems as simultaneous equations and multiple variables. While the Unify algorithm was designed to resolve constraints expressed as intervals in a Conceptual Graph, it does not have the expressive power that true CSP systems have. Indeed, where CSP can manage large amounts of arithmetic constraints very efficiently, Conceptual Graphs, and the Unify algorithm in particular, are designed to manage large amounts of domain knowledge efficiently. The best approach for a complete constraint system would

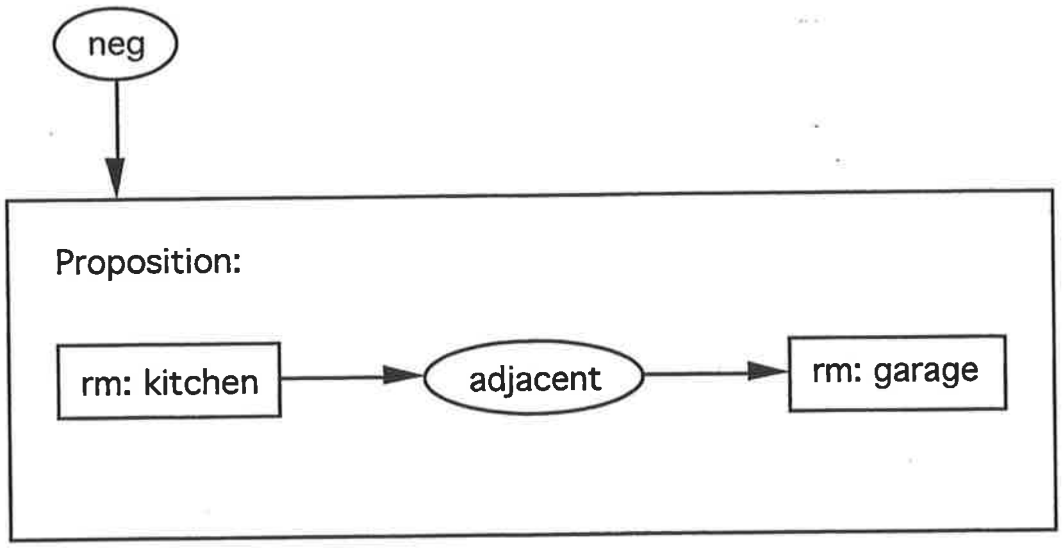


Figure 27. A Conceptual Graph with a negative context.

be to combine these two types of knowledge representation into a single constraint handling system.

The idea of a single toolbox for knowledge combination is discussed in Chapter Five, while comparisons between CSP and the Unify algorithm are covered in section 4.5.

4.4.3 Unary Relations

Unary relations are something of an exception in Conceptual Graph theory, and indeed in this thesis as well. Sowa [Sowa 1999; Sowa 1984] discusses the *negation* unary relation, but applies it only to *propositions* of graphs. In the context of our domain of discussion, this would be used, for instance, to assert a proposition that a kitchen is not adjacent to the garage, as shown in Figure 27.

Contrary to what a user might expect, if we were to attempt to use the Unify algorithm to unify this graph with a similar graph which did not have the negation relation, the unification would succeed, and the unary relation would appear in the final result. That is to say that if we unify some Proposition A with the negation of Proposition A, the resultant (incorrect) unified graph would be the negation of Proposition A. This is because all relations which are compatible in type and arity, and which have compatible arguments are always included in the resulting unified graph. If there is some relation R is one graph, but no corresponding relation in the other graph, then relation R is carried through into the result, since it is considered to be not incompatible with any relation in the other graph. The negation relation would fall into this category.

The deeper problem for the negation relation is obvious. Since the purpose of that relation is to assert the negation of the meaning of some graph, we need to be able to express the idea that it must not unify with its opposite. The Unify algorithm does not correctly handle this situation, as a deep understanding of the semantics of the graph is required in order to do this. The unify algorithm, while correctly resolving constraints and type hierarchies, nevertheless has no real understanding of the semantics of a graph.

4.4.4 Canonicity and Validity

One major drawback with all unification techniques is that the user can guarantee canonicity, but there is still no way to guarantee validity. The validity of a Conceptual Graph has more to do with Truth Maintenance Systems than it does with the structural correctness of the graph. The Unify algorithm presented in this thesis is no exception. While we guarantee that two canonical graphs that are unified under the algorithm presented in Chapter Three will produce a canonical graph, there is currently no method for validating the meaning of the graph.

Take, for example, the two graphs shown in Figure 28 and Figure 29. Fig. 28 shows a Conceptual Graph which asserts that there exist in the world some animals which are a particular shade of blue. Figure 29 says that our cat Felix has the attribute of color. As shown in Figure 30, the unification of these two graphs says that Felix is colored blue. While we may not know Felix's current state, and whether he has recently fallen into a bucket of paint, it is generally accepted that cats do not come in the color blue. It would be easy to say that Unify has derived an invalid graph.

In fact, this problem is more in the arena of the knowledge domain designer and the truth maintenance researcher. One way to assert that we will not allow blue cats to exist in our system is to arrange the concept type hierarchy for colors to disallow certain colors for certain animals (or other objects). Figure 31 shows such an arrangement for the color type hierarchy. While many hues may apply to the objects under consideration in our domain, we can use this hierarchy to exclude colors such as blue and green from being used with cats. Once this hierarchy is in place in our system, we can assert the canonical formation rule shown in Figure 32. When specialized, this canonical formation rule would not allow Felix to have the color blue as an attribute, and the graph in Fig. 29 would have to be modified to comply with the rule. The unification of the canonical formation rule shown in Fig. 32 with the graph in Fig. 28 would fail, since blue is not a *cat-color*.

There's a sense that this problem is bound up with the idea of the user properly defining the domain, and setting up the type hierarchies. Yet, we cannot guarantee that the user will not derive "Felix is orange" when he's really black. This is the domain of truth maintenance and theorem resolution, which are not dealt with in this thesis. In other words, the user can assert, "there are no blue cats," but they cannot say whether it is really true that Felix is black!

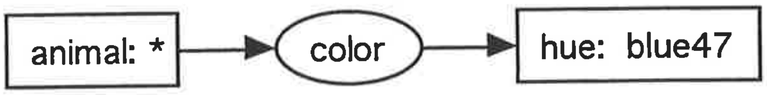


Figure 28. There are animals which are blue.

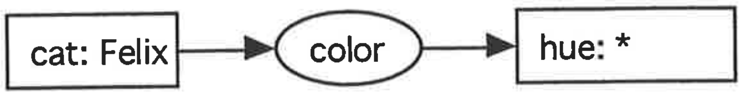


Figure 29. Felix has some color.

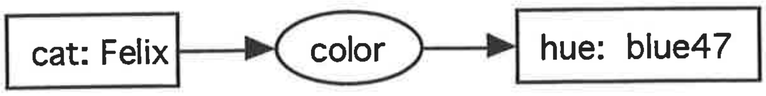


Figure 30. Felix is blue.

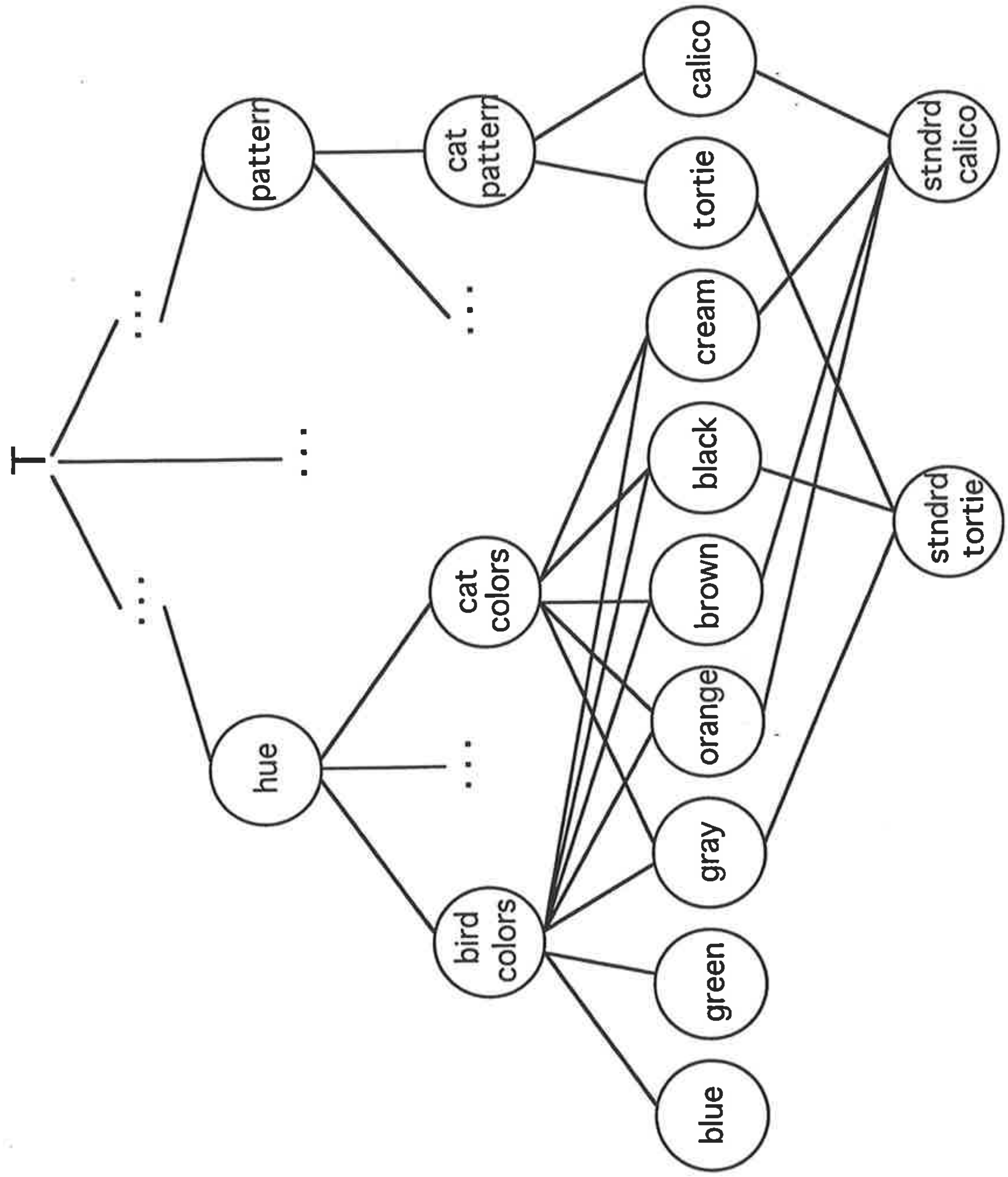


Figure 31. A concept type hierarchy for cat colors.

$$\begin{tikzcd} \boxed{\text{cat: *}} \arrow[r] & (\text{color}) \arrow[r] & \boxed{\text{cat-color: *}} \end{tikzcd}$$

Figure 32. A canonical formation rule restricting the color of cats.

However, there's also a sense that designing a house is a continual refining and specifying of the design, as was mentioned previously. In this sense, at every step the user will produce a *valid house design* (minus specializations). While this is not the same as guaranteeing the truth of a graph, we can guarantee that once the domain hierarchies and canonical formation rules have been properly defined, the only structures that will be allowed will be those that represent reasonable knowledge in the domain, and in this sense are valid.

4.4.5 Conclusions Regarding the SEED Project

In discussions with the architect members of the SEED Project team, several issues of unification, constraints and matching were identified. The three main areas where the SEED Project needs the contribution of Conceptual Graph unification are in type subsumption, knowledge-level reasoning, and pattern matching. Each of these three areas is discussed below, along with a qualitative judgment of how well Conceptual Graphs and the Unify algorithm deal with these concerns.

The SEED architects want to be able to use type subsumption to make statements such as, "An office (or kitchen, or corridor) is a kind of room. All the properties which apply to one should apply to its specializations." This is distinct from the object-oriented objective of objects inheriting all the properties of a class of objects. The essential difference is in treating a kitchen as you would any generic room. A generic room can be placed, occupy space, and have attributes like color and number of doors. A *class* of rooms will have attributes, but cannot be said to occupy a space or have specific dimensions, or have a specific count or placement of doors. The generic room can have constraints placed on its attributes, and finally can be specialized into a kitchen.

Fundamentally, a generic room can take the place of a specialized room, unlike a class of objects. The room can stay generic for as long as the user needs it to be generic, and then specialized. Further, the room could be specialized wholly or in part. If partly specified, it can be matched against other specifications to find appropriate matches.

Conceptual Graphs and the Unify algorithm give this ability to the architects. The Unify algorithm allows the user to specialize designs by matching (unifying) previous designs with the current design problem. Since all characteristics, attributes and constraints are carried along in the unification, the specialization represents all of the design concepts included in the more generic design. Further, and more importantly, there is no real separation between generic and specific, since all points in between can be represented. Conceptual Graphs combined with the ability to specialize using unification are the ideal tool for the knowledge combination approach and the constructive nature of architectural design.

The second major concern of the SEED Project designers was the ability to have knowledge-level reasoning. That is, they want to be able to speak in the language of the architect, not the language of the computer (or CAD system). The user wants to be able to refer to the "North Wall" or "door" without resorting to discussing geometric coordinates in space. The user wants to depart from previous CAD-based data-level processing, and work at the knowledge level in the architecture domain.

This is certainly another area where Conceptual Graphs and unification combine to bring a solution to this domain. While spatial coordinates (and their constraints) can be stored in a graphical representation of a room, there is no need for the user to bother with using them. The graph can be manipulated as a whole, and treated as a room, rather than a square in a diagram. The completed SEED system will not deal with lines and boxes, but rather with specializing entire designs for rooms (or houses, or office buildings). This approach frees the architect from dealing with data-level concerns of numbers and coordinates, and allows the architect instead to deal with the architectural design.

The third major concern of the SEED team is in the area of pattern matching. The users want to be able to start with a high-level, generic description of a building, perhaps represented as a hierarchy of Functional Units. Then they want to be able to make queries such as, "Can this bay structure be used in the support structure?" or, "Do the constraints match up adequately for a particular technology to be used? If yes, tell me the constraints under which it is usable."

Once again, the work presented in this thesis meets the requirements of the SEED Project team. A query can be represented as a Conceptual Graph. The user can specify a type of structure for support, and make the query by attempting to unify the structure with the more generic design. If the unification fails, then the user knows that the proposed structure does not meet the constraints of the design problem. If the graphs unify, then the resulting graph will contain the constraints which must be met in order to make the design work.

Overall, the system of unification over constraints on Conceptual Graphs presented in this thesis gives a set of tools to the designer. The ability to use knowledge combination with constraints to handle objects at the knowledge level greatly leverages the ability of the designer to work efficiently.

4.5 Comparisons to Other Systems

4.5.1 Implementations

Up to this point, the discussions have centered on the implementation of the theories discussed in Chapter Three, and how that implementation performs in a particular design domain. This section changes the direction of this chapter by comparing the implementation of the Unify algorithm to the unification and constraint systems implemented by others. As discussed in some detail in Chapter One, other models of CG unification or constraint resolution offered in the literature are only partially defined. A formal comparison, in terms of validity, complexity and completeness is therefore not possible. We can, however, compare methodologies by discussing how each system would be used to solve the problems in our example domain. The aim of this section, then, is to demonstrate that the Unify algorithm solves problems that others have failed to solve, or that they solve in inefficient ways.

4.5.2 Structural Unification

Recall from Chapter One that Willems' approach to unification is to create a "unifier" for two Conceptual Graphs which, in his definition, is the common generalization of similar segments in the two graphs [Willems 1995]. The unifier is therefore a graph in which every concept can subsume its corresponding concept in both graphs being unified, and every relation can subsume its corresponding relation.

His method starts by finding a subgraph which is similar in both graphs, and then finding the meet of the two subgraphs to form the unifier. The idea is then to match the two graphs together along the unifier, and then just attach all other nodes (ie the nodes that are not in common). Willems refers to the process as "gluing together" the two graphs, but he does not present an algorithm for accomplishing this gluing process.

Willems' process of unification is illustrated in Figure 6, which is reproduced here from Chapter One. The two graphs G and G' are to be unified. The unifier U is found by taking the meet of G' and the segment of G which corresponds to "a man with a name, which is some word." The unifier is then "a person with a name, which is some word," shown as graph U in Fig. 6. Projections for $U \rightarrow G \rightarrow G''$ and $U \rightarrow G' \rightarrow G''$ are created to find the unification G''. The first major problem with this approach is that there is clearly more than one possible projection. Why is 'Smith' the girl's name, and not the man's? This ambiguity is a small example of the possible loss of information that can occur in Willems' algorithm.

The other major problem with Willems' approach is that Willems does not actually describe how to find the projection into G''. He only describes an algorithm for finding the unifier U (essentially a graph isomorphism approach). Also recall from Chapter One that it is possible for the unified graph G'' to be undefined in Willems' algorithm.

We now examine the advantages and disadvantages of the algorithm defined in this thesis compared to Willems' algorithm, using the SEED domain of building

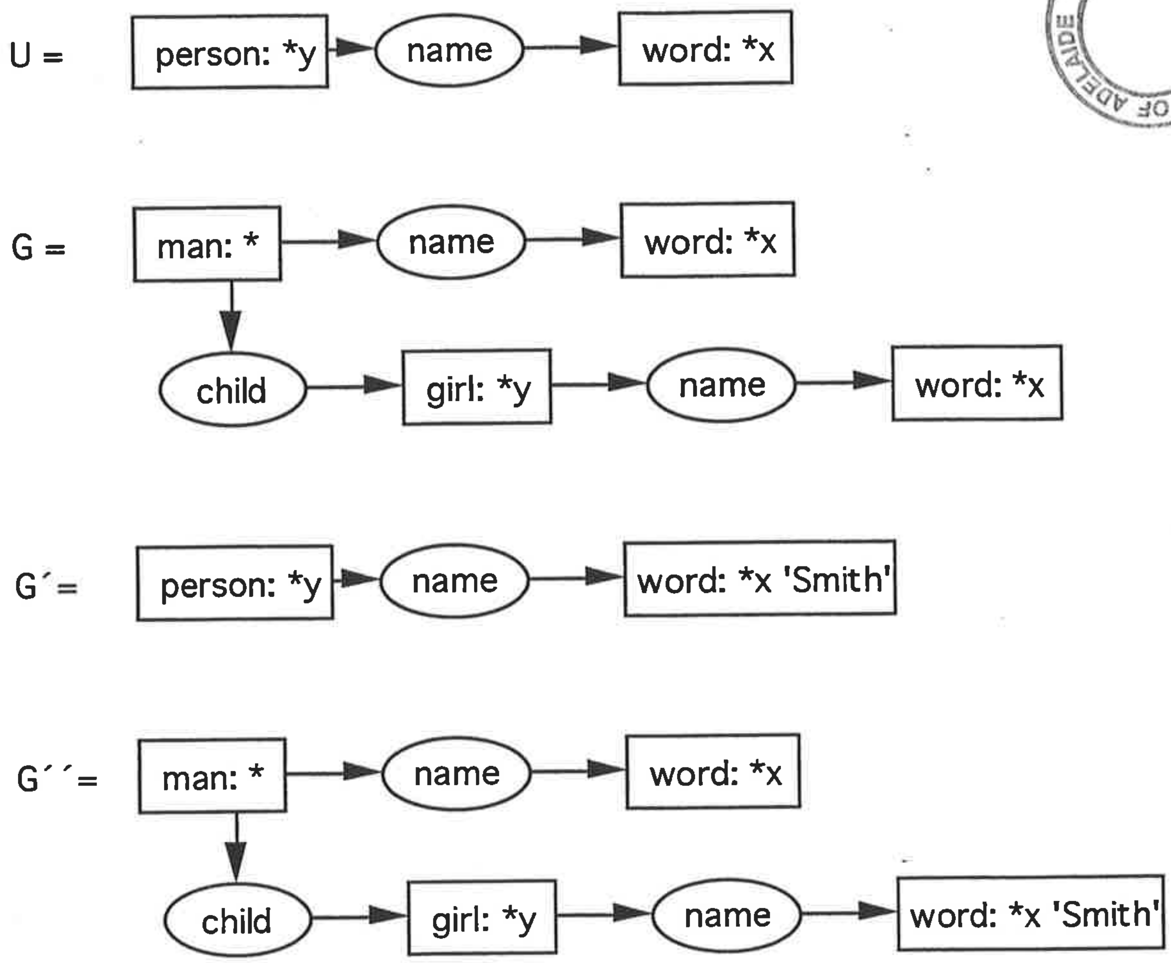


Figure 6. Unification in the style of Willems.

architecture. Figure 33 represents the unifier that would be produced by Willems' algorithm when unifying the two partial house designs shown in Fig. 22. In this case, the unifier is just the meet of each of the nodes from the two graphs from Fig. 22. We use T as a representation of a concept which is more generic than the intervals in the original graphs. The wiring and plumbing plans which were in the originals have now been replaced by generic plans. If we accept Willems' argument that we can then use projection through the original two graphs, we should have a result something like Fig. 23, which was produced by the Unify algorithm in this thesis. However, since Willems does not specify how to perform these projections, it is unclear exactly what the result would be, or how the result would be derived.

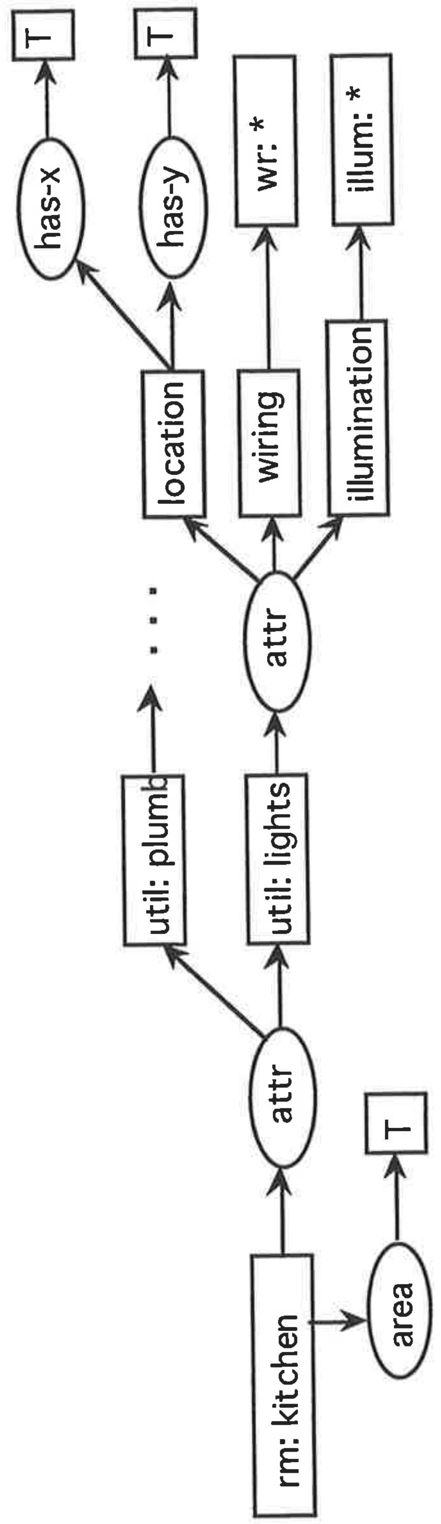


Figure 33. Unifier of two partial designs.

Even though this result is theoretically the same, this step of finding the unifier which is more general than the graphs being unified is an extra step that is not used by the Unify algorithm. The Unify algorithm does not use the idea of

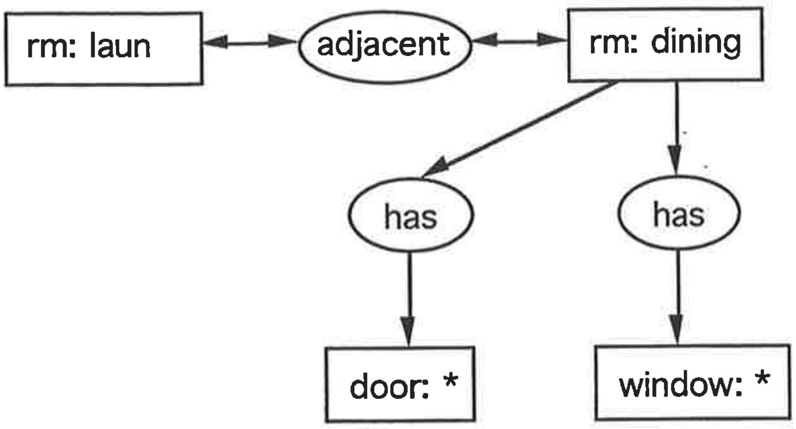


Figure 34. Graph *G* to be unified.

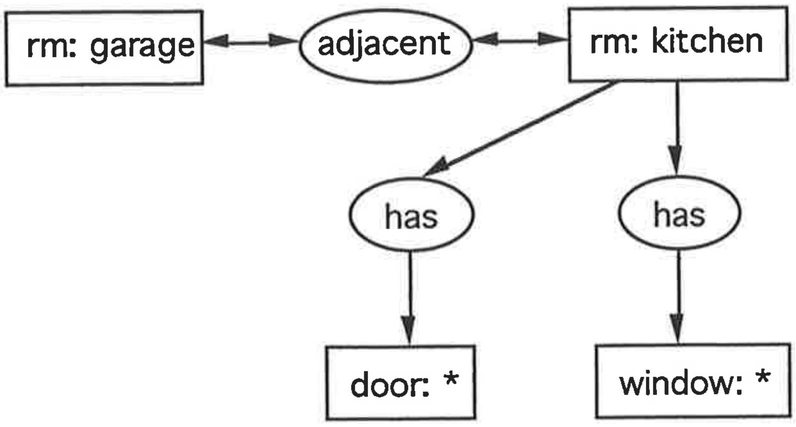


Figure 35. Graph *G´* to be unified.

finding compatible projections on a common unifier, but rather uses the definitions of projection and join to find a unique most general common specialization. This would supposedly be the next step in Willems' algorithm, but he leaves this step unspecified.

There is at least one area where Willems' algorithm has an advantage over the Unify algorithm. Figures 34 and 35 show two graphs, *G* and *G´*, which are partial house designs to be unified. Using Willems' algorithm, the common segments found in these two graphs are the two rooms, and the relations to the doors and windows. Therefore, the unifier would be a description of two adjacent generic rooms, plus the window and door of one of the rooms, as shown in Fig. 36.

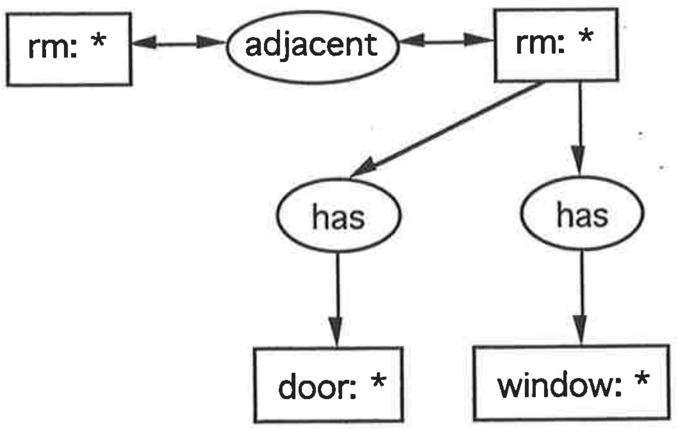


Figure 36. U, the unifier of *G* and *G´*, by Willems' algorithm.

In the Unify algorithm presented in this thesis, it would also be difficult to unify these two graphs, because there is no obvious head node. If we specify, for example, that *dining* is the head node of G, and *kitchen* is the head node of G´, the graphs still do not unify due to incompatible head nodes. In this case, the Willems algorithm has some advantage. Under Willems, there is at least some basis for comparison of the two graphs. The Willems unifier produced would at least indicate that there is common ground and a common domain between the two graphs. The Unify algorithm would, in essence, fail before it even starts.

However, the ability to produce a more general unifier may not always be an advantage. First, there is the cost associated with finding the common generalization, which is at least as costly as finding the common specialization. Then there is the problem of finding compatible projections down through each of G and G´ to the new unified graph. Willems does not specify how to use (or even find) the compatible projections, except that the projections $U \rightarrow G \rightarrow G''$ and $U \rightarrow G' \rightarrow G''$ must be compatible and produce a common graph, G'' [Willems 1995]. In our case, it is unclear what G'' would be. There is a projection $U \rightarrow G$ which includes the *dining* room and the *laundry*, but there is no way to make that compatible with $U \rightarrow G'$ where the rooms must be *kitchen* and *garage*.

Recall that Müller demonstrated that a join on compatible projections is not the same as finding a most general common specialization (or *greatest lower bound*)

[Müller 1997]. In this example it is not possible to find any common specialization, and therefore there can be no most general common specialization.

The lack of an ability to produce the final unified graph is a concern, but an additional concern would be that even a common generalization might not be useful. Returning to the common generalization shown in Fig. 33, since the intervals and the plumbing and wiring plans have been made generic, the detail from the original plans is lost. Producing such a common generalization may show that the two graphs being unified have some common ground, but this graph may be so lacking in detail as to be useless. Further, it is the constructive nature of design that is of the most interest to the SEED project. The continual refining and specifying cannot be represented using a Willems-like unification algorithm which relies on generalization to find common elements. Rather, a method for combining graphs into a representation of the knowledge from both of the graphs, without losing any of the knowledge through generalization, is what is needed in this domain.

While both algorithms would have difficulty with the problem just presented, the Unify algorithm can still work in certain cases. We can modify the graphs slightly, so that in graph *G´ kitchen* is not only adjacent to the *garage*, but also adjacent to *dining*, as shown in Fig. 37. Then we can specify *dining* to be the head node in both graphs. The head nodes are now compatible, and the graphs unify to produce the graph shown in Fig. 38.

The point is that there is intuitively a combined graph which contains all the information from both *G* and *G´*. This is the graph which represents the construction of a new design from old knowledge. Figure 38 shows the result of using the Unify algorithm from this thesis, with the headed versions of G and G´ as the input graphs to be unified.

Where the Unify algorithm can now find a useful common specialization , in the case of Willems' algorithm it is still not clear how a the common specialization could be formed. The common generalization would still be as shown in Fig. 36,

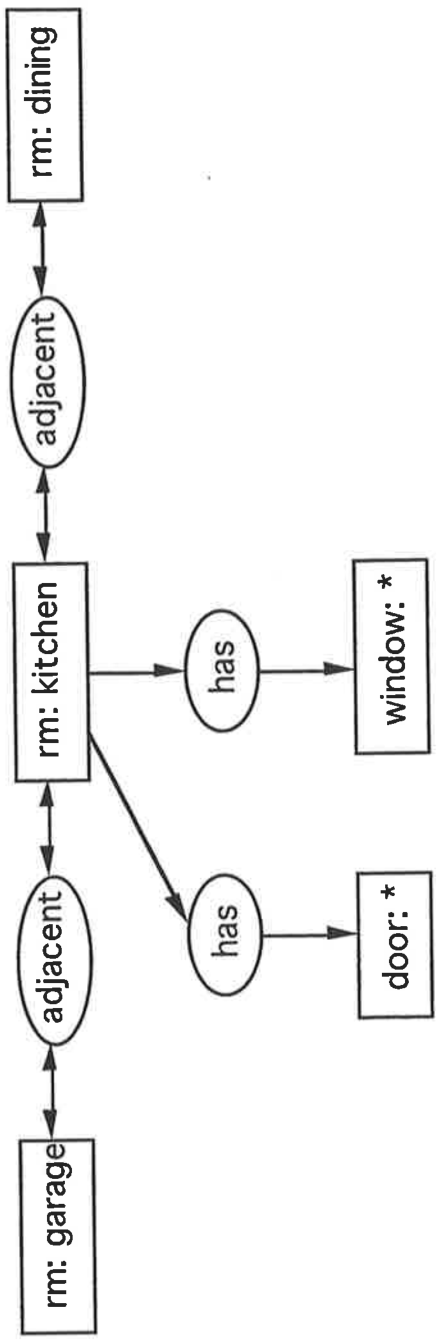


Figure 37. Graph *G´* with a head node.

and in comparison with Fig. 38, the result of the Unify algorithm, is lacking in detail and not very useful.

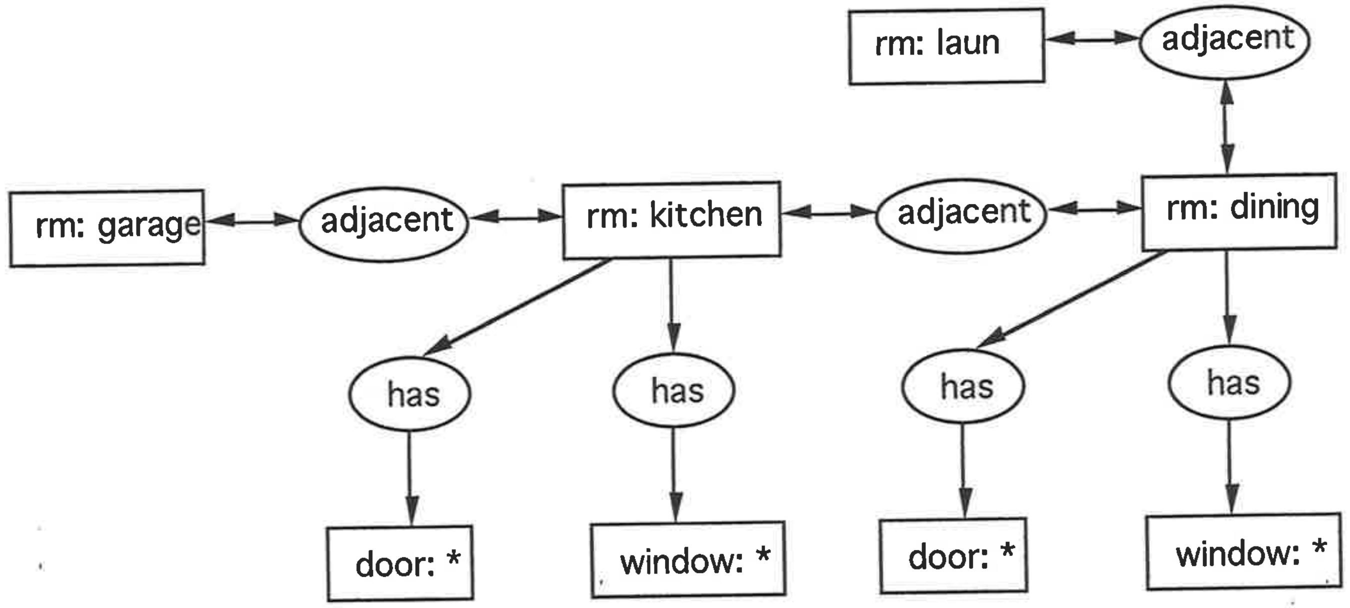


Figure 38. The unification of Fig. 34 and Fig. 37.

4.5.3 Structural Constraints

Recall from Chapter One that most methods for constraining Conceptual Graphs deal with constraining the structure of the graph. One exception that was discussed was the negative canonical models of Kocura [Kocura 1996]. Kocura's method defined Canonical Formation Rules, and then defined exceptions by specifying certain specializations of the rules which were not allowed. If any of these negative models have a projection into a graph, then that graph is outside of the allowed rules, and therefore not canonical.

Kocura's approach is another which guarantees canonicity, as does the Unify algorithm defined in this thesis. Kocura's algorithm refines the canon of acceptable graphs in the domain. When two graphs are unified, Kocura's algorithm checks that the result is still canonical, by checking that the resulting graph is not a specialization of a negative model. By this method, Kocura constrains the canon, but not the values of the individual referents.

An example of this type of constraint is the Blue Cat problem, discussed in this chapter, in section 4.4.4. If all colors could be used in describing cats, Kocura's solution would be to create a negative canonical model, which would disallow the colors blue and green being used as a cat color. Specializations of this negative model would also be disallowed. The solution in this thesis was just to rearrange the type hierarchy by adding an additional layer, and then changing the rules to specify that cats could only have cat-colors.

As another example from the SEED domain, take the graphs shown in Figure 39. Here we see parts of the type hierarchy describing types of flats and types of rooms. The positive canonical model, labeled CM+, says that flats have rooms associated with them. The one negative canonical model, CM-, makes the exception that studio flats do not have bedrooms. Any specialization of CM- will not be allowed under Kocura's scheme.

The method for handling this situation under the scheme described in this thesis is shown in Figure 40. Essentially, what has been done is simply to

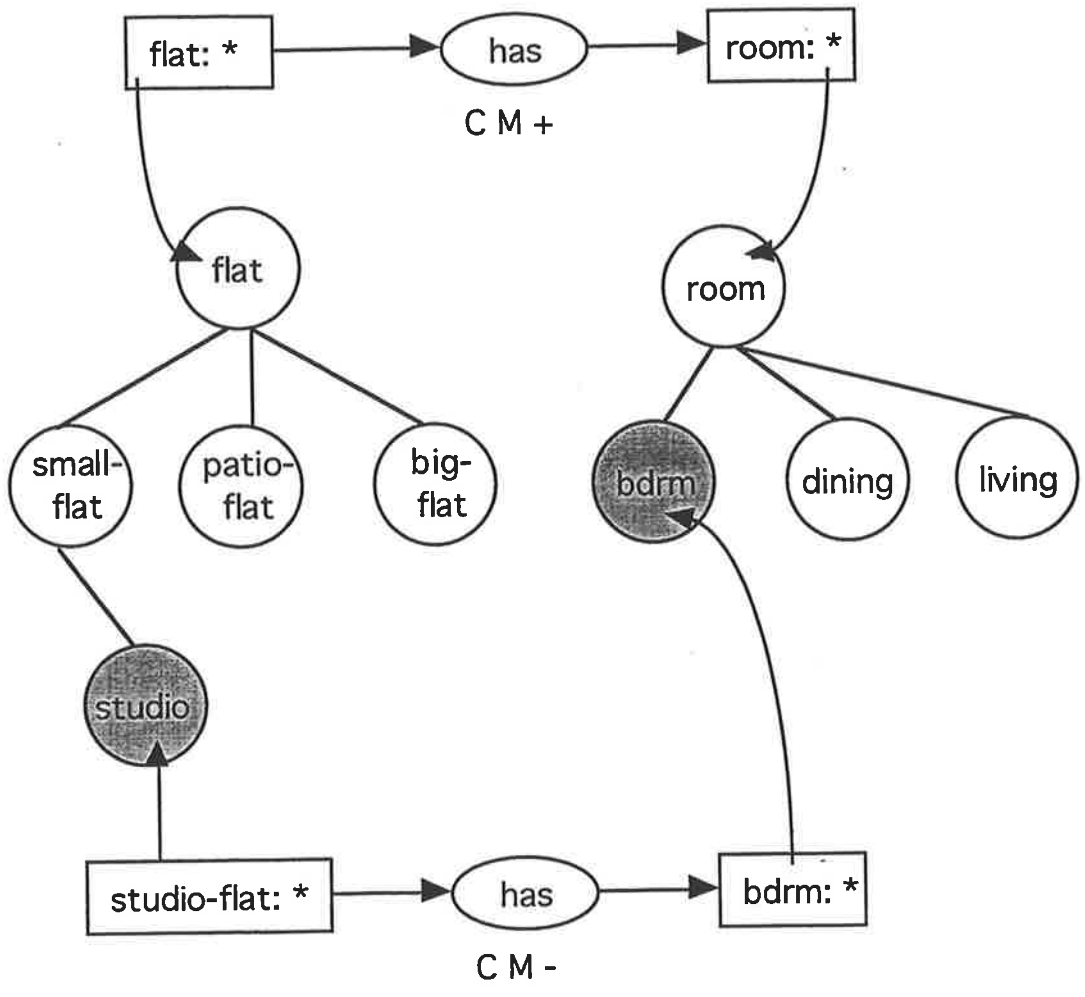


Figure 39. Kocura's negative canonical model in the SEED domain.

rearrange the type hierarchies to better represent the knowledge that is needed in the domain. Then the canonical formation rules, shown in the bottom of the figure, accurately represent the types of structures needed with each type of flat. The studio type of flat is no longer an exception to the rules, but instead has a rule which requires an open plan for the studio. Now, the Unify algorithm can be used to specialize any of these rules into a more specified design for a flat.

Kocura's algorithm and the Unify algorithm both guarantee to produce canonical graphs. Both approaches constrain the graphs that can be unified or specialized. The essential difference in the two algorithms is that Kocura still requires an additional step to produce the results. Whenever a canonical formation rule is specialized, Kocura's algorithm must perform an additional

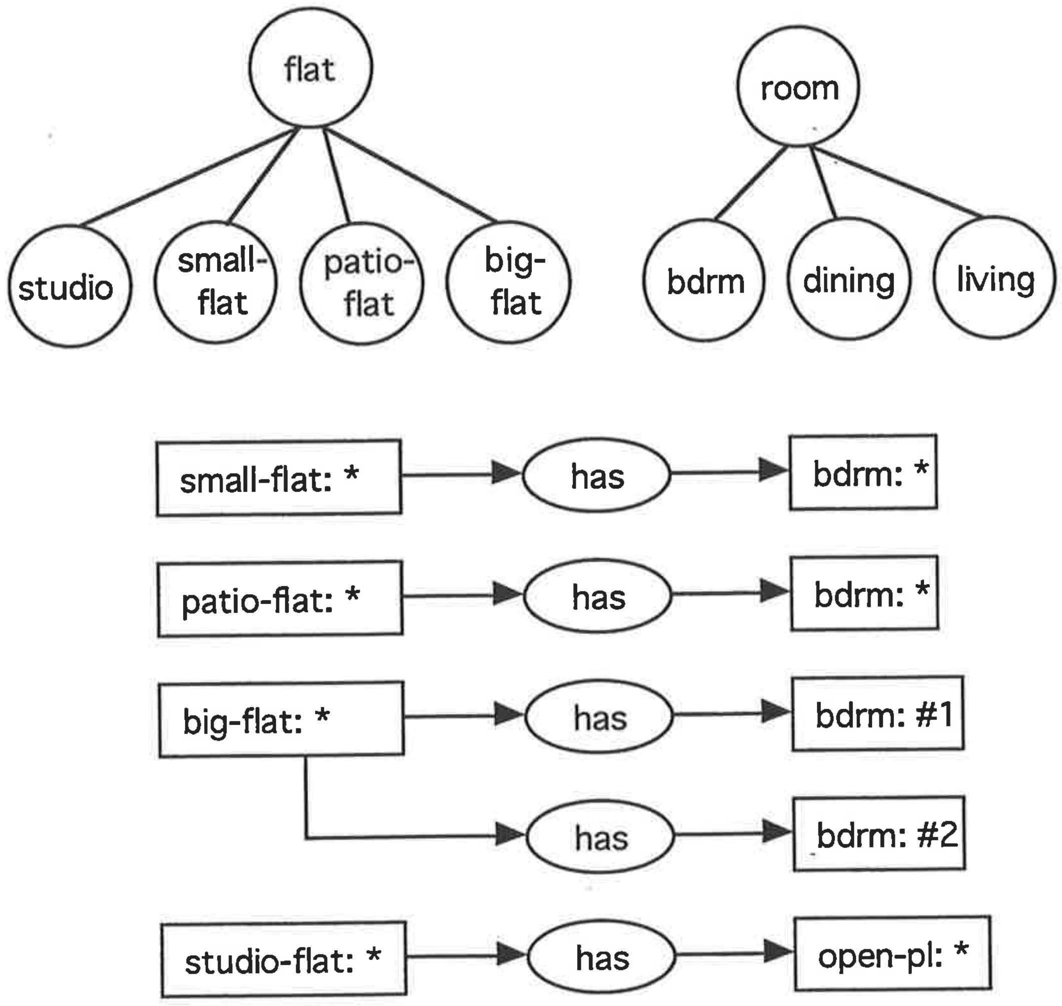


Figure 40. Another method for representing the bedroom constraint.

check to see whether any of the negative models have projections into the newly specialized graph. If there is a projection, then the graph is invalid, and the unification must be undone.

The Unify algorithm does not have any problem in this case. The rules as specified in Fig. 40 spell out that the studio flat must have an open plan design. The open plan design can be further specified in the type hierarchy to have several functions in one large room. Since studio-flat is not a specialization of any of the other types of flat, there is no danger that a separate bedroom will be specified for the studio. Further, the constraint processing is built into the algorithm, where Kocura's algorithm is an added process performed at the end of the unification.

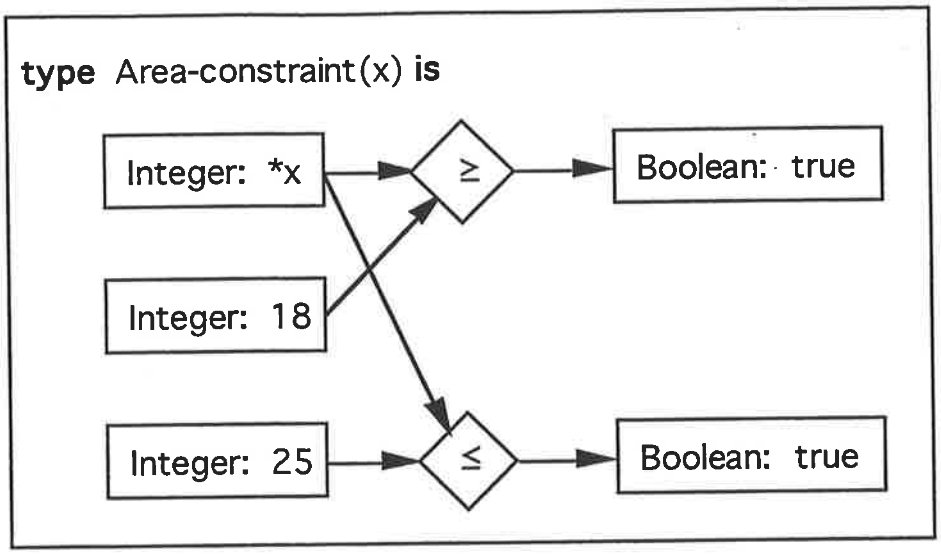


Figure 41. An area constraint expressed in Mineau's actors.

Once again, the more important issue is that the Unify algorithm looks at constraints on the value of concepts. We demonstrate that the situation encountered in Fig. 39 can be reconfigured just by using the type hierarchies to rearrange the domain knowledge, but this type of structural constraint is not the point of the Unify algorithm. In this domain, specific values of the concepts are handled in a manner that represents the domain knowledge accurately.

4.5.4 Constraints with Actors

Recall from the first chapter Mineau's work on using actors to represent domain constraints. Mineau specifies domain constraints by defining procedural attachments to graphs which are activated when their concepts are instantiated [Mineau and Missaoui 1997]. These procedural attachments (or actors) should always have a projection into the graphs where they are used. If a projection does not exist, then the graph violates the constraints, and is invalid.

Each actor constraint is defined as a type. Figure 41 shows the definition of the type *area-constraint* which constrains the area of a room to be between 18 and 25. Presumably, one could define variables for the two input values, making the type definition more flexible. A concept of this type could then be used with an area relation to constrain the area values. The ≥ and ≤ actors would be invoked

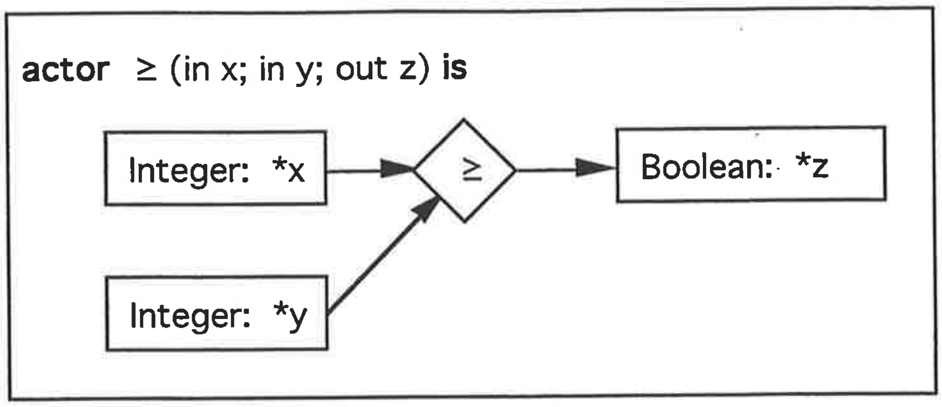


Figure 42. Mineau's ≥ actor, from [Mineau and Missaoui 1997].

each time a graph is joined to a concept of the *area-constraint* type. If the constraints are not met, then the join fails. The ≥ actor is shown in Figure 42, which is taken directly from [Mineau and Missaoui 1997].

Mineau suggests a method for a shorthand representation for these constraint actors. The constraint on the area of a room shown in Figure 41 would be expressed in Mineau's shorthand as: [INTEGER: *x ∈ [18, 25]]. Mineau goes to great lengths to explain that this notation stands only as a contraction for the more explicit version of the type definition. The shorthand is similar in appearance to the constrained concepts introduced by this thesis.

Note that the two main differences, however, are that the interval type defined in this thesis is the actual type used in the concepts, and that Mineau does not define a type for constraints on real numbers. The interval type defined in Chapter Three can be used to define concepts, and can then be unified with graphs to validate the constraints represented by the intervals. This definition of interval type, along with the unification algorithm presented in Chapter Three, rely only on projection and join, while Mineau requires an additional step to call actors to check the graphs during the attempted join.

4.5.6 Knowledge Conjunction Using Tree Isomorphism

As was mentioned in Chapter Two, the standard graph technique of tree isomorphism can be used as an efficient implementation of unification of CGs, if the further restriction of only allowing CGs which are finite trees is allowed. The definition of tree here is quite strict: The graph must have a root (or head) node, and be finite, directed, and acyclic with no node having multiple parents. Under the following definition of Tree Conceptual Graph, all relations must be binary.

As with standard tree isomorphism algorithms, we also allow relation arguments to be permuted. This departs from standard CG techniques, but allows a greater flexibility in dealing with knowledge conjunction. The example given here demonstrates that allowing the permutation of relation arguments can sometimes aid in the appropriate combination of the domain knowledge represented in the trees.

Definition 11. Tree Conceptual Graph. A tree conceptual graph with respect to a canon is a tuple $G = (C, q\ , R_2, type, referent, arg_1, \ldots, arg_m)$ where

q is the head or root node, as defined previously

$R_2 \in R$ is the set of conceptual relations, restricted to binary relations

In a Tree Conceptual Graph, since all relations are binary relations, there will necessarily be concept nodes at the leaves, as well as at the root. The relations which point to the leaves will be referred to as "Leaf Relation Nodes". This algorithm has been adapted for CGs from [Aho et al. 1974].

Algorithm 2. Tree Isomorphism Unification.

To unify two tree conceptual graphs, G and G', start with the leaf nodes of each CG. Then:

- Construct the set of all concepts at the current level i, from each CG, L_G and $L_{G'}$, preserving the relation structure.
- Sort each set by type, and within each type from most specific to most general. (This sorts the sets from most constraining value to least constraining.)
- For each $c \in L_G$:

- Find a corresponding $c' \in L_{G'}$ where $c \sqcap c'. \neq \perp$, by scanning $L_{G'}$ from most constraining to least constraining. If none is found, unify fails.
- Set $c'' = c \sqcap c'$.
- For the $r \in R$, and $r' \in R'$, where $arg_1(r) = c$ and $arg'_1(r') = c'$, join r and r'. If $r \vee r' = \perp$ then unify fails.
- Set $r'' = r \sqcap r'$.
- Set $arg''_0(r'') = arg_0(r) \sqcap arg'_0(r')$.
- Go to next $c \in L_G$.
- If any $c \in L_G$ fails, unify fails.
- Construct a new L_G and $L_{G'}$ from the concept nodes in the next level above the leaf nodes just processed (i - 1). Go back to the sorting step and proceed as before.
- When the head nodes are reached, unify them, and refer to the join as q''. If this join fails, unify fails.
- Return G'' as the result of unification.

The sorting step is necessary, since otherwise we would allow a leaf concept to unify with the first concept that it matches, which might be a compatible concept with a generic referent. The generic referent must be preserved until it is the last choice, to ensure that all concepts are considered for matching, and not only the concepts with generic referents. Next, we show an example where the sorting step is essential to the correct matching of the concepts.

Figure 43 illustrates the usefulness of CG unification by tree isomorphism. The CG labeled as G in Fig. 43 represents a customer requirement for a house to be designed by an architect. The Conceptual Graph G′ represents a partial design of the house, which has been retrieved from the architect's past designs. The two graphs are unified, resulting in the graph shown in Figure 44, which represents a unification of the customer's requirements with a design which partially matched the requirements.

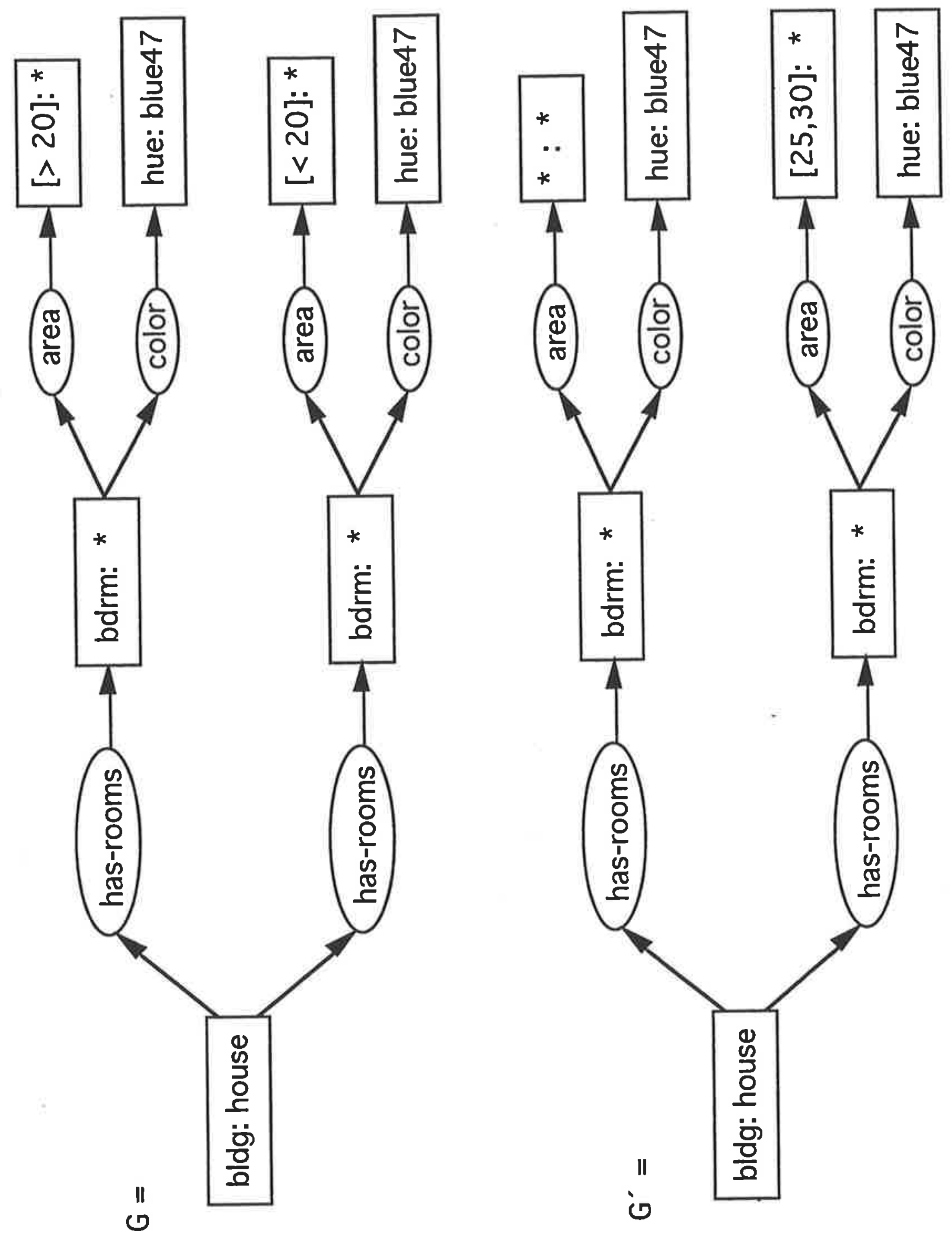


Figure 43. Unification by tree isomorphism.

Note that since the algorithm starts at the leaf nodes and works up, and since the nodes at each level are sorted by type and specificity, backtracking on the leaf node assignments is avoided. A simple tree traversal-type unification which started at the head nodes could have assigned the first bdrm the constraint of area > 20 to the generic referent, and the colors would have matched, then the other

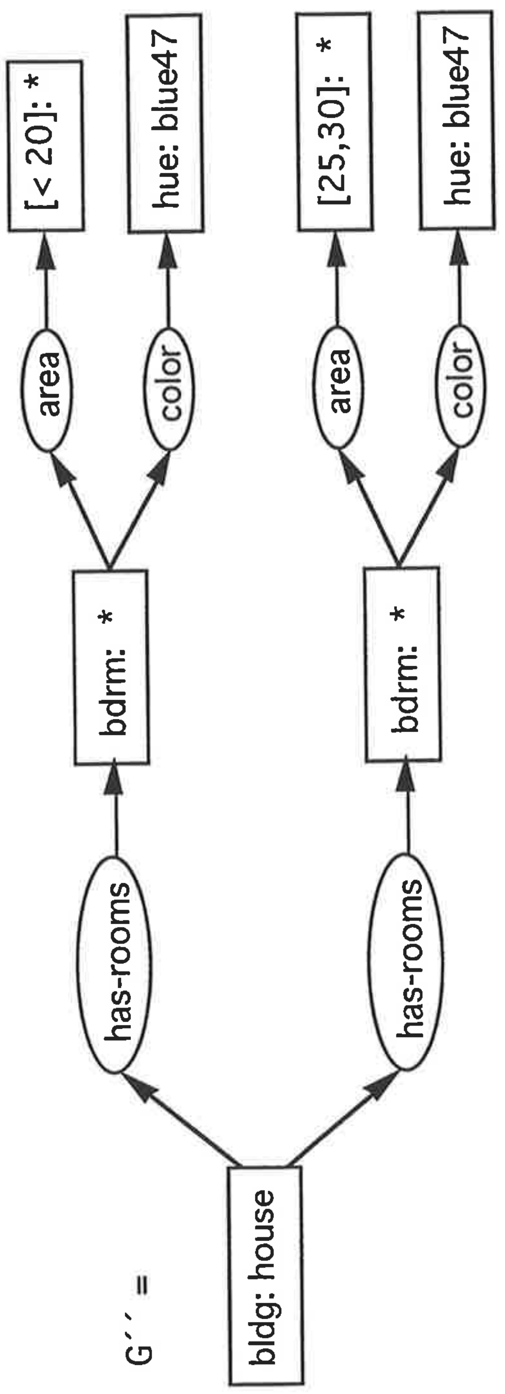


Figure 44. Result of tree isomorphism unification.

bdrm constraint of area < 20 could not have been matched. Instead, the tree isomorphism algorithm assigned the > 20 constraint to the interval [25, 30] and the > 20 constraint to the generic referent, which then allowed unification to proceed.

We discuss a formal comparison of Tree Isomorphism to the Unify algorithm (and others) in Chapter Five. Here we only state that the clear advantage of the

Unify algorithm is one of the flexibility of the knowledge structure. The Tree Conceptual Graph is very limited in structure and scope, and could only be used in certain domains and circumstances. Given these domains, however, Tree Isomorphism Unification could be an efficient method for merging knowledge.

4.5.5 Constraint Satisfaction Systems

As was mentioned in Chapter Two, there already exist many systems for handling constraint satisfaction problems. We now turn our attention to some of these systems, and discuss how these implementations of CSP compare to the system described in this thesis.

We first examine two CSP systems which use fairly standard techniques to resolve constraints. Numerica [Van Hentenryck et al. 1997] and ILOG [ILOG 1996] can also be used to perform the constraint processing of the type presented in this chapter. In fact, these two tools in particular can provide a high level of efficiency in processing simultaneous equations and simple types of constraints on variables. Numerica in particular handles constraints by using intervals, similar to the method used in this thesis.

One way to take advantage of this efficiency with constraints is to use the Unify system described in this thesis to handle the Conceptual Graph unification, and then make an external call to Numerica or ILOG to do the constraint processing. This could possibly make the constraint processing more efficient in the Unify system, if the constraints could be set up in a format compatible with the external tool. Essentially, the question of solving the constraints internal to the Unify system, or externally through a CSP tool is one of implementation.

The major point here, though, is that this promotes working outside of the Conceptual Graph formalism to solve parts of a CG problem. The Unify algorithm always works inside the CG formalism, using projection, subsumption and join. Making a call outside of the Unify system would have to be handled in a manner which would be sensitive to the CG formalism. For example, once the user exits the CG formalism, can the system guarantee a canonical graph as a result? Again, this is mostly an implementation issue, but an issue which does not exist when strictly using the Unify algorithm.

Recall from Chapter Two that Mugnier and Chein convincingly demonstrate that any CSP problem can be represented as a mathematical morphism [Mugnier and Chein 1996]. Their proof of the strong correspondence between CSP and the general problem of morphism (or projection) also demonstrates that a type hierarchy, such as used in Conceptual Graphs, can be effective in representing and solving a CSP problem. They develop, and prove the soundness of, algorithms for transferring CSP problems to a projection problem, and for transferring projections back to a CSP representation.

Mugnier and Chein demonstrate that the algorithmic techniques that they develop for resolving the problem of the existence of a solution to a Constraint Satisfaction Problem also can enumerate the solutions. Further, these are transferable from one domain to another [Mugnier and Chein 1996].

Prolog IV [PrologIA 1997] is an extension of Prolog which can also handle constraints. The difference between Prolog IV and constraint tools such as ILOG and Numerica is that Prolog IV is not just a constraint tool. Prolog IV is the full implementation of Prolog, complete with unification algorithm and Horn clause representation of knowledge, and it handles constraints using intervals.

The question again is whether Prolog IV can be used to implement Conceptual Graph unification with constraints as described in Chapter Three. This is certainly possible. The Unify algorithm was implemented in Lisp, using lists to represent the Conceptual Graphs. This method could certainly be used in Prolog to represent the CGs. If carefully implemented in Prolog, the built-in unification engine can be used to perform the unification of the CGs. But here we return to the main point: the algorithm described in Chapter Three can be implemented in many different systems and languages, including Prolog, but this is still working within the CG formalism to produce an extension to Conceptual Graph unification and constraints. The contribution of this thesis is not the implementation, but rather an algorithm for unifying Conceptual Graphs.

Unification (by projection) is the mechanism used in this thesis to find the solution to constraints on the values in CGs. The difference with this thesis is that here, the structures are also carrying a complex, powerful knowledge representation scheme along with the constraints. As discussed previously, unification starts off as the identifying of two logical formulae by variable substitution. In this thesis, unification is a tool which performs the work of identifying two structures using subsumption, where the elements of the structure can be constrained.

The systems discussed in this section really represent specific implementations of the CSP formalism. Prolog IV also contains some tools for unification and data structures. These implementations cannot readily be compared to the Unify system, because Unify is not simply another CSP tool, but rather an extension to the CG formalism. However, in Chapter Five, we discuss the broader implications of this system.

4.6 Summary

The main contribution of this chapter was to explore the workings of the formalisms presented in Chapter Three, and to explore the boundaries of the implemented work. The chapter began with a detailed description of the SEED project. SEED (and building architecture) is a complex and interesting domain, which until now had lacked a sufficiently powerful knowledge representation scheme to be implemented properly.

Most of the examples in this chapter are taken from the SEED domain, and show that this domain can be represented completely by the system of Conceptual Graphs with unification and constraints defined in this thesis. In section 4.4.5, we conclude that knowledge representation for the SEED project had three major requirements: 1. type subsumption to handle generic objects in the domain, 2. knowledge-level reasoning in the language of the domain, and 3. pattern matching for query and "what if" processing. All three of these requirements are met by the

Unify system described earlier in this thesis. The software as implemented allows the theory and software of the SEED project to move forward.

The Unify algorithm was tested against a large data set, which included various problems and situations. The Unify system performed well in constraint processing, graphs with cycles, and graphs with multiple nodes which could be considered the head node.

When tested in a backtracking problem, the Unify system failed to unify. At first, this seemed like a problem in not being able to permute the arguments to the relations in the graph, but it must be remembered that permutation of the relation arguments is not a part of the Conceptual Graph formalism. Each argument of a relation must unify with the same argument of its corresponding relation. Thus, the two graphs in Fig. 26 do not unify. The last section of this chapter discusses the implementation of an algorithm which will accept these same two arguments and unify them, according to a tree isomorphism algorithm. Tree isomorphism as used here is outside of the standard formalisms of Conceptual Graphs, but may be of use in certain implementations. Chapter Five presents several methods of unification for comparison. Some of these methods could be used to extend the unification model presented earlier, some could be alternatives, and some may extend beyond the current Conceptual Graph canon. These are discussed further in the next chapter.

With respect to standard constraint satisfaction techniques, this chapter emphasizes that Conceptual Graphs and the Unify algorithm are designed to manage large amounts of domain knowledge efficiently. CSP tools are designed to manage large amounts of arithmetic constraints efficiently. A merger of these two techniques, such as calling ILOG from the Unify algorithm, may produce a very efficient CG constraint system. However, it is most important to remember that the Unify system is a theoretical contribution to extending the Conceptual Graph formalism, which could be implemented in any number of ways. CGs are a complex and powerful knowledge representation scheme. The contribution of this thesis is that now they can also carry constraints on the values of the concepts, and unify the graphs in a way which guarantees a canonical result.

One area which gave the Unify algorithm some trouble was in handling unary relations. Since unary relations are generally used to change the truth value of a graph, or for a non-semantic purpose, such as indicating the head node, it is difficult to incorporate their use into a unification routine. This discussion led into the discussion on the validity of graphs produced with the Unify algorithm. While the algorithm guarantees a canonical result, there is no attempt at performing truth maintenance on the knowledge base. In other words, while we can guarantee that the graph produced by the Unify algorithm will be within the defined canon, it is up to the user or the truth maintenance programmer to validate the graphs against the current state of the world. It is possible to keep impossible situations from happening, such as having blue cats, but it is not possible in the Unify algorithm to say whether Felix really is a black cat in the real world.

We then focused our attention on Conceptual Graph unification and constraint systems which have already been implemented. Willems created an interesting attempt at unification, which concentrates on finding the lowest common generalization of two graphs. While this can be useful in some domains, it does not meet the requirements of finding a Greatest Lower Bound of two graphs. However, generalization has the advantage over Unify of finding any common ground between two graphs. We discuss in Chapter Five how this might be used in situations where it is not possible to produce a Greatest Lower Bound, and we suggest that the complete Conceptual Graphs Implementor's Toolbox might include a Willems-style generalization tool for progressing data exploration when no other common ground can be found.

Kocura's structural constraints were presented in an example from the SEED domain. SEED structures and constraints could be represented using Kocura's method of constraining the type hierarchy by using negative canonical models. However, it seems that the type hierarchy can be rearranged in cases such as the "studio-flat" and the "blue cat" in order to provide a type hierarchy which conforms more closely with the domain under consideration. Kocura's method also guarantees a canonical result, but makes an additional step to produce the results. Whenever a canonical formation rule is specialized, Kocura's algorithm must perform an additional check to see whether any of the negative models have projections into the newly specialized graph. The Unify algorithm does this all in the same process.

Mineau uses actors, implemented as procedural attachments, to represent domain constraints. Whenever a new graph is created which involves the use of one of the actors, the new graph is checked to see whether the actor still has a projection into the new graph. If not, then the graph violates the constraints, and is invalid. Mineau's method produces a canonical graph, but takes an additional pass through the graph to accomplish it. As with some previous systems, Mineau's technique works outside of the standard formalism of Conceptual Graphs, by calling additional processes to check a new graph after an attempted unification. The Unify algorithm can propagate constraints through a unification process which works with projection, subsumption and join inside the Conceptual Graph formalism.

Overall, this chapter demonstrates that the algorithm described in Chapter Three has been implemented, works on various constraint and unification problems, and exceeds the limits of previous CG tools. Unify provides both a CG unification tool and a constraint solver for values of the concepts. A combination of the Unify algorithm with efficient constraint solvers might be an interesting and efficient way forward for new CG tools. On the other hand, this thesis can also be used as a beginning of the combination of complex knowledge structures with standard CSP techniques.

Finally, this chapter presents an initial method for bringing together Conceptual Graphs, constraints and Knowledge Representation to solve a unique problem in design and architecture. The Unify algorithm and the Conceptual Graph formalism provide the representational power needed to explore constructive architectural design in a manner easily understood by the user. We discuss in Chapter Five how these methods have general applicability, and can be combined into a useful "toolbox" of knowledge conjunction tools for the knowledge engineer. Chapter Six presents a discussion on the future directions of research which combines constraints, CGs and unification.

Chapter 5

Placing the Unification Model in Context

5.1 Introduction

This chapter will explore how the unification model presented in this thesis can become part of a practical set of software tools for the Conceptual Graph designer, programmer and user. We examine some methods for organizing the new extensions and the old formalism into a unified set of tools. This new organization will show how these tools can be compared to each other for usefulness in a given domain or set of software.

This chapter also explores the applicability of our method and algorithm to other domains. Some of the work in this chapter was presented in a preliminary form in some of my earlier publications [Corbett 2001; Corbett 2000; Corbett and Woodbury 1999].

5.2 A Framework of Unification Methods

5.2.1 The Purpose of the Framework

The early chapters of this thesis examined several types of graph combination techniques. It was demonstrated that some attempts at merging graphs often lose some of the knowledge contained in the graphs being merged. Other techniques have a high cost in complexity. In many domains, it is essential that the combining of knowledge be represented as a continual refining and specifying of the knowledge, so that none of the knowledge is lost in the process of combining with other knowledge [Corbett 2000; Corbett and Woodbury 1999; Woodbury et al. 1999; Chang and Woodbury 1996].

In order to produce feasible, usable structures, unification of two conceptual structures must be shown to produce a unique result, which is a sensible conceptual structure. The fact that CGs admit a unification operation is crucial to their efficient application to the representation of knowledge conjunction. Knowledge conjunction is the idea that when two structures are unified, the result contains no more and no less information than the two original graphs. By using a consistent unification technique, knowledge from a number of distinct sources can be combined into a unique representation [Carpenter 1992].

In this section, we present examples of Conceptual Graph unification techniques, organized into an ordering on complexity and flexibility of the algorithms. Some recent unification algorithms are quite flexible and easy to use, but require additional checking functions or additional structures which increase the complexity of the algorithms. Other algorithms do not have a need for the additional functions, and therefore are much more efficient, but only at the expense of restricting the CGs in some way. These restricted CGs are still usable in many domains, but do not have the complete flexibility of unrestricted CGs.

Each type of unification technique is accompanied by an analysis of the technique, so that they can be compared to each other, and to future techniques. The intent of this section is to present a framework of knowledge conjunction on

Conceptual Graphs, which will allow a comparison of unification, join, or other merge technique to each other. The framework presented in this section is incomplete, as only one example is selected in each category. The intent is for this framework to be filled out in the future.

5.2.2 The Common Sub-graph Approach

Recall from Chapter One that Willems' approach to unification is to use type subsumption to compare compatible nodes [Willems 1995]. Two CGs unify in Willems' sense if they each have some common sub-graph, which can be joined under the usual type subsumption rules for CGs. The unified graph is then the joined graph, plus all the other relations and concepts in the two original graphs.

Willems' unification algorithm [Willems 1995] basically finds a segment of the CG which is common to both of the CGs being unified. In this approach, Willems finds a projection which is at least as (or possibly more) general than both of the graphs being unified. This makes the unification algorithm efficient, as it takes advantage of the built-in CG attributes of subsumption and type hierarchies.

Willems' approach makes no attempt to implement a method for producing the unified graph, however. His effort is mainly to find the unifier. It is therefore unclear whether creating a generalization of a concept loses the essential knowledge that the user wants to retain.

Willems' algorithm constructs a set of all concepts that are of the same type (similarly for relations). The algorithm then deletes pairs of concepts (or relations) which do not preserve the original structure of concept and relation arguments. When all these pairs have been deleted, then what's left is either the unifier, or an empty set (indicating a failed attempt to unify). Since Willems restricts his definition of projection to a simpler operation, called polyprojection, he claims that his unification algorithm is executable in polynomial time on the number of nodes in the graph. (In the worst case, the algorithm deletes all pairs of nodes, one by one.)

Analysis

Useful domains: Generalization (anti-unification) of conceptual graphs.

Complexity: Polynomial.

Flexibility: No restriction on the representation, but the algorithm only finds a common generalization as the unifier, not a common specialization.

5.2.3 Fuzzy Conceptual Graph Unification

Recall again from Chapter One the work of Cao et al [Cao et al. 1997] in the implementation of fuzzy values in Conceptual Graphs. These implementations are useful for specifying boundaries for the values in concepts, but Cao has implemented them for the purposes of unification and resolution proof procedures for fuzzy programs.

Cao defines a modus ponens, among other Fuzzy CG operators and functions. The fuzzy tolerance degree is used to unify the program to a query, and to produce a unified fuzzy CG as a result. Cao discusses the algorithm in [Cao and Creasy 1998; Cao et al. 1997] but does not come to a conclusion as to the complexity of the algorithm, as some completeness issues are still being resolved. He implies that a polynomial complexity could be expected.

Analysis

Useful domains: Uncertain data or domain rules.

Complexity: Probably polynomial.

Flexibility: No restriction on the representation. Most useful as a resolution proof procedure in a fuzzy domain.

5.2.4 Term Resolution

As discussed in Chapter One, Müller [Müller 1997] concludes that CG unification is only guaranteed for graphs which have a labeled "head" (or root) node. Müller demonstrates that the most general unifier of two headed conceptual graphs can be constructed by restricting the types of both head concepts to their join, and restricting their referents if necessary following an external join. Müller then concludes that since an external join is a linear operation, and since there exist implementations of lattices that have a join operation of constant complexity [Aït-Kaci et al. 1989], the mgu of two headed CGs can be computed with linear complexity. Müller does not discuss how he handles backtracking or cycles.

Analysis

Useful domains: CGs as domain objects; Resolution Theorem Proving.

Complexity: Linear.

Flexibility: Restricted to headed CGs; assumes efficient lattice operators; preserves knowledge conjunction.

5.2.5 The Unify Algorithm

The method described in this thesis uses intervals to bound the value of an attribute, thus capturing the idea of a constraint on a real number. Subsumption of an interval (defined on an interval lattice) is used to decide whether two concepts of the same type are still unifiable. The lattice operators "join" and "projection" are used to decide subsumption. We are then able to define real-valued constraints in concepts, and use the standard join operation to decide whether a concept is valid according to the domain knowledge. The detailed analysis of this algorithm is contained in Chapters Three and Four.

Analysis

Useful domains: Has been implemented on architectural design domain; some work on grammars requiring head concepts.

Complexity: Linear

Flexibility: Restricted to finite, headed CGs. All standard CG techniques (type subsumption, canonical formation rules, etc.) still apply.

5.2.6 Tree Isomorphism

Tree Isomorphism was discussed in the previous chapter. Here we summarize the analysis of the algorithm:

Analysis

Useful domains: Taxonomies and other strict hierarchies.

Complexity: Linear.

Flexibility: Strict rules on the structure of the knowledge; finite trees only, with binary relations.

The techniques and analyses discussed in this section are summarized in Table 1.

Technique	Domains	Complexity	Flexibility
Willems (segment gluing)	Generalization	Polynomial	Unrestricted structure; no common specialization
Cao, et al. (fuzzy CGs)	Uncertainty	Polynomial	Unrestricted structure; useful for fuzzy proof resolution
Müller (term resolution)	Resolution Theorem Proving	Linear	Headed CGs only
Intervals	Design	Linear	Finite headed CGs
Tree Isomorphism	Taxonomies	Linear	Finite trees only; binary relations

Table 1. Summary of Results.

5.3 A Toolbox for Knowledge Conjunction on Conceptual Graphs

5.3.1 The Complete Toolbox

All of the techniques discussed in this chapter are useful for a comprehensive system of unification in Conceptual Graphs. All of these tools of unification, generalization, "gluing" of CGs, fuzzy matching and tree isomorphism can be combined into a comprehensive set of tools for handling CG knowledge conjunction.

This thesis has discussed two important approaches which help to make Conceptual Graphs tractable when it comes to unification. These are the formal definition of Headed CGs, and the formal definition of an algorithm for unification based on projection. We now consider what other operations might be useful for a complete toolbox of Knowledge Conjunction over Conceptual Graphs.

5.3.2 Generalization for Knowledge Conjunction

As discussed in Chapter Two, generalization is the dual function to unification, and can be used in many of the same domains as unification [Knight 1989]. Generalization is also known as anti-unification [Lassez et al. 1988]. The usual definition of generalization is, given two objects x and y, is there some third object z, of which both x and y are instances? In parallel to the greatest lower bound, the object which is the most specific of all common generalizations of two objects is know as the *least upper bound*. The most specific generalization of two terms retains some information that is common to both terms, introducing new variables (essentially "unbinding" variables) when information conflicts [Knight 1989]. Knight gives the example that the most specific generalization of the two terms $f(a, g(b, c))$ and $f(b, g(x, c))$ is $f(z, g(x, c))$. Since the first argument to f can be a or b, generalization abstracts the terms to the variable z. Unification of these terms, however, would simply fail.

It is this functionality of generalization that we're trying to capture in this case. Take for example the "Blue Bedroom" problem shown in Figures 43 and 44. In a complete toolbox, the user would be told that the two graphs cannot unify by using any of the standard techniques for CGs. The user would be offered the choice of allowing the unify to fail, using tree isomorphism to conjoin the graphs, allowing a generalization of the concepts, user choice of the identification of branches, or allowing a permutation of the branches of the tree (and therefore backtracking). These last two operations take care of any non-determinism by allowing the user to select the match, or by allowing backtracking. The toolbox could even leave the choice of whether and how to backtrack to the user.

Using the example from Figures 43 and 44, if the user chooses to use the generalization option, then the referents in the concepts would be generalized and compared for compatibility. The hue: blue47 referent could be generalized to be just hue: * to represent the idea that some color must be represented there. There would be some confusion with the area constraints however, since <20 and >20 are in a sense already general. It is also unclear how one would generalize the referent [25, 30]. Questions of how knowledge conjunction would work with generalization when there are constraints over the referents are left to future work.

The generalization option allows the user to assert that the graphs in question have some elements in common, but can not be unified. It may be useful in some domains to be able to state that graphs have common elements, even when they can not be unified. This is similar to Willems' approach [Willems 1995] discussed in earlier chapters.

5.3.3 Flexibility and User Choices

Testing of the Unify software included experiments with several ideas which are not part of the standard CG literature. These experiments included the definition of variables and variable binding. A value constraint in a concept could be specified as, for example, [10, *n*], where *n* is a real variable. In this case, if the [10, *n*] constraint is being joined with an interval less than 10, then the unify would obviously fail. If the other constraint is greater than 10, then the constraint is considered to be undecided until the variable is bound to some value. The constraint is carried in its original form, and propagated through the system until the constraint can be resolved. Since it is the original form which is propagated, the variable can be assigned another value and the constraint tested again. In Common Lisp, variables can never be unbound, but unbinding a variable was simulated by having the software recognize nil as meaning unbound.

Similarly, the > and < symbols were allowed in the software. It is possible to specify constraints such as [> 10] or even [< *n*]. However, this is not much of a departure from the standard, since this really only represents an alternative notation for [10, ∞] or [-∞, *n*].

A third area which was tested was in allowing the user to specify the priority that some constraints have over others, and in allowing constraints "tolerances." In terms of constraint priority, the user could specify that the color of the room is not important enough to cause a failure of unification. In this case, the user could specify that if Unify were going to fail only because of a clash of colors, then the constraints could be generalized (to something like [hue: *]) and the rest of the graph unified in the normal way. Priorities were implemented in the Unify algorithm by allowing the algorithm to attempt to unify the concepts. When that failed, the algorithm would examine the priority of the constraint involved (as set by the user) and use a least upper bound of the types to find a compatible concept. Of course, this returns us to the problem of preserving constraints through a generalization of the concepts, which this algorithm makes no attempt to do.

Tolerances were specified for each type before the software was called. So, in the case of an interval that is pointed to by an area relation, we could specify that if the intervals do not match, but were within 5% of the size of the interval, the unify would succeed, with a value selected half-way between the two end values.

The (as yet unimplemented) vision that we have of the Knowledge Conjunction on Conceptual Graphs Toolbox is one of a complete set of tools for the CG designer and user. The ultimate knowledge conjunction toolbox would allow the user to draw the Conceptual Graphs, while the system verified that they were canonical. The user could then use the graph drawn on the screen as an index key to search through a knowledge base for a matching concept. (This is the idea underlying the SEED system.) When a suitable match is found, the user can attempt to extend and specify his creation by unifying it with the new-found graphs. If the Unify algorithm fails, the user can choose to allow some concepts to be generalized, rather than specialized. The user could constrain the values on appropriate concepts, and even use variables to specify unknown values.

5.3.4 Some Thoughts About a Unification Toolbox

An important issue which needs to be considered in the near future by the CG community is the question of what exactly we're trying to get from unification of graphs. Are we after unification at all costs, or do we really want to concentrate on flexibility of domain knowledge structures? Clearly, there are compromises to be made, but does the choice of compromise depend on the domain, or on the unification technique or algorithm? Does the domain knowledge necessarily dictate the structure of the graphs, and therefore the unification technique to be used, or can we use flexible, all-purpose graphs for many types of domains?

Besides the answers to these questions, the contributions of other researchers to help fill out the framework of knowledge conjunction in the previous section would be welcome. An equitable method for comparing the various uses, domains, complexity and costs of these methods would allow the users in the CG community to select an appropriate scheme for their own use.

5.4 Extending the Model

5.4.1 Other Domains and General Applicability

Up to this point, our examples have mostly been from the domain of building architecture. In order to demonstrate how the ideas presented in this thesis apply generally, we now present some examples of the use of unification and constraints in a different domain. The intent of this section is to show that the method described in earlier chapters can not only be applied to other domains, but that many standard Artificial Intelligence techniques can also be represented using this method.

5.4.2 Rules of an Operations Officer

As our example, we discuss the use of unification and constraints for applying rules in a defense domain. An Air Operations Officer (usually known as an OPSO) is the defense officer responsible for deciding the appropriate defensive response to an air threat. A study of the Operations Officer decision-making methods was recently conducted, using a cognitive modeling technique [Mitchard et al. 2000; Mitchard 1998]. The study was used to show the usefulness of cognitive modeling in deriving rules from expert knowledge. In this section, we only make use of the rules which resulted from the study; the cognitive modeling technique is not discussed here.

In the domain of the Operations Officer, the magnitude of the response to an air threat is in proportion to the threat itself. So, if the opposing aircraft are very

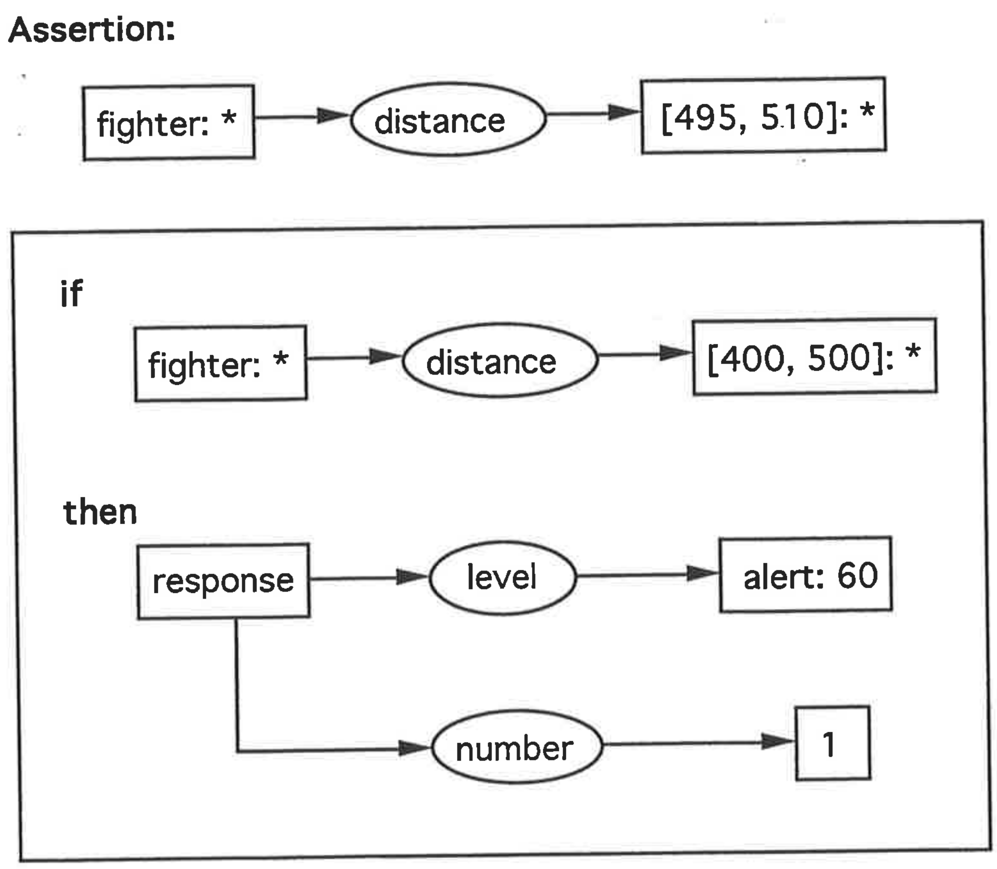


Figure 45. A rule in the defense domain, which uses constraints.

close, or if the aircraft is of a type which can cause a great deal of damage (known as a *strike* aircraft), then the response is large. If the threat is smaller, then the response is smaller. For example, Figure 45 shows a rule in this domain. (We have borrowed the style of Cao (as illustrated in Chapter One) to express the rule, although we do not employ fuzzy reasoning here.) This graph expresses the rule that if a fighter aircraft (small threat) is between 400 and 500 nautical miles distant, then assert a threat level of "alert 60" (the lowest level of alert, in which response fighters must be ready to take off within sixty minutes), and a single fighter is assigned to deal with this threat.

The assertion shown in Fig. 45 unifies with the "if" portion of this rule. The "then" portion represents the response to the situation, and it is asserted into the current world knowledge. In this manner, we can represent the decision-making capabilities of the Operations Officer.

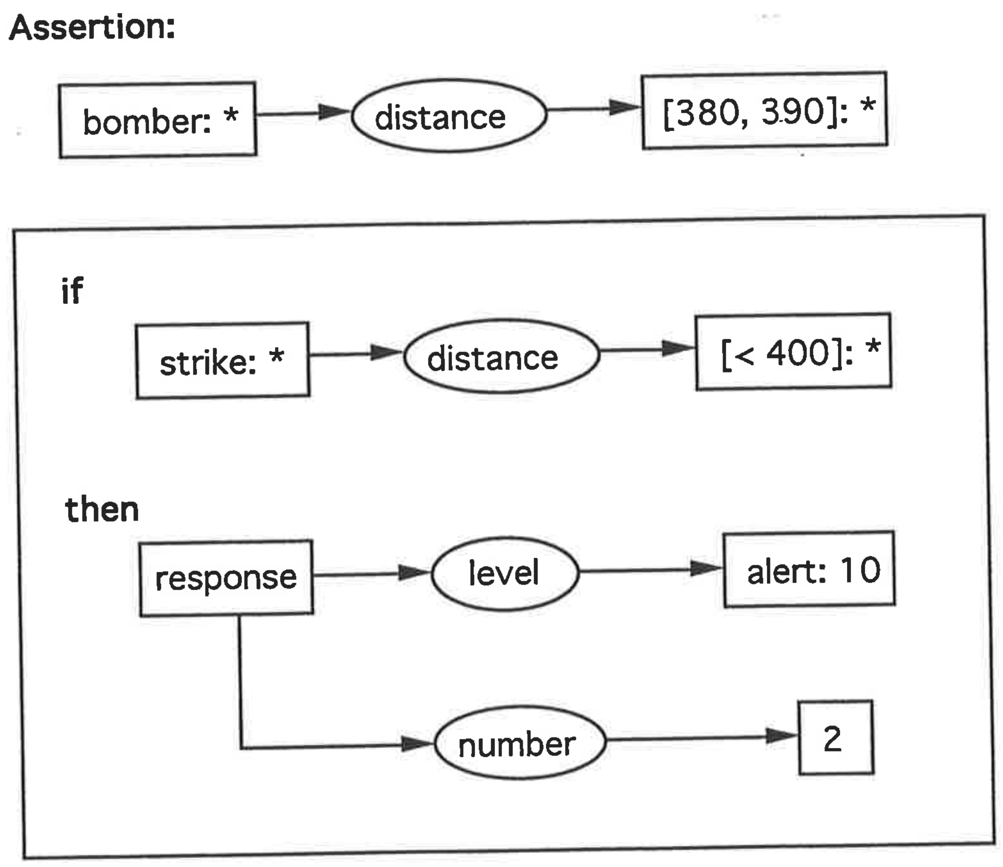


Figure 46. Another rule from the same domain.

The rule shown in Figure 46 is used for a bigger and more impending threat. Any threat aircraft which is closer than 400 nautical miles is considered an immediate threat, and a response squadron must be ready very quickly. Further, a strike aircraft is one which can inflict a great deal of damage, and is therefore dealt with more severely than a fighter craft.

The assertion shown in Fig. 46 states that a bomber is known to be between 380 and 390 nautical miles distant. Our type hierarchy indicates that a bomber is a type of strike aircraft. Because of the proximity of the threat, the response aircraft are put on "alert 10" status. Because of the enormity of the threat, two fighters are assigned to deal with the target aircraft. Again, the assertion unifies with the "if" portion of the rule, causing the "then" portion of the rule to be asserted.

5.4.3. Conclusions Regarding General Applicability

The intent of this chapter was to demonstrate that the Unify algorithm can apply generally to a wide variety of applications, domains and situations.

Ultimately, unification of Conceptual Graphs (as with unification of any type) is merely theorem resolution - in the case presented in this thesis, with constraints added. For example, rule learning could be implemented by starting with a set of simple rules specified in CGs. Every time there is an exception to a rule, generalization could be used to make the rule more general. Unification could be used to make more specific rules by combining rules.

Variable arity (as proposed in the next chapter) can be used in domains where it is not clear how all the objects in the domain relate to each other, or in domains where the relations between objects can change. Tree isomorphism can be used to eliminate backtracking in domains where all of the knowledge can be represented as simple trees.

Essentially, what this thesis has done is to demonstrate an efficient implementation of theorem resolution on Conceptual Graphs with constraints. This chapter serves to demonstrate that this type of resolution can be applied to many types of domains and knowledge.

5.5 Summary

The contribution of this chapter was to show that the theory of Conceptual Graph unification presented in the previous chapters is applicable generally to many domains and uses.

In the section on the framework of unification methods, we discussed the idea that the various methods of unification could be organized into a table-like frame which could be used to compare unification methods to each other. The methods were compared in terms of flexibility of structure, complexity of the algorithm and the possible domains which would benefit. There was also some discussion on how such a framework could be used for future comparisons of unification methods. The framework would be most useful if unification researchers in the CG community were to expand the framework with further attributes and methods.

We then turned our attention to one of the proposals for the Conceptual Graphs community in this thesis: The Knowledge Conjunction Toolbox. The Knowledge Conjunction Toolbox is proposed as a future research project for the Conceptual Structures community. Such a toolbox would allow the user of CGs great flexibility in using Conceptual Graphs for knowledge representation. There would be a wide range of tools for creating, editing and manipulating knowledge using Conceptual Graphs.

In order to demonstrate applicability of the Unify algorithm to various domains and structures, we explored a rule-based system in the defense domain. Rules were constructed which represented an assessment of the level of threat posed by attacking aircraft. If an assertion of the current state of the world unifies with the "if" portion of a rule, the "then" side of the rule is asserted into the knowledge base. This approach captures the idea of rules which can be activated by unification with an assertion in the current context. Constraints can be introduced into either the rules or the assertions.

Finally, there was a discussion regarding the general applicability of the work. The nature of the Unify algorithm is such that it can be bent and pushed in a number of directions to be useful. Ultimately, it all comes down to theorem resolution using Conceptual Graphs. Since this is the case, the algorithm has broad practicability and usefulness.

Chapter 6

Conclusions and Future Work

6.1 Contributions

This thesis has made several major and minor original contributions to the areas of Conceptual Graphs, constraints on concepts, unification of Conceptual Graphs, and the Conceptual Graphs formalism. These contributions are detailed in this first section.

Before it was possible to define real-value constraints for Conceptual Graphs, it was first necessary to define a formalism for real values in CGs, as there was no formal definition of real numbers. This was done by defining an interval concept type, and the meaning of the interval type hierarchy. Real numbers are represented by bounding the limits of the number with intervals. A formal definition of the interval type hierarchy was presented, and the background of similar work was explored. It was shown that while this defined a new functionality for Conceptual Graphs, the definition of interval was consistent with previous work in constraints and interval representations.

The next contribution of this thesis was the formal definition of real-value constraints in CGs. Prior to this work, there was no standard method for checking the validity of the values of the concepts, only for checking the structure or canonicity of the graph. This thesis presents a method such that, when the knowledge of two graphs is combined, the constraints on the concepts can be verified.

The third contribution of this thesis is in creating new definitions of constraints which are compatible with the previous body of research in Conceptual Graphs by working within the existing CG framework. This was not simply a matter of applying the new techniques to CGs, but rather required a reworking of some of the basic definitions of Conceptual Graphs, so that they are consistent with the constraint definitions. The new definitions can still operate with all previous definitions of Conceptual Graphs, such as type hierarchies, join, subsumption, etc. The new definitions of old functions involved collecting the various versions of the formal definitions of projection, Conceptual Graph, type hierarchy and canon, eliminating inconsistencies, and then laying out their formal definitions in a manner which would be sympathetic to the Headed Conceptual Graph, unification and constraint definitions. Also, the join function was placed in its proper context with respect to these new definitions.

One of the major contributions of this thesis is that a unification method is defined that leads to a useful and efficient implementation of constraints over CGs. The major significance of this work is that it improves on previous work in allowing constraints to be placed on real values in the concepts. The constraints are defined as a concept type, and therefore can be used as a type in the normal way with Conceptual Graphs. The unification algorithm used in this system is an improvement on previous attempts at CG unification, has a complexity of O(n) on the number of relations, and is guaranteed to terminate since we restrict the CGs to a special subset. The Unify algorithm presented in this thesis will guarantee that two canonical graphs unified under the algorithm presented in Chapter Three will produce a graph which is canonical and is the Greatest Lower Bound of the two graphs, if they have a glb.

This thesis also introduces the notion of *knowledge conjunction* for Conceptual Graphs. Unification of CGs must be a system in which two pieces of partial information can be combined into a single unified whole. Unification here is the combining of pieces of knowledge, represented as Conceptual Graphs, in a domain. We define unification as an operation that simultaneously determines the consistency of two pieces of partial or incomplete knowledge, and if they are consistent, combines them into a single result.

The next contribution was a major component in formally defining unification for Conceptual Graphs. Projection and Sowa's π operator were redefined to work with unification, constraints and variable-arity relations. The formal definition of projection was based on some of the recent work in graph unification, but this was the first formal redefinition of Sowa's π operator. The previous definitions of projection would not have worked with constraints on the referent values of the concepts.

These new formal definitions led to the first definition of real numbers and real-valued constraints for Conceptual Graphs. The definition of real numbers in CGs allows an opening for Constraint Satisfaction Problem research in CGs. The discussion in Chapter Two on the usefulness of CSP described recent work that demonstrates that any CSP problem can be represented as a mathematical morphism, and therefore as a projection problem. The strong correspondence between CSP and the general problem of projection also demonstrates that a type hierarchy, such as used in Conceptual Graphs, can be effective in representing and solving a CSP problem. While this thesis does not deeply explore the possible connections between CG constraints and CSP, the door is left open for future work in linking these two areas.

This thesis also contributes a proposal for the formal definition of variable-arity relations in Conceptual Graphs in the Future Directions section of this chapter. Allowing variable arity relations opens the door to new domains and new knowledge representation possibilities.

Finally, in order to test the formal aspects of this thesis, all of these techniques and definitions were implemented in software in Allegro Common Lisp. The software was tested not only in the SEED domain, but also in unrelated domains, such as air combat. Many structures and types of knowledge were tested, in real-world domains, using test data from domain experts. There were also experiments with variables and variable binding, approximation and constraint prioritization. These successful experiments will lead to future exploration in areas of knowledge conjunction.

6.2 Conclusions

This thesis has addressed two important areas in the field of Conceptual Structures. The first is the unification of Conceptual Graphs, and the consequent work in projection and in type hierarchies. The second is the definition of constraints, especially real-value constraints on the concept referents, with particular attention to handling constraints during the unification of Conceptual Graphs.

The significance of this thesis is that a unification method is defined that leads to a useful and efficient implementation of constraints over CGs. This improves on previous work in allowing constraints to be placed on real values in the concepts. The constraints are defined as a concept type, and therefore can be used as a type in the normal way with Conceptual Graphs. The constraints are enforced in the unification and join operations. If a join operation violates the constraints on one of the concepts, the join fails. The unification algorithm used in this system is guaranteed to terminate, has a low complexity, and is sound and complete.

The Unify algorithm produces not only a canonical graph, but one which is valid in the domain. For example, in our domain of building architecture, there's a sense that designing a house is a continual refining and specifying of the design. In this sense, at every step the user will produce a *valid house design* (minus specializations). The Unify algorithm guarantees that, for any domain, once the domain hierarchies and canonical formation rules have been properly defined, the only structures that will be allowed will be those that represent reasonable knowledge in the domain.

This research contributed significantly to an automated architectural design project, SEED. The three main areas where the SEED Project benefited from the formalization of Conceptual Graph unification are in type subsumption, knowledge-level reasoning, and pattern matching. The Unify algorithm allows the user to specialize designs by matching (unifying) previous designs with the current design problem. Since all characteristics, attributes and constraints are carried along in the unification, the specialization represents all of the design concepts included in the more generic design.

In the area of knowledge-level reasoning, a graph can be manipulated as a whole, and treated as a room (for example), rather than a square in a diagram. This approach frees the architect from dealing with data-level concerns of numbers and coordinates, and allows the architect instead to deal with the architectural design.

As for a pattern matching ability, the user can specify a type of structure for support, and make a query by attempting to unify the structure with the more generic design. If the unification fails, then the user knows that the proposed structure does not meet the constraints of the design problem. If the graphs unify, then the resulting graph will contain the constraints which must be met in order to make the design work.

6.3 Future Directions

The approaches taken in this thesis suggest several other possible avenues of research in the areas of Conceptual Graphs, Knowledge Representation, and Knowledge Conjunction.

6.3.1 Variable-arity relations

The one area where Feature Structures appear still to have an advantage over Conceptual Graphs is in extensible arity. The ability to allow flexible arguments, not restricted by order, is one feature lacking in Conceptual Graphs. Recall also from Chapter Four, Figures 18 and 19 which could not be unified due to a mismatch of the arity of the *adjacent* relation. In this section, we propose a new model for Conceptual Graphs with variable arity relations. Our proposal for a variable-arity model for Conceptual Graphs will involve having a new look at the projection operator, and also our definition of unification.

There are some domains which currently do not fit easily into the standard definitions and formalisms of Conceptual Structures, due to the dynamic nature of the knowledge being represented. Some simple examples include a variable number of students in a class, or several exits from a room. In these cases, the number of concepts to be "attached" to the relation is not known beforehand. The use of variable arity relations will allow representation of knowledge where a relation may not point to the same number of concepts in every situation.

To make variable arity relations consistent with the rest of the CG formalism, it is first necessary to modify the standard definition of a CG slightly [Corbett 1999].

Definition 2´´. A conceptual graph is a tuple $G = (C, q, R, type, referent, arg_1, \ldots, arg_m)$ where:

C is the set of concepts, $type : C \rightarrow T$ indicates the type of a concept.

$R \subset T \times C \{\times C \times \ldots \times C\}$ is the set of relations between the concepts. T is the set of types and $(t, c_i, \{c_j, \ldots, c_m\}) \in R$ yields an arc of type t from concept i to concept j, and possibly also to other concepts.

In this definition, there are optional arguments to each relation, which may be used in a given circumstance, thus creating relations of variable arity.

Some thought must then be given to the definitions of the join and projection operators. The objective of a join operation is to find a graph which represents knowledge which is *more specific* than the two concepts being joined. It is not difficult to define a join operation which includes variable arity, if we let *higher* arity mean *more* specific. For example, a kitchen design specifies a complete lighting design, but only relates to a few plumbing concepts. Another kitchen design specifies different aspects of the plumbing. The join of these two graphs is a graph with both lighting and plumbing completely specified. The joined graph is more specific because it is more completely specified, and the arity of the relations is higher in the joined graph.

The projection operator expects that each relation in the previous graphs will still point to the same concepts in the new graph. Recall the third requirement from the definition of Projection in previous chapters, which states that each argument from the more general graph must point to the appropriate concept in the more specific graph:

$$\forall r \in R, arg'_i(h_R(r)) = h_C(arg_i(r))$$

In our proposed definitions for variable arity unification, we still want to guarantee that if a relation points to a given concept in a one graph, it will still point to it in the new, unified graph, but we do not guarantee that the order of the arcs is preserved. The semantics of the relation are still guaranteed though, in that even though a concept has changed places, its meaning is still intact. This obviously would not work for all domains, and the canonical formation rules would specify which relations were to have a variable arity in a given domain.

Another issue which arises from the definition of variable arity is the definition of the mgu in a unification process. This definition follows on in a straightforward manner from the modification to the projection operator discussed above. If we accept that a relation of higher arity (for example *adjacent*\3) is more specific than the same relation with lower arity (e.g.,

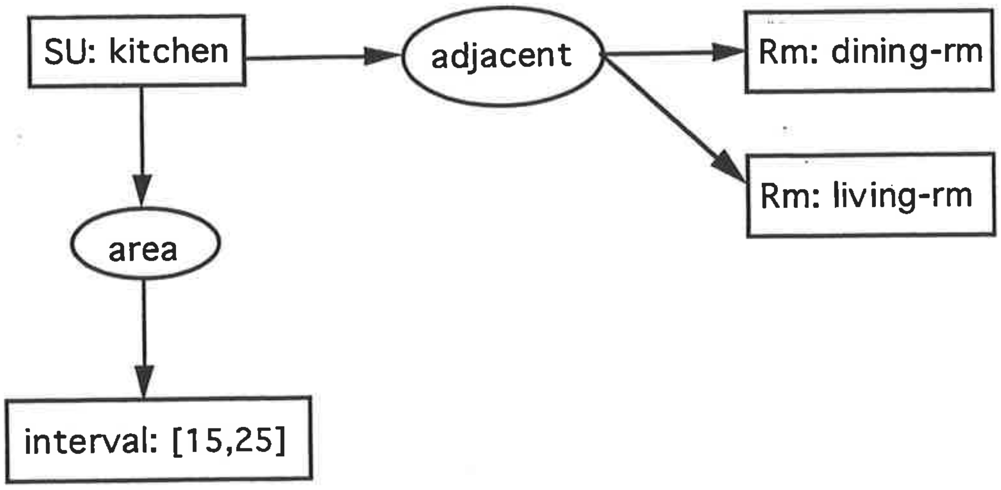


Figure 47. A design with arity 2 relations.

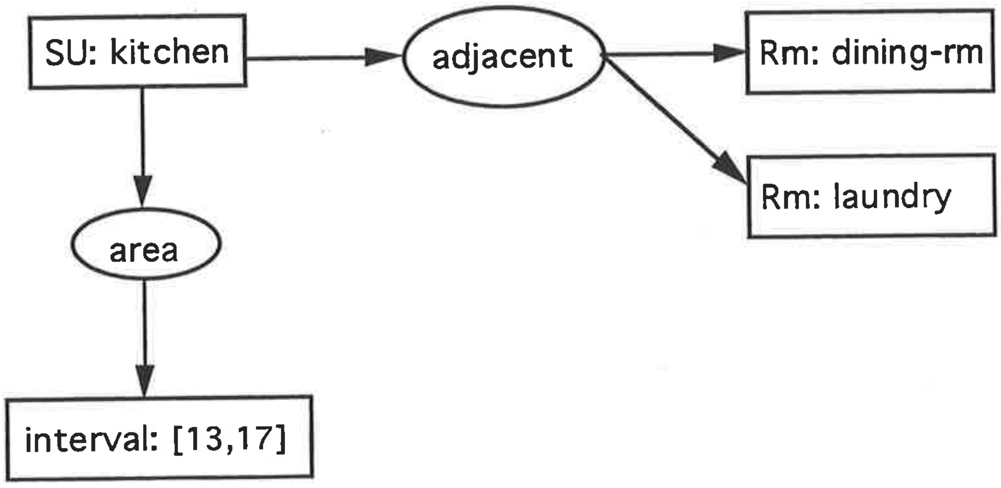


Figure 48. Another design with arity 2 relations.

adjacent\2) then all of the other definitions described in previous chapters are still valid.

Using these new definitions, plus the definition of Unify from Chapter Three, we can now combine Conceptual Graphs of variable arity. Variable arity was implemented in the Unify algorithm by taking all of the arguments of one relation, and attempting to find the corresponding arguments in its comparable relation without regard to the order of the arguments, thus violating one of the rules of Conceptual Graph semantics. After all matches from both CGs have been made, then all remaining arguments are included in the new, unified relation.

The design and architecture domains can benefit from the use of variable-arity relations. Designers often attempt to reuse previous designs, and apply them

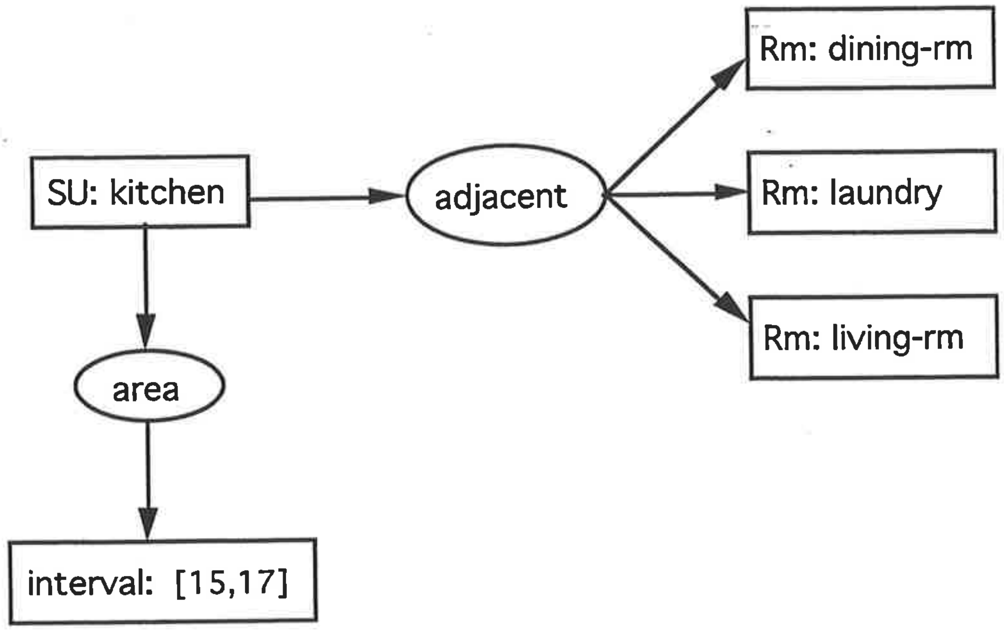


Figure 49. The unification of the two graphs, using variable arity.

to new problems. In the case of designing a building, we want unified designs to reflect all of the knowledge contained in the originals, even if the number of exits (or the rooms which are adjacent, or the exact floor area) is not an exact match. For example, a retrieved design may show the kitchen adjacent to the living room and dining room as shown in Figure 47, while the customer requirements specify that the kitchen must be adjacent to the dining room and laundry as shown in Figure 48. The *adjacent* relation is of arity 2 in these instances. Once unified, however, the new graph of the kitchen must show it as adjacent to the laundry, living room and dining room as shown in Figure 49. This change in arity preserves the knowledge from both of the original graphs, and is in keeping with the designer's intent.

In the natural language domain, a grammar is used to specify that adjectives can precede nouns (in English). However, it is not known in advance how many adjectives will be used in a noun phrase. A variable-arity relation "adjective" would allow for any number of adjectives. Since word order is essential in language understanding, and given that our join operator does not guarantee the order of concepts after a join, a modification to the join operator would be necessary for the natural language domain.

6.3.2 Steps toward Conceptual Graphs as a Programming Language

There have recently been some attempts to define a programming language from (or for) Conceptual Graphs. These include Cao's Fuzzy CG Programs [Cao et al. 1997] as mentioned in Chapter One, Kabbaj's executable Conceptual Graphs [Kabbaj 1999a; Kabbaj 1999b] and Ghosh and Wuwongse's definitions of the Declarative Semantics of CGs [Ghosh and Wuwongse 1995; Ghosh and Wuwongse 1994]. Others have mentioned work toward a programming language, or in software tools for CGs. This is clearly an area which needs to be explored more thoroughly, and more formally.

It would be a fairly straightforward task to define a subsumption ordering on the integers, and use projection or subsumption to increment an integer variable (but see objections to a partial ordering of integers on a lattice in Chapter Two). Once it is possible to define a successor function on the integers, then, using constraint processing, one could provide loop control, variables, etc. This would help to define a very rudimentary programming language, implemented over CGs, and able to use CGs as the main data/knowledge structures.

However, the real effort toward a programming language for CGs needs to first define the semantics behind the language. The Declarative Semantics of Ghosh and Wuwongse are a very good start in this direction. Another approach, however, would be to consider using Axiomatic Semantics for some of the formal semantics definitions. The appeal of Axiomatic Semantics is the formal logical rigor, and its ability to show completeness of an individual operator.

Axiomatic Semantics prescribe, in an abstract way, a minimal set of constraints that any implementation of the subject language must satisfy in its treatment of the various types of construct but say nothing about the details of how this might be achieved. A proof of correctness of a program using Axiomatic Semantics alone is actually only a proof of partial correctness, that is correctness subject to the assumption of termination [Pagan 1981]. In order to verify total correctness, it is necessary also to prove termination. Since a proof of termination for Headed CGs is offered here in Chapter Three, the Axiomatic approach may prove useful in prescribing the constraints of the CG language. Axiomatic Semantics is not useful for state-based semantics, though, so possibly a Denotational Semantics would also need to be defined.

6.3.3 Partial Graphs for Indexing

As was shown in Chapter Four, a partially specified graph can be added to a larger graph by using unification. In a similar manner, and by using the generalization techniques contained in the CG toolbox, we can use small graphs to retrieve designs by simply sketching a part of the desired graph. If the query graph matches any graphs in the knowledge base (or possibly a generalization or a specialization of a graph) the graph from the knowledge base is retrieved for inspection by the user. Practical user interface issues would need to be resolved, but see Mineau and Miranda's approach in [Mineau and Miranda 1998].

6.3.4 Connections with Previous Work

Mineau's work, discussed in the first two chapters, really is more of an attempt to define valid structures in a canon, while the work presented here constrains the values in concepts. A major difference in these approaches is that we rely only on projection and join, while Mineau has implemented actors to check the graphs after an attempt at a join. An interesting research direction might be to combine these two approaches to produce a more comprehensive constraint system. Using intervals to express the topological constraints that Mineau uses might lead to some short cuts in the processing time.

Many of the projects discussed in the first chapters are attempts to either constrain the structure of the CGs, or to specify what types of concepts are valid. One notable exception is the work of Cao et al [Cao and Creasy 1998; Cao et al. 1997] in the implementation of fuzzy values in the concepts. These implementations are useful for specifying boundaries for the values in concepts, but Cao has not pursued constraint processing technology. The possibility of combining the ideas presented in this thesis with Cao's ideas by implementing fuzzy membership functions in the value of a constraint is an intriguing idea which is waiting to be explored.

6.3.5 Ontologies for Knowledge Conjunction

Perhaps most significantly, the work in this thesis does not merely have implications for the area of unification of Conceptual Graphs. One research area which has emerged from this work is the combination of knowledge bases represented as ontologies. Research has already started in the area of combining ontologies from disparate domains [van Zyl and Corbett 2000a; van Zyl and Corbett 2000b]. This new research area will involve finding the common semantics underlying ontological representations of the knowledge of a domain, and linking that knowledge with the knowledge of another ontology in another domain. This work goes beyond mere unification, into understanding the fundamental nature of the combination of knowledge.

References

[Aho et al. 1974] Aho, A. V., J. E. Hopcroft and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Reading, Massachusetts, Addison-Wesley, 1974.

[Aït-Kaci 1986] Aït-Kaci, H. "An Algebraic Semantics Approach to the Effective Resolution of Type Equations." *Theoretical Computer Science* **45**(3): 293-351, 1986.

[Aït-Kaci et al. 1989] Aït-Kaci, H., R. Boyer, P. Lincoln and R. Nasr. "Efficient Implementation of Lattice Operations." *ACM Transactions on Programming Languages and Systems* **11**(1): 115-146, 1989.

[Aït-Kaci and Nasr 1986] Aït-Kaci, H. and R. Nasr. "LOGIN: A Logic Programming Language with Built-in Inheritance." *Journal of Logic Programming* **3**, 1986.

[Aït-Kaci et al. 1992] Aït-Kaci, H., A. Podelski and G. Smolka. *A Feature Constraint System for Logic Programming with Entailment*. Paris, France, Digital Equipment Corporation, 1992.

[Baader and Siekmann 1994] Baader, F. and J. Siekmann. "Unification Theory". *Handbook of Logic in Artificial Intelligence and Logic Programming*, Vol. 2, pgs. 41-126, D. M. Gabbay, C. J. Hogger and J. A. Robinson (eds.). Oxford, Clarendon Press. 1994.

[Bürckert 1991] Bürckert, H.-J. *A Resolution Principle for a Logic with Restricted Quantifiers*. Berlin, Springer-Verlag, 1991. Lecture Notes in Artificial Intelligence #568

[Burrow and Woodbury 1999] Burrow, A. L. and R. Woodbury. "p-Resolution in Design Space Exploration". In *Proc. CAAD Futures*, Atlanta, Georgia, USA, Academic Publishers, August, 1999.

[Cao 1995] Cao, T. H., *Fuzzy Conceptual Graph Programs*, Masters Thesis, Computer Science, Asian Institute of Technology, Bangkok, Thailand, 1995.

[Cao and Creasy 1998] Cao, T. H. and P. N. Creasy. "Fuzzy Order-Sorted Logic Programming in Conceptual Graphs with a Sound and Complete Proof Procedure". In *Proc. Sixth International Conference on Conceptual Structures*, Montpellier, France, Springer Verlag, August, 1998.

[Cao et al. 1997] Cao, T. H., P. N. Creasy and V. Wuwongse. "Fuzzy Unification and Resolution Proof Procedure for Fuzzy Conceptual Graph Programs". In *Proc. Fifth International Conference on Conceptual Structures*, Seattle, Washington, USA, Springer-Verlag, August, 1997.

[Carpenter 1992] Carpenter, B. *The Logic of Typed Feature Structures*. Cambridge, Cambridge University Press, 1992.

[Champesme 1996] Champesme, M. "Opérateurs de Raffinement Idéaux pour les Graphes Conceptuels." *Revue d'Intelligence Artificielle* 10(1): 101-131, 1996.

[Chang and Woodbury 1996] Chang, T.-W. and R. F. Woodbury. "Sufficiency of the SEED Knowledge-Level Representation for Grammatical Design". In *Proc. Australian New Zealand Conference on Intelligent Information Systems*, Adelaide, Australia, IEEE Press, November, 1996.

[Chein and Leclère 1994] Chein, M. and M. Leclère. "A Cooperative Program for the Construction of a Concept Type Lattice". In *Proc. Second International*

Conference on Conceptual Structures, College Park, Maryland, USA, Springer-Verlag, August, 1994.

[Chein and Mugnier 1992] Chein, M. and M.-L. Mugnier. "Conceptual Graphs: Fundamental Notions." *Revue d'Intelligence Artificielle* 6(4): 365-406, 1992.

[Chibout and Vilnat 1998] Chibout, K. and A. Vilnat. "Computational Processing of Verbal Polysemy with Conceptual Structures". In *Proc. Sixth International Conference on Conceptual Structures*, Montpellier, France, Springer-Verlag, August, 1998.

[Chomsky 1980] Chomsky, N. *Rules and Representations*. Oxford, Blackwell, 1980.

[Cleary 1997] Cleary, J. G. "Constructive Negation of Arithmetic Constraints Using Dataflow Graphs." *Constraints* **2**: 131-162, 1997.

[Cogis and Guinaldo 1995] Cogis, O. and O. Guinaldo. "A Linear Descriptor for Conceptual Graphs and a Class for Polynomial Isomorphism Test". In *Proc. Third International Conference on Conceptual Structures*, Santa Cruz, California, USA, Springer-Verlag, August, 1995.

[Cohen 1990] Cohen, J. "Constraint Logic Programming Languages." *Communications of the ACM* **33**(7): 52-68, 1990.

[Cohen 1996] Cohen, J. "Logic Programming and Constraint Logic Programming." *ACM Computing Surveys* **28**(1): 257-259, 1996.

[Colmerauer 1990] Colmerauer, A. "An Introduction to Prolog III." *Communications of the ACM* **33**(7): 69-90, 1990.

[Corbett 1991] Corbett, D. R., *A Conjunction Analysis Technique Applied to an Abductive Framework of Natural Language Processing*, MS Thesis, Department of Computer Science, Wright State University, Dayton, Ohio, USA, 1991.

[Corbett 1999] Corbett, D. R. "A Case for Variable-Arity Relations: Definitions and Domains". In *Proc. Seventh International Conference on Conceptual Structures*, Blacksburg, Virginia, USA, Springer-Verlag, July, 1999.

[Corbett 2000] Corbett, D. R. "A Framework for Conceptual Graph Unification". In *Proc. Eighth International Conference on Conceptual Structures*, Darmstadt, Germany, Shaker Verlag, August, 2000.

[Corbett 2001] Corbett, D. R. "Extending Conceptual Graphs with Unification over Constraints." *Computational Intelligence*(in revision), 2001.

[Corbett and Burrow 1996] Corbett, D. R. and A. L. Burrow. "Knowledge Reuse in SEED Exploiting Conceptual Graphs". In *Proc. Fourth International Conference on Conceptual Structures*, Sydney, NSW, Australia, UNSW Press, August, 1996.

[Corbett and Woodbury 1999] Corbett, D. R. and R. F. Woodbury. "Unification over Constraints in Conceptual Graphs". In *Proc. Seventh International Conference on Conceptual Structures*, Blacksburg, Virginia, USA, Springer-Verlag, July, 1999.

[Dasigi 1988] Dasigi, V., *Word Sense Disambiguation in Descriptive Text Interpretation: A Dual-Route Parsimonious Covering Model*, PhD Dissertation, Department of Computer Science, University of Maryland, College Park, Maryland, USA, 1988.

[Dasigi 1991] Dasigi, V. "Parsing = Parsimonious Covering? (Abduction in Logical form Generation)". In *Proc. Twelfth International Joint Conference on Artificial Intelligence*, Sydney, Australia, August, 1991.

[Davey and Priestley 1990] Davey, B. A. and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge, Cambridge University Press, 1990.

[Dibie et al. 1998] Dibie, J., O. Haemmerlé and S. Loiseau. "A Semantic Validation of Conceptual Graphs". In *Proc. Sixth International Conference on Conceptual Structures*, Montpellier, France, Springer-Verlag, August, 1998.

[Eisinger and Ohlbach 1993] Eisinger, N. and H. J. Ohlbach. "Deduction Systems Based on Resolution". *Handbook of Logic in Artificial Intelligence and Logic Programming*, Vol. 1, pgs. 183-271, D. M. Gabbay, C. J. Hogger and J. A. Robinson (eds.). Oxford, Clarendon Press. 1993.

[Ellis 1995] Ellis, G., *Managing Complex Objects*, PhD Thesis, Department of Computer Science, University of Queensland, Brisbane, Australia, 1995.

[Esch and Levinson 1995] Esch, J. and R. Levinson. "An Implementation Model for Contexts and Negation in Conceptual Graphs". In *Proc. Third International Conference on Conceptual Structures*, Santa Cruz, California, USA, Springer-Verlag, August, 1995.

[Flemming and Woodbury 1995] Flemming, U. and R. F. Woodbury. "Software Environment to Support Early Phases in Building Design (SEED) Overview." *Architectural Engineering* **1**(1), 1995.

[Frost and Dechter 1994] Frost, D. and R. Dechter. "In Search of the Best Constraint Satisfaction Search". In *Proc. (American) National Conference on Artificial Intelligence*, Seattle, Wash, USA, August, 1994.

[Gerbé 1997] Gerbé, O. "Conceptual Graphs for Corporate Knowledge Repositories". In *Proc. Fifth International Conference on Conceptual Structures*, Seattle, Washington, USA, Spring-Verlag, August, 1997.

[Gerbé et al. 1998] Gerbé, O., R. K. Keller and G. W. Mineau. "Conceptual Graphs for Representing Business Processes in Corporate Memories". In *Proc. Sixth International Conference on Conceptual Structures*, Montpellier, France, Springer-Verlag, August, 1998.

[Ghosh and Wuwongse 1994] Ghosh, B. C. and V. Wuwongse. "Inference Systems for Conceptual Graph Programs". In *Proc. Second International Conference on Conceptual Structures*, College Park, Maryland, USA, Springer-Verlag, August, 1994.

[Ghosh and Wuwongse 1995] Ghosh, B. C. and V. Wuwongse. "Conceptual Graph Programs and Their Declarative Semantics." *IEICE Transaction* **E78-D**(9): 1208-1217, 1995.

[Gini and Rogialli 1994] Gini, G. C. and C. Rogialli. "CONSTRICTOR: A Constraint-Based Language." *Computer Systems Science and Engineering* **9**(4): 255-261, 1994.

[Godin et al. 1995] Godin, R., G. Mineau, R. Missaoui and H. Mili. "Méthodes de Classification Conceptuelle Basées sur les Treillis de Galois et Applications." *Revue d'Intelligence Artificielle* **9**(2): 105-137, 1995.

[Heisserman 1991] Heisserman, J. A., *Generative Geometric Design and Boundary Solid Grammars*, PhD Thesis, Department of Architecture, Carnegie Mellon University, Pittsburgh, Penn, USA, 1991.

[Heisserman 1995] Heisserman, J. A. "Generative Geometric Design." *IEEE Computer Graphics and Applications* **14**(2): 37-45, 1995.

[ILOG 1996] ILOG. *ILOG Solver Reference Manual*. Mountain View, California, USA, 1996.

[Jaffar and Lassez 1987] Jaffar, J. and J.-L. Lassez. "Constraint Logic Programming". In *Proc. ACM Symposium on Principles of Programming Languages*, 1987.

[Jaffar and Maher 1994] Jaffar, J. and M. J. Maher. "Constraint Logic Programming: A Survey." *Journal of Logic Programming*(July/August): 503-581, 1994.

[Kabbaj 1999a] Kabbaj, A. "Synergy as an Hybrid Object-Oriented Conceptual Graph Language". In *Proc. Seventh International Conference on Conceptual Structures*, Blacksburg, Virginia, USA, Springer Verlag, August, 1999.

[Kabbaj 1999b] Kabbaj, A. "Synergy: A Conceptual Graph Activation-Based Language". In *Proc. Seventh International Conference on Conceptual Structures*, Blacksburg, Virginia, USA, Springer Verlag, August, 1999.

[Kirchner et al. 1990] Kirchner, C., H. Kirchner and M. Rusinowitch. "Deduction with Symbolic Constraints." *Revue d'Intelligence Artificielle* **4**(3): 9-52, 1990.

[Knight 1989] Knight, K. "Unification: A Multidisciplinary Survey." *ACM Computing Surveys* **21**(1): 93-124, 1989.

[Kocura 1996] Kocura, P. "Conceptual Graphs and Semantic Constraints". In *Proc. Fourth International Conference on Conceptual Structures*, Sydney, NSW, Australia, University of NSW Press, August, 1996.

[Kowalski 1979] Kowalski, R. A. "Algorithm = Logic + Control." *Communications of the ACM* **22**(7): 424-436, 1979.

[Lassez et al. 1988] Lassez, J. L., M. J. Maher and K. Marriot. "Unification Revisited". *Foundations of Deductive Databases and Logic Programming*, Vol. , pgs. 587-625, J. Minker , Morgan Kaufman. 1988.

[Leclère 1996] Leclère, M. "C-CHiC: Construction Coopérative de Hiérarchies de Catégories." *Revue d'Intelligence Artificielle* **10**(1): 57-100, 1996.

[Leclère 1997] Leclère, M. "Reasoning with Type Definitions". In *Proc. Fifth International Conference on Conceptual Structures*, Seattle, Washington, USA, Springer-Verlag, August, 1997.

[Lehmann 1992] Lehmann, F. "Semantic Networks." *Computers & Mathematics with Applications* **23**(2-5): 1-50, 1992.

[Mackworth 1992] Mackworth, A. K. "The Logic of Constraint Satisfaction." *Artificial Intelligence* 58(1-2): 3-20, 1992.

[Mann 1998] Mann, G. A. "Procedural Renunciation". In *Proc. Sixth International Conference on Conceptual Structures*, Montpellier, France, Springer-Verlag, August, 1998.

[Mineau 1998] Mineau, G. "From Actors to Processes: The Representation of Dynamic Knowledge Using Conceptual Graphs". In *Proc. Sixth International Conference on Conceptual Structures*, Montpellier, France, Springer-Verlag, August, 1998.

[Mineau 1999] Mineau, G. "Constraints on Processes: Essential Elements for the Validation and Execution of Processes". In *Proc. Seventh International Conference on Conceptual Structures*, Blacksburg, Virginia, USA, Springer-Verlag, August, 1999.

[Mineau and Miranda 1998] Mineau, G. W. and C. Miranda. *Computer-Aided Design and Artificial Intelligence: Exploration in Architectural Plan Reuse*. Unpublished manuscript, 1998.

[Mineau and Missaoui 1997] Mineau, G. W. and R. Missaoui. "The Representation of Semantic Constraints in Conceptual Graph Systems". In *Proc. Fifth International Conference on Conceptual Structures*, Seattle, Washington, USA, Springer-Verlag, August, 1997.

[Mitchard 1998] Mitchard, H., *Cognitive Model of an Operations Officer*, Honours Thesis, Computer and Information Science, University of South Australia, Adelaide, South Australia, 1998.

[Mitchard et al. 2000] Mitchard, H., J. Winkles and D. R. Corbett. "Development and Evaluation of a Cognitive Model of an Air Defence Operations Officer". In *Proc. Fifth Biennial Conference of the Australasian Cognitive Science Society*, Adelaide, South Australia, May, 2000.

[Moulin 1998] Moulin, B. "A Logical Framework for Modeling a Discourse from the Point of View of the Agents Involved in It". In *Proc. Sixth International Conference on Conceptual Structures*, Montpellier, France, Springer-Verlag, August, 1998.

[Mugnier and Chein 1996] Mugnier, M.-L. and M. Chein. "Représenter des Connaissances et Raisonner avec des Graphes." *Revue d'Intelligence Artificielle* **10**(6): 7-56, 1996.

[Müller 1997] Müller, T., *Conceptual Graphs as Terms: Prospects for Resolution Theorem Proving*, Masters Thesis, Department of Computer Science, Vrije Universiteit Amsterdam, Amsterdam, Netherlands, 1997.

[Nadel 1990] Nadel, B. A. "Representation Selection for Constraint Satisfaction: A Case Study Using n-Queens." *IEEE Expert*: 16-23, 1990.

[Nicolov et al. 1995] Nicolov, N., C. Mellish and G. Ritchie. "Sentence Generation from Conceptual Structures". In *Proc. Third International Conference on Conceptual Structures*, Santa Cruz, California, USA, Springer-Verlag, August, 1995.

[Older 1997] Older, W. J. "Involution Narrowing Algebra." *Constraints* **2**: 113-130, 1997.

[Pagan 1981] Pagan, F. G. *Formal Specification of Programming Languages*. Englewood Cliffs, New Jersey, USA, Prentice-Hall, 1981.

[PrologIA 1997] PrologIA. *Prolog IV Manual*. Marseilles, France, Prolog IV is a registered trademark of PrologIA., 1997.

[Reggia et al. 1983] Reggia, J., D. Nau and P. Wang. "Diagnostic Expert Systems Based on a Set Covering Model." *International Journal of Man-Machine Studies* **19**: 437-460, 1983.

[Reynolds 1970] Reynolds, J. C. "Transformational Systems and the Algebraic Structure of Atomic Formulas." *Machine Intelligence* **5**, 1970.

[Ribière 1998] Ribière, M. "Using Viewpoints and CG for the Representation of Dynamic Knowledge Using Conceptual Graphs". In *Proc. Sixth International Conference on Conceptual Structures*, Montpellier, France, Springer-Verlag, August, 1998.

[Robinson 1965] Robinson, J. A. "A Machine-Oriented Logic Based on the Resolution Principle." *Journal of the ACM* **12**: 23-41, 1965.

[Sowa 1999] Sowa, J. "Conceptual Graphs: Draft Proposed American National Standard". In *Proc. Seventh International Conference on Conceptual Structures*, Blacksburg, Virginia, USA, Springer-Verlag, July, 1999.

[Sowa 1984] Sowa, J. F. *Conceptual Structures: Information Processing in Mind and Machine*. Reading, Mass, Addison-Wesley, 1984.

[Sowa 1992] Sowa, J. F. "Conceptual Graphs Summary". *Conceptual Structures: Current Research and Practice*, Vol. , pgs. , (eds.). Chichester, UK, Ellis Horwood. 1992.

[Tepfenhart 1998] Tepfenhart, W. M. "Ontologies and Conceptual Structures". In *Proc. Sixth International Conference on Conceptual Structures*, Montpellier, France, Springer-Verlag, August, 1998.

[Van Hentenryck 1989] Van Hentenryck, P. *Constraint Satisfaction in Logic Programming*. Cambridge, Massachusetts, USA, MIT Press, 1989.

[Van Hentenryck et al. 1997] Van Hentenryck, P., L. Michel and Y. Deville. *Numerica*. Cambridge, Massachusetts, USA, MIT Press, 1997.

[van Zyl and Corbett 2000a] van Zyl, J. D. and D. R. Corbett. "A Framework for Comparing Methods for Using or Reusing Multiple Ontologies in an Application". In *Proc. Eighth International Conference on Conceptual Structures*, Darmstadt, Germany, Shaker Verlag, August, 2000.

[van Zyl and Corbett 2000b] van Zyl, J. D. and D. R. Corbett. "A Framework for Comparing the Use of a Linguistic Ontology in an Application". In *Proc.*

European Conference on Artificial Intelligence Workshop on Applications of Ontologies and Problem Solving Methods, Berlin, Germany, August, 2000.

[Wermelinger 1995] Wermelinger, M. "Conceptual Graphs and First-Order Logic". In *Proc. Third International Conference on Conceptual Structures,* Santa Cruz, California, USA, Springer-Verlag,

[Wermelinger 1997] Wermelinger, M. "A Different Perspective on Canonicity". In *Proc. Fifth International Conference on Conceptual Structures,* Seattle, Washington, USA, Springer-Verlag, August, 1997.

[Wermelinger and Lopes 1994] Wermelinger, M. and J. G. Lopes. "Basic Conceptual Structures Theory". In *Proc. Second International Conference on Conceptual Structures,* Maryland, Springer-Verlag, August, 1994.

[Wille 1992] Wille, R. "Concept Lattices and Conceptual Knowledge Systems." *Computers and Mathematics with Applications* **23**(6-9): 492-515, 1992.

[Wille 1996a] Wille, R. "Conceptual Structures of Multicontexts". In *Proc. Fourth International Conference on Conceptual Structures,* Sydney, Australia, Springer-Verlag, August, 1996.

[Wille 1996b] Wille, R. *Short Introduction to Formal Concept Analysis.* Unpublished manuscript, 1996b.

[Wille 1997] Wille, R. "Conceptual Graphs and Formal Concept Analysis". In *Proc. Fifth International Conference on Conceptual Structures,* Seattle, Washington, USA, Springer-Verlag, August, 1997.

[Willems 1995] Willems, M. "Projection and Unification for Conceptual Graphs". In *Proc. Third International Conference on Conceptual Structures,* Santa Cruz, California, USA, Springer-Verlag, August, 1995.

[Wing et al. 1998] Wing, H., R. M. Colomb and G. W. Mineau. "Using CG Formal Contexts to Support Business System Interoperations". In *Proc. Sixth*

International Conference on Conceptual Structures, Montpellier, France, Springer-Verlag, August, 1998.

[Woodbury et al. 2000] Woodbury, R., S. Datta and A. L. Burrow. "Erasure in Design Space Exploration". In *Proc. Artificial Intelligence in Design*, Worcester, Massachusetts, USA, June, 2000.

[Woodbury et al. 1999] Woodbury, R. F., A. L. Burrow, S. Datta and T. W. Chang. "Typed Feature Structures in Design Space Exploration." *AIEDAM* **13**(4): 287-302, 1999.

[Wuwongse and Cao 1996] Wuwongse, V. and T. H. Cao. "Towards Fuzzy Conceptual Graph Programs". In *Proc. Fourth International Conference on Conceptual Structures*, Sydney, Australia, Springer-Verlag, August, 1996.