APPROACHES TO PROCEDURAL ADEQUACY IN LOGIC PROGRAMMING
USING CONNECTION GRAPHS

By
THEODORE WARREN BERNELOT MOENS
B.Sc., York University, 1982

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in
THE FACULTY OF GRADUATE STUDIES
(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
June 1987
© Ted Moens, 1987
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date August 19, 1987
Abstract

Kowalski's connection graph method provides a representation for logic programs which allows for the incorporation of better procedural control techniques than standard logic programming languages. A proposed search strategy for visual recognition which combines top-down and bottom-up techniques has been incorporated in a connection graph implementation.

The connection graph representation also allows for the natural incorporation of constraint satisfaction techniques in logic programming. Kowalski's approach to incorporating constraint satisfaction techniques in connection graphs is examined in detail. It is shown that his approach is not efficient enough to be used as a general preprocessing algorithm but that a modified version may be of use.

Increased control of search and the incorporation of consistency techniques increase the procedural adequacy of logic programs for representing knowledge without compromising the descriptive capacity of the form.
# Table of Contents

Abstract ii

Table of Contents iii

List of Figures v

Acknowledgment vi

Chapter 1: Introduction 1

Chapter 2: Literature Review 3

2.1 Connection Graphs 3
   2.1.1 The Connection Graph Proof Procedure 5
   2.1.2 Compatibility of Substitutions 9

2.2 Search in Knowledge Representations 10
   2.2.1 Representational Adequacy 10
   2.2.2 A Fundamental Tradeoff 11
      2.2.2.1 Extending the Expressiveness of Logic Programs 14
      2.2.2.2 Extending the Procedural Efficiency of Logic Programs 15
   2.2.3 Flexible Search 19
   2.2.4 Constraint Satisfaction 22
      2.2.4.1 Freuder's k-consistency Algorithm 30

2.3 Connection Graph Research 32
   2.3.1 Search in Connection Graphs 34
      2.3.1.1 Graph Reductions 34
      2.3.1.2 Planning Refutations 38
      2.3.1.3 Heuristic Search 42
   2.3.2 Constraint Satisfaction in Connection Graphs 43

2.4 Implementation Philosophy 47
   2.4.1 Object Oriented Programming 47
TABLE OF CONTENTS

Chapter 3  51

3.1 Flexible Search in Connection Graphs  51
   3.1.1 Matching  52
   3.1.2 Completion  55
   3.1.3 Expectation  55
   3.1.4 Summary  59

3.2 Constraint Satisfaction in Connection Graphs  59
   3.2.1 An Example  61
   3.2.2 The M-FAlgorithm  64
   3.2.3 Connection Graphs and k-consistency  65
   3.2.4 Revising the Link Filtering Algorithm  74

3.3 Implementation  76
   3.3.1 Object Oriented Connection Graphs  76
   3.3.2 Object Oriented Search  80
      3.3.2.1 Backtracking  80
      3.3.2.1 Local Guidance  82
      3.3.2.3 Hybrid Search  83
   3.3.3 Implementation Details  85
      3.3.3.1 Structure Sharing  85
      3.3.3.2 Links and Clauses  88
      3.3.3.3 Object Oriented Language Implementation  88

Chapter 4: Discussion  92
   4.1 Flexible Search in Connection Graphs  92
   4.2 Approximation Techniques in Connection Graphs  93
      4.2.1 Extending Constraint Satisfaction in Connection Graphs  95
   4.3 Object Oriented Programming  97
   4.4 Conclusion  98
   4.5 Future Work  99

Appendix: Connection Graph Implementation  101

Bibliography  119
List of Figures

Figure 1  A Simple Connection Graph  
Figure 2  Connection Graph Proof Procedure  
Figure 3  Graphic Representation of a Constraint Satisfaction Problem I  
Figure 4  Graphic Representation of a Constraint Satisfaction Problem II  
Figure 5  Freuder's k-consistency Algorithm  
Figure 6  Link Selection Heuristics I  
Figure 7  Link Selection Heuristics II  
Figure 8  Estimating Refutation Lengths  
Figure 9  Example Terminator Configurations  
Figure 10  Incompatible Links  
Figure 11  Kowalski's Link Filtering Algorithm for Connection Graphs  
Figure 12  Recursive Recognition Cycle  
Figure 13  Constraint Satisfaction in Connection Graphs I  
Figure 14  Constraint Satisfaction in Connection Graphs II  
Figure 15  The M-F algorithm
Acknowledgment

I am grateful to William Havens for important discussions about my thesis work. I am also indebted Nancy Hamilton for her support as well as her help in the editing of this manuscript.

This work was supported by a graduate scholarship from the Natural Sciences and Engineering Research Council.
Chapter 1: Introduction

Logic programing systems in general and Prolog in particular have been criticized as knowledge representation formalisms because of their lack of procedural adequacy. Their rigid control structures tend to preclude the use of flexible search strategies for processing the knowledge base. The connection graph proof procedure introduced by Kowalski [1975] provides a way of incorporating various flexible search methodologies in a logic programming system. Connection graph search can easily mimic any standard resolution refinements but it is not restricted to either goal driven or data driven strategies. Search in connection graphs can take advantage of some powerful heuristics based on the structure of the problem and its search space.

A major focus of this thesis is on the use of local control of search in a simple connection graph implementation. As an example, an hybrid search strategy from the computer vision domain is incorporated in a connection graph implementation.

Related to the idea of procedural adequacy of knowledge representation formalisms are the so called consistency or constraint satisfaction algorithms. Consistency algorithms are used to preprocess certain types of problems to remove local inconsistencies before search commences. Their main benefit is to
CHAPTER 1: INTRODUCTION

reduce the size of the solution space before it is searched. It has been suggested that the use of consistency algorithms could be applied to logic programming. Kowalski describes a filtering algorithm which makes use of a constraint satisfaction approach to preprocessing logic programming problems, but he claims that it is subsumed by the normal connection graph proof procedure.

Thus, the second major focus of this thesis is to consider the use of consistency techniques in connection graphs. Kowalski's filtering algorithm is formally analysed in terms of well known consistency techniques and it is shown that the normal operation of the link filter is too extensive to be of use as a preprocessing operation. In spite of this, consistency techniques may still be of some value if used to a lesser degree.

Finally, the implementation of connection graphs and search therein mesh well with the the concepts of object oriented programming. This notion is pursued in the discussion of the implementation aspects of this thesis. A simple object oriented connection graph implementation which incorporates a flexible search strategy is included in the appendix.
Chapter 2: Literature Review

In this chapter, a review of relevant literature is presented in order to develop the motivation for this thesis. The connection graph proof procedure is reviewed in section 2.1. A review of search mechanisms in some existing knowledge representation systems is presented in section 2.2, followed by a discussion of search techniques in existing connection graph systems in section 2.3. Some of the concepts and philosophy of object-oriented programming which are pertinent to the implementation of search procedures presented in this thesis are reviewed in section 2.4. The main objectives of this thesis will be highlighted throughout this chapter.

2.1 Connection Graphs

A connection graph is a data structure for representing a set of logic clauses and indicating possible resolutions between clauses [Chang 1979]. An example of a set of clauses and the corresponding connection graph is given in Figure 1. Every literal in a clause corresponds to a node in the graph. Two nodes are linked if they can be made complementary by applying some substitution (after appropriate renaming of variables). Each link is labeled by the most general unifier (mgu) of the two literals it connects. The nodes formed from the literals of each clause form a separate partition in the graph.
CHAPTER 2: LITERATURE REVIEW

Figure 1: A set of clauses a) and the corresponding connection graph b). In this illustration as in the rest of the thesis, literals of a connection graph will be represented by ovals and clauses will be represented by rectangles around a group of literals. The links between literals will be represented by lines joining the ovals.

In order to use connection graphs in an inference procedure, operations on this data structure must be introduced. Kowalski's [1975] proof procedure involves selecting one of the links in the graph, resolving the two clauses it connects, forming the resolvent and adding it to the graph. The selected link is then removed and a new link is selected for processing. The procedure terminates successfully when the empty clause is derived and unsuccessfully when there is no link left to chose and the empty clause has not been derived. The details of the proof procedure are given below as well as some variations in other connection graph implementations.
2.1.1 The Connection Graph Proof Procedure

The connection graph proof procedure is reproduced in Figure 2 from Kowalski [1975]. The first two steps of the algorithm deal with the creation of a connection graph from a set of clauses. Each clause is added to the connection graph† and every pair of potentially complementary literals in the clause set are linked in the graph. The third step describes how to process a graph. It is repeated until the termination condition is met. The processing of the graph involves selecting a link and resolving the two clauses joined by that link. The link is then removed from the connection graph and the resolvent is connected into the graph. In step 3c, the final phase of processing a connection graph allows for the removal of graph structures such as tautologies, which can never participate in a refutation.

The notion of compatible unifiers or substitutions as used in step 3bi) was not provided by Kowalski but has since been described by several authors. Intuitively, two unifiers are compatible if they can themselves be unified. A more rigorous definition of compatibility of unifiers is given below.

†This thesis does not consider factoring.
CHAPTER 2: LITERATURE REVIEW

1) In order to determine that a set of clauses $S$ is unsatisfiable:
   a) Construct and
   b) Process the initial connection graph for $S$.
2) In order to construct the initial connection graph for $S$:
   a) Include in the graph all clauses in $S$,
   b) For every pair of potentially complementary literals in distinct clauses in the graph,
      insert a link connecting the literals.
   c) Label each link by the most general unifying substitution which makes the literals
      complementary.
3) In order to process the connection graph:
   If the graph contains the empty clause, terminate successfully.
   If the graph does not contain the empty clause:
      a) Select a link in the graph (and call the associated mgu $\theta$), generate the associated
         resolvent, add the resolvent to the graph and delete the selected link.
      b) Add links connecting literals in the newly generated resolvent to other literals in the
         graph
         i) If [literal] $L\theta$ in the resolvent descends from $L$ in a parent clause,
            If a link labeled by $\alpha$ connects $L$ to $K$, and
            If $\alpha$ and $\theta$ are compatible, then
            Add to the graph a link connecting $L\theta$ and $K$ labeled by the substitution $\alpha \ast \theta$ which is
            the most general unifier of $L\theta$ and $K$.
         ii) if $L\theta$ in a new clause is potentially complementary to a literal occurrence $K$ in a
             parent clause, then
             add a link connecting $L\theta$ and $K$, labeled by the most general unifier of $L\theta$ and $K$.
      c) Delete a clause and all links connected to its literals
         i) If the clause is a tautology or,
         ii) If the clause contains a pure literal (one not connected by a link to any other literal
             in the graph.
      d) Process the new connection graph.

Figure 2: The connection graph proof procedure from Kowalski [1975].

The connection graph proof procedure can simulate several other deduction
systems. This power stems from the non-determinism inherent in step 3a) of the
proof procedure. If, for example, the link selection is governed by a strategy
which requires one end of the link to be attached to a unit clause, then the system
will simulate a unit preference resolution system. Thus, the connection graph
proof procedure can use any of several standard resolution strategies or even
combine them during the search for a proof [Kowalski 1975].

Search strategy in connection graphs is not limited to standard resolution
strategies, however. Search strategy is simply determined by the order in which
links are selected for processing. This allows the incorporation of flexible
strategies which use heuristics, graph structure or meta-knowledge for
guidance. This thesis focuses on search in connection graphs. Section 2.2
provides some background on the search problem in knowledge representation
systems and section 2.3.1 deals with search techniques in existing and proposed
connection graph implementations.

Another feature of connection graphs which seems superior to other
resolution based theorem proving systems is in the application of preprocessing
algorithms. Preprocessing algorithms are used to transform one set of clauses
into another set which is in some way simpler than the original. This could
mean reducing the number of clauses in the set or more fully instantiating those
clauses. In general, preprocessing algorithms are used to supplement a proof
procedure. Connection graphs however, can incorporate this type of operation
directly so that it is applicable at all stages of the proof procedure rather than as
an initial operation on the clause set [Kowalski 1975]. Step 3c) of the proof
procedure is exactly such an operation. If a literal is pure (not linked to any other
literal in the graph), it can never be resolved away since it is known not to be
potentially complementary to any other literal in the graph. Therefore, any
clause containing a pure literal can never become part of a refutation and thus
can be removed from the graph along with all links attached to that clause. This
rule can often be used to remove one or both parents of a resolvent or even the
resolvent itself, thus removing some portion of the search space which would not be eliminated in other systems.

By using some of the preprocessing operations as integral parts of the proof procedure, the connection graph method can not only simulate other deduction systems but can even remove some of the most glaring inefficiencies in their operation. SL-resolution [Kowalski 1971] for example, has the problem that it can generate contradictory unit clauses on different branches of a search tree but since a clause can only be resolved against an ancestor or an axiom, the contradiction can not be found. In the connection graph simulation of SL-resolution, this deficiency is overcome by the incorporation of a preprocessing step which will give priority to the removal of any link which can directly lead to a refutation [Kowalski 1975].

In resolution proof procedures, an excessive amount of time can be spent searching the database for a clause to resolve against a selected literal. The search is generally limited to the input clause set and can be improved by indexing the input clauses by predicate name. The situation can be improved still further if the indexing is extended to include the structure of the terms in the arguments of the literals. The cost of improving the search for potential resolving clauses is the preprocessing involved in computing the indexing. In connection graphs, the idea of indexing is taken to its extreme in that each literal is explicitly linked to every other literal which it can be resolved against. Thus, there is no search required to find a resolving clause, but preprocessing is maximal since unification must be attempted between each pair of literals in the input set. At first this may seem expensive, but if essentially the same set of axioms is to be used to solve several problems, as is the case in many logic programming
systems, then the cost is not unreasonable. Each new resolvent must also be connected to the existing graph, but this is an efficient operation since all links to the resolvent are inherited directly from the links present in the parents of the resolvent. Research on the connection graph proof procedure and search in connection graphs will be reviewed in section 2.3.

### 2.1.2 Compatibility of Substitutions

Each link in the connection graph is labeled with the *most general unifier* (mgu) of the two literals it connects. Since an mgu is the most general substitution which makes the two literals complementary, the terms compatible substitutions/ mgus/links are used interchangeably. Several researchers [Bruynooghe 1976], [Chang 1979], [van Vaalen 1975] have filled the gap left by Kowalski and provided definitions of *compatible substitutions*. The definition presented here is from [Chang 1979].

Let \( \theta_1 = \{ t_1 /v_1, ..., t_n /v_n \} \), \( \theta_2 = \{ t_1 /v_1, ..., t_m /v_m \} \)

be substitutions. From \( \theta_1 \) and \( \theta_2 \) we define two expressions:

\[
E_1 = (v_1, ..., v_n, v_2, ..., v_m), \quad E_2 = (t_1, ..., t_n, t_2, ..., t_m)
\]

Then \( \theta_1 \) and \( \theta_2 \) are said to be *compatible* iff \( E_1 \) and \( E_2 \) are unifiable.

Examples: Consider \( \theta_1 = \{ a /x \} \) and \( \theta_2 = \{ b /x \} \). The \( E_1 = (x, x) \) and \( E_2 = (a, b) \). Since \( E_1 \) and \( E_2 \) are not unifiable, \( \theta_1 \) and \( \theta_2 \) are not compatible. Consider \( \theta_1 = \{ f(g(u))/x, f(v)/y \} \) and \( \theta_2 = \{ y /x, g(u)/v \} \). The \( E_1 = (x, y, x, v) \) and \( E_2 = (f(g(u)), f(v), y, \)

---

\(^{1}\)most general unifiers are defined in the usual way. See [Lloyd 1984].
CHAPTER 2: LITERATURE REVIEW

g(u)). Since $E_1$ and $E_2$ are unifiable (with unifier $= \{ f(g(u)/x, f(g(u))/y, g(u)/v) \}$, $\theta_1$ and $\theta_2$ are compatible.

2.2 Search in Knowledge Representations

First order logic (FOL), or some well defined subset thereof, has been suggested for use as a formalism for representing knowledge [Levesque 1985], [Moore 1982]. In this section some criteria for an adequate knowledge representation (KR) formalism are introduced and approaches to achieving representational adequacy in logic based representation systems are discussed.

2.2.1 Representational Adequacy

Mackworth [1987] developed criteria for evaluating the adequacy of a visual KR system. There are two main categories under which these criteria fall. The descriptive adequacy of a system is the extent to which the representation can describe situations in the real world. Mackworth gave eleven properties which a descriptively adequate KR must have. They include the ability to represent a possibly unbounded number of objects using a finite knowledge base, the ability to represent complex objects through the composition of other objects, and the ability to represent specializations of object classes. In contrast, procedural adequacy is the ability of the system to use and acquire knowledge. A procedurally adequate system must be able to retrieve the facts that are explicit in the knowledge base (KB) as well as those which are implied by that knowledge in a sound and complete fashion. It must be able to use the knowledge in a flexible manner, acquire new knowledge and be efficient in its use and acquisition of knowledge.
CHAPTER 2: LITERATURE REVIEW

Although the descriptive capacity of a logic based representation system is not altered by the use of the connection graph representation, the same is not the case for procedural efficiency. Thus the focus of this section is on the procedural aspects of logic based representations rather than the descriptive aspects.

### 2.2.2 A Fundamental Trade-off

Levesque [1985] pointed out that there is a fundamental trade-off between descriptive and procedural adequacy in any representational system. Full FOL is a particularly descriptive KR formalism but since it is undecideable, it is not procedurally adequate under Mackworth's criteria. This is not a fault of the representation, since any formalism with the descriptive capacity of FOL will necessarily be undecideable.

Levesque suggested that if a logic based representation system is to be able to calculate what is implicit in a KB, the system must either restrict the requirements of completeness and consistency or somehow restrict the descriptive power of the system until that service can be provided. The completeness requirement could be reduced by putting a time bound on the computations performed and giving an indefinite answer when none has been found within that time bound. While this is infinitely preferable to an inconsistent system, the resulting KR system can no longer be said to calculating what is implicit in the KB and the truth theory of the resulting logic becomes far more complicated than that of full FOL.

The alternative is to restrict the descriptive capacity of the system until it is computationally tractable to calculate the information implicit in a KB. Levesque
described several such representation systems. For each system he described what portion of FOL can be represented and what the computational advantages of the system are. Levesque concludes that since no system can represent all of FOL and be able to infer all facts implicit in a KB, it is worthwhile to devise KR systems which are ‘only’ capable of representing some subset of FOL provided that the representation allows for some efficient (and effective) form of inference and provided that the FOL subset is useful for KR. For any such system however, the representational restriction should be clearly specified as well as the advantages of inferencing within that system.

In this light, some of the representational restrictions and procedural advantages of logic based systems are reviewed. In the database form [Levesque 1985], the KB consists of atomic sentences (no negation, disjunction or existential quantification). This means that the KB contains only positive facts about the relationships between specific objects. The database form also assumes a version of the closed world assumption [Reiter 1978]: the only instances of the relations are the ones stated in the KB. The descriptive capacity of such a system is severely restricted. In the database form it is not possible to represent any degree of uncertainty. For example, it is not possible to state that there is a number greater than seven without stating what that number is. Nor is it possible to represent classes of objects or specializations in the database form. Representation takes the form of analogy - for every object in the world there is an analogous object in the database. However, because there is no incompleteness in the knowledge, “inference reduces to calculation” in the database form of KR [Levesque 1985].

The second restricted form of FOL which Levesque considers is the logic program form. The logic program form is a generalization of the database form.
CHAPTER 2: LITERATURE REVIEW

It is embodied in programs of languages such as Planner [Hewitt 1971] and Prolog [Clocksin 1981]. The logic program form requires that all sentences in the KB have the form:

$$\forall x_1 \ldots x_n [P_1 \land \ldots \land P_m \supset P_{m+1}]$$

where $m \geq 0$ and each $P_i$ is atomic.

The conjunction on the left side is called the *body* of the sentence and $P_{m+1}$ is called the *head*. Sentences of this form are called *Horn clauses* [Lloyd 1984] and hence the form is also known as the *Horn clause form*. If $m = 0$ and all arguments to the predicates are constant, the database form results [Levesque 1985]. The Horn clause form still lacks much of the descriptive capacity of full FOL. In particular, negations ($\neg P$), disjunctions ($P \lor Q$), or existentially quantified sentences ($\exists x P(x)$) are not permitted. However, it is now possible to define relations between objects without stating all the elements of that relation. It is also possible to define classes of objects without enumerating their members.

The logic program form is more descriptive than the data base form, but since not all facts about the world are stated explicitly, inference may be required to retrieve knowledge which is implicit in the KB. While there exist many sound and complete proof procedures for Horn clauses, determining if a ground atomic sentence is implied by a collection of Horn clauses is still undecidable. Thus the use of the logic program form does not provide a fundamental computational advantage over the use of full FOL although computation is generally much more efficient in the logic program form [Voda 1986]. Several implementations of logic program systems can be viewed as attempts to provide a greater degree of procedural or representational adequacy to the logic program form. Some of these are reviewed below.
2.2.2.1 Extending the Expressiveness of Logic Programs

The implementation of negation as failure in Horn Clauses is an attempt to increase the descriptive capacity of the form without increasing the complexity of inference within it. In systems which use negation as failure, negative literals may be included in the body of rules in the KB as well as in queries to the KB. However, negation is not treated as negation in full FOL. Instead, a negative goal (¬P) succeeds when the corresponding positive goal (P) fails. In general this is not satisfactory for a number of reasons. First, the logic no longer satisfies the extension property [Reiter 1978] since a consistent knowledge base may be used to infer ¬P but with the addition of some consistent facts and rules may now infer P. Second, in many Prolog implementations negation as failure is unsound, allowing computation to succeed with incorrect results. MU-Prolog's implementation of negation as failure has been made sound by delaying the attempted solution of any negative goal until there are no unbound variables present in the goal [Lloyd 1984]. This solution has the problem that computations can terminate without a solution if the only goals remaining to be solved are negative literals with unbound variables. Finally, problems arise in modeling changing systems when using default inference rules such as negation as failure. In particular, inferences which have already been made using the default assumption may have to be revised if new information which overrides the assumption is acquired by the system. This leads to the need for complicated bookkeeping procedures such as the one described by Reiter [1978].

Lloyd [1984a] also showed that the expressiveness of logic programs can be extended considerably by using a sound implementation of negation as failure. In
particular, *extended programs* can include arbitrary first order formulas in the clause bodies or in queries. Lloyd described a system with negation as failure in which inference is sound. Some work has been done [Shepherdson 1985], [Lloyd 1984] on the justification of negation as failure based on the closed world assumption [Reiter 1978] and data base completion [Clark 1978]. It is however, beyond the scope of this thesis to detail their results here. However, Shepherdson claims that it is "doubtful that there is in general any simple logical characterization of negation as failure".

Negation as failure does not permit the solution of negative goals with unbound variables. Voda's [1986a] *precomplete negation* can compute solutions for these goals but encounters serious efficiency problems when doing so. Voda deliberately used precomplete negation instead of full classical negation since incorporating classical negation in logic programs was too costly in terms of efficiency.

**2.2.2.2 Extending Procedural Efficiency of Logic Programs**

As mentioned above, the general problem of inference in logic programs is undecidable. Thus there is no hope of achieving full procedural adequacy in the KR form. However, inferencing in logic programs is far more efficient than inferencing in full FOL [Voda 1986]. In addition, there are several ways in which implementations of the logic program form can be made more efficient thus achieving greater degrees of *procedural efficiency* in lieu of procedural adequacy.
Depth First Search

In both Prolog [Clocksin 1981] and Planner [Hewitt 1971] the general procedural strategy is depth-first, left to right search with automatic backtracking on failure. This strategy can be easily implemented but is an inefficient approach for many problems. The use of the depth-first strategy can also lead to a form of incompleteness which is not present in the underlying resolution proof procedure. In particular, the depth-first approach might look down an infinite branch of the search tree yielding no solution when one exists elsewhere in the tree. This incompleteness can be avoided by using a "fair" selection rule - one which is guaranteed to look down every branch of the tree eventually - but only at serious cost to the efficiency of the system. In effect, the use of depth-first search trades off two aspects of procedural adequacy: efficiency for completeness.

Ordering the Knowledge Base

In both Prolog and Planner the ordering of clauses in the KB and the ordering of literals within the clauses affects the way in which search is performed. This gives the user a degree of control over search for a given program. The use of this technique often allows users to write programs which run almost deterministically (with little backtracking). However, one aspect of a procedurally adequate KR system is that knowledge be used in a flexible manner, that it allow control to flow in any direction: from data to goal or goal to data. Mackworth [1987] calls this synthesis/analysis flexibility. In standard implementations such as Prolog, ordering the KB is done to expedite the goal driven search used by the interpreter but this is often at the expense of being able to reverse the flow of control.
CHAPTER 2: LITERATURE REVIEW

Extra-Logical Features

Many logic programming systems give the user additional control over search by providing some extra-logical control features. Prolog's cut allows the user to tell the interpreter not to repeat a particular approach to a solution upon backtracking. Unfortunately this feature can in some cases lead to a system which answers “no” to a goal which is in fact implied by the KB [Lloyd 1984]. MU-Prolog [Naish 1984] provides the wait primitive with which the user can indicate that a particular goal is not to be attempted until specific arguments are ground. Like clause ordering, these features enable users to write efficient programs at the expense of synthesis/analysis flexibility. They also put the onus on the programmer to decide how the knowledge will be used in order that it be used efficiently.

Metalogical information

In Planner [Hewitt 1971], the user can specify via recommendations which path search should follow depending on the conditions which exist during the search. Vasak [1985] outlines a method for the incorporation of metalogical control for logic programs. The proposed control takes the form of annotations to predicate definitions which on a logical level represent “simple theorems about ground terms that will satisfy a clause”. The annotations are generally used to obtain termination conditions for recursively defined predicates and thus are only of use when predicates are used in a non-deterministic mode. Vasak proposes that the annotation could also be used “constructively to guide the computation in the correct direction” when predicates are being used in the generate and test mode, an approach which is amenable to synthesis/analysis flexibility in search.
CHAPTER 2: LITERATURE REVIEW

Automatic Control

In order to remove onus from the programmer to the system, Naish [1985] developed some techniques for automating the control of logic programs. He described a program which will automatically generate \textit{wait} declarations for recursive Prolog predicates which if called incorrectly would "guess" at one of an infinite number of solutions. This relieves the programmer of having to foresee all ways in which a predicate might be used and having to add the appropriate control information. This approach differs from those mentioned above in that the control heuristics developed by Naish are applicable in general and do not require that the user understand the underlying control facilities of the system.

Summary

The control features mentioned above are generally motivated by practical considerations. Some, such as the depth-first search and cut intentionally sacrifice the completeness of the underlying resolution strategy for the benefit of efficient implementations and the ability to write efficient programs. Others sacrifice synthesis/analysis flexibility for efficient one-way computation. In most cases these trade-offs were chosen because the implementations are devised as programming languages rather than knowledge representation systems. Thus the decisions made during development reflected that bias. In addition, most of the control facilities are provided to allow users to guide search in a particular program only. None of these systems offer the user a choice of altering the general top-down search.
2.2.3 Flexible Search

Among the requirements for procedural adequacy is the ability of a KR system to use its knowledge in a flexible manner. As discussed above, standard implementations of logic programming are particularly inflexible in their approach to search, since most use a depth-first, left to right strategy to solve all problems. Additionally, users are in practice encouraged to create programs which operate efficiently in this environment using special control features as well as their own knowledge of the interpreter's behaviour. This style of programming tends to remove the synthesis/analysis flexibility inherent in the logic program representation. If the logic program form is to be of greater use for knowledge representation, then the emphasis must be shifted away from creating programs and towards creating KBs in which the data can be used in a more flexible manner. This may involve a trade-off between two aspects of procedural adequacy - efficiency for flexibility. A goal of this thesis is to demonstrate the flexibility of search in a logic based KR form using connection graphs.

In this section, a search strategy taken from the computer vision domain is presented. The strategy requires flexibility with regards to search direction. In section 3.1 it is shown how this search can be brought into the connection graph proof procedure, illustrating the flexibility of search in the representational formalism.

Two standard approaches to search are top-down or bottom-up. Havens argues that neither in itself is sufficient to achieve procedural adequacy. Top-down visual recognition requires that the existence of some complex object in an image be hypothesised. The object is recognized by finding its component parts
which may themselves be complex objects. The problem with such a search is that a correct guess must be made as to what will be found before it can be found - the chicken and egg problem [Havens 1983]. Furthermore, any complex object will have many possible compositions, hence the search strategy will have many choices at every level of composition and must make the right one fairly quickly in order to avoid thorough exploration of an exponential search space. Unfortunately, the only way to be sure that the correct choice has been made is to verify the choice with data from the image, the lowest rung on the composition hierarchy. Top-down search is also known as goal driven search since the search is directed by the desired result. It is the standard search employed by logic programming systems. Clearly this strategy has merits when the goal is explicitly stated, as it generally is in logic programs. However in visual recognition, the end goal is usually undefined or very weakly defined and hence goal driven search loses its effectiveness.

On the other hand, a bottom-up (or data driven) search requires looking at all the lowest level visual cues and hypothesising the existence of each of the objects which each cue could possibly be part of. Then more and more complex objects are formed from those already hypothesised. One problem with the bottom-up approach is that many objects will be hypothesised which must later be discarded when they are discovered to be inconsistent with the rest of the data. The problem is compounded since a large amount of time can be spent trying to compose objects based on incorrect hypotheses before those hypotheses can be reasonably rejected.

Havens [1983] proposed a hybrid search strategy which integrates top-down (goal-driven) and bottom-up (data-driven) search. The control structure for the
strategy is based on the formal context-free parsing method of Earley. Earley's algorithm creates a process for some rule in a context-free grammar “whenever (and not until) the left hand side of the rule can be part of some global interpretation beginning at the current input symbol”. This is called the predictor function. Then “each process tries to find an occurrence of its left hand side by finding a sentential form in the input satisfying each symbol in the right hand side of the rule” [Havens 1983]. This is achieved by the scanner and completer functions in the case of terminal and non-terminal symbols respectively, of the rule's right hand side. This results in a control structure in which processes cooperate when low level processes complete subtasks for higher level processes and which compete when many processes try to recognize a given left hand side at the same point in the input.

However, according to Havens, Earley's algorithm is too limited to use for artificial intelligence recognition tasks. These tasks, unlike context-free parsing, do not assume intrinsically ordered input and are believed to be at least context-sensitive. Thus, Havens proposed a search for visual recognition makes use of Earley's control structure but is only loosely based on Earley's algorithm. Havens' search has three phases called expectation, matching and completion which are roughly analogous to the predictor, scanner and completer functions of Earley's algorithm. The details of Havens' search are presented in section 3.1 at which point its incorporation into connection graphs is also discussed.
CHAPTER 2: LITERATURE REVIEW

2.2.4 Constraint Satisfaction

The logic program form can not be made procedurally adequate due to the complexity of inferencing in Horn clauses. As has been discussed throughout this section however, measures can be taken to attain greater degrees of procedural efficiency within the form. Another approach to achieving procedural efficiency is to incorporate efficient approximation techniques such as constraint satisfaction algorithms [Mackworth 1987]. A major focus of this thesis is the applicability of constraint satisfaction algorithms to the logic program form using connection graphs. To this end, a review of constraint satisfaction techniques is presented below for comparison with approaches to constraint satisfaction in connection graphs.

Many combinatorial search problems can be formulated as constraint satisfaction problems. Constraint satisfaction problems have been well studied and the principles and algorithms for solving them have been applied to search in various applications [Freuder 1978], [Haralick 1980], [Havens 1985], [Mackworth 1985], [Montanari 1974], [Waltz 1972].

Mackworth [1985] characterized the boolean constrain satisfaction problem as follows:

"given is a set $V$ of $n$ variables $\{v_1, v_2, ..., v_n\}$, associated with each variable $v_i$ a domain $D_i$ of possible values. On some specified subset of those values there are constraint relations given which are subsets of the Cartesian product of the domains of the variables involved. The set of solutions is the largest subset of the Cartesian product of all the given variable domains such that each $n$-tuple in the set satisfies all the given constraint relations. One may be required to find the entire set of solutions, one member of the set or simply to report if the set has any members."
Figure 3a is a graphical representation of the following constraint satisfaction problem over the variables \( x, y, \) and \( z \). Constraint \( D_x \) limits the values of \( x \) to be one of \( a \) or \( b \), i.e. \( D_x = \{a, b\} \). The unary constraints on \( y \) and \( z \) are \( D_y = \{c, d\} \) and \( D_z = \{e, f, g\} \) respectively. The domains for each variable are the same as the corresponding unary constraint predicate, e.g. the domain of the variable \( x \) is \( \{a, b\} \). The binary constraint between \( x \) and \( y \) specifies that either \( x = a \) and \( y = c \), or \( x = b \) and \( y = d \), i.e. \( P_{xy} = \{ac, bd\} \). Likewise \( P_{yz} = \{ce, df, dg\} \) and \( P_{xz} = \{ae, af, be\} \).

Several equivalent graphical representations for constraint satisfaction problems have been used. Figure 3a uses the representation of Freuder [1978] in which the nodes in the graph represent the constraint relations over the variables. The variables themselves are not explicitly represented in the graph but can be understood to be associated with the unary relations. Two nodes representing the relations \( P \) and \( Q \) are joined by an arc if the variables related by \( P \) are a subset of those related by \( Q \) or vice versa.
Mackworth [1985] has provided a review of several algorithms which have been used to solve constraint satisfaction problems. The principles and performance of some of these algorithms are summarized below. The generate and test approach to solving constraint satisfaction problems will explore the entire space of possible solutions, that is, the Cartesian product of the variable domains. Backtracking algorithms instantiate the variables in some fixed order. Backtracking algorithms avoid searching large portions of this space by not considering all the possible instantiations for uninstantiated variables when the currently instantiated variables do not satisfy the constraint relations. Nevertheless, backtrack search will usually exhibit ‘thrashing’ behavior; that is, “repeated exploration of the search tree which differ only in inessential features” [Bobrow 1974]. Neither of these approaches use the available constraint information to guide the order in which solutions are considered or, in the generate and test case, to avoid considering impossible solutions. Rather, they
use the constraints to confirm or deny that a chosen solution or partial solution is valid. In Figure 3a for instance, the generate and test approach will consider each of \{acg adg bcg bdg\} as possible solutions for the problem in spite of the fact that the constraint \(P_{xz}\) makes it impossible for \(z\) to have the value \(g\) in any solution.

Consistency algorithms generally operate to remove local inconsistencies from the graph before search commences. A constraint graph can be locally consistent to varying degrees, depending on the number of variables in the graph. A constraint satisfaction problem is said to be \(k\)-consistent if given any \((k-1)\) variables instantiated such that all the constraint relations among those variables are satisfied, there exists some value in the domain of any \(k\)th variable such that the instantiations of all \(k\) variables taken together satisfy all the constraint relations between the variables [Mackworth 1985].

Thus for a constraint graph to be \(1\)-consistent, there must exist a value in the domain of each variable such that the unary constraint relation on that variable is satisfied. Usually, this simply means that the unary constraint relation on each variable must always be non-empty since the unary constraint relation on a variable is usually a subset of the values in the domain of that variable.

For a constraint graph to be \(2\)-consistent, for every pair of variables \(v_i\) and \(v_j\) and for every element in the domain of \(v_i\) which satisfies the unary constraint on \(v_i\), there must be at least one element in the domain of \(v_j\) which satisfies the unary relation over \(v_j\) and which satisfies the binary relation over \(v_i\) and \(v_j\). The consistency graph in Figure 3a is not \(2\)-consistent since for the value \(g\) in the domain of \(z\) there is no value in the domain of \(x\) which satisfies \(P_{xz}\). Note that the
CHAPTER 2: LITERATURE REVIEW

graph would become 2-consistent if the element \( g \) were removed from the relation \( Dz \). The element \( dg \) in the relation \( Pyz \) can no longer cause values for \( x \) and \( y \) to become consistent, so it too can be removed from the graph. The 2-consistent version of the graph is given in Figure 3b. Several algorithms have appeared in the literature which essentially compute 2-consistency in some representation of a constraint satisfaction problem. Among them are Waltz’s [1972] picture interpretation algorithm and Mackworth’s [1977] arc-consistency algorithm.

For a graph to be 3-consistent, given any two variables \( v_i \) and \( v_j \) which are instantiated with all the constraints over \( v_i \) and \( v_j \) satisfied, then for any third variable \( v_k \), there must exist a value in the domain of \( v_k \) such that all the constraints over \( v_i, v_j \) and \( v_k \) are satisfied. The constraint graph in Figure 3b has been made 2-consistent but it is not yet 3-consistent. Consider the values \( b \) for \( x \) and \( d \) for \( y \), (all the constraints over \( x \) and \( y \) are satisfied); there is no value for \( z \) which will simultaneously satisfy \( Pyz \) and \( Pxz \). If the element \( bd \) is removed from the constraint relation \( Pxy \) in order to impose 3-consistency, the graph becomes 2-inconsistent once again (Figure 4a). The value \( b \) for \( x \) is no longer consistent with any value for \( y \) under \( Pxy \) and likewise \( d \) for \( y \) is not consistent with \( x \) under \( Pxy \). If those elements are removed, the elements \( eb \) in \( Pxz \) and \( df \) in \( Pyz \) can be removed also. The value \( f \) for \( z \) now becomes inconsistent with \( y \) under \( Pyz \). In order to make the graph 2-consistent the element \( f \) can be removed from \( Dz \) and subsequently \( af \) can be removed from \( Pxz \). The resulting constraint graph is presented in Figure 4b. Note that it is now 2-consistent and 3-consistent. Before Freuder’s [1978] k-consistency algorithm unified the notions of consistency in constraint graphs, both Mackworth [1977] and Montanari [1974] introduced
algorithms which compute 3-consistency as *path consistency* in constraint graphs.

Freuder provided an algorithm for achieving k-consistency for any value of \( k \leq n \) in a constraint satisfaction problem of \( n \) variables. The algorithm synthesizes successively higher order constraints while maintaining consistency within the graph. The details and notation of his algorithm are presented in section 2.2.4.1.

If the k-consistency algorithm causes all the members to be deleted from any constraint relation, then there can be no solution to the problem. The converse is not true however. Making a graph k-consistent for \( k < n \) does not guarantee that a solution exists. That is, there exist k-consistent graphs \((k<n)\) which have no solution. In order to guarantee that consistency techniques will provide solutions to a constraint satisfaction problem, the problem must be made
k-consistent for $k=n$. Freuder's algorithm for achieving $k$-consistency (for any value of $k \leq n$) is $O(n^k)$ [Mackworth 1985] which makes it too inefficient to be of practical use for finding complete solutions to constraint satisfaction problems. Although computing $k$-consistency for $k<n$ does not always guarantee that a solution will be found, it will remove many inconsistent values from the constraint relations thus greatly reducing the size of the combinatorial solution space which must subsequently be searched. Therefore it is generally the case that when consistency algorithms are used, they are used as a preprocessing step to some form of search like backtracking. "By doing a limited amount of local computation, constraint satisfaction algorithms can optimize the backtrack search sufficiently to achieve an overall improvement in performance on some difficult problems" [Mackworth 1985].

A natural question arises: for what value of $k$ is it reasonable to compute $k$-consistency? For $k=n$ the consistency algorithm appears to be too inefficient to be of practical use. Haralick [1980] compared several algorithms for finding solutions to the n-queens problem for various values of $n$ as well as some other randomly generated constraint satisfaction problems. Their lookahead algorithm computes 2-consistency. Their forward checking algorithm computes a weaker form of 2-consistency. In forward checking, consistency checks are not made until some subset of the variables have been assigned a value from their respective domains and then only the relations over the instantiated variables are checked for consistency. They also compared backtracking, backchecking and backmarking (keeping track of failed search paths to cut down on thrashing behavior). Their empirical results showed that forward checking actually outperformed lookahead under several performance metrics. This suggests that
the benefits of k-consistency preprocessing for k>2 will not outweigh the costs of computing it and that even full 2-consistency preprocessing is of dubious value! This result may depend on a number of factors. If the search space contains many solutions (as is the case for the n-queens problem) the consistency algorithms will be at their least effective since fewer consistency checks will result in graph reductions than in a problem with relatively few solutions. If the search space contains many solutions and only a single solution is being sought, then arc consistency may spend a lot of effort processing parts of the search space which a single solution backtracking scheme would never consider. Also, since we were given no assurance that the algorithms used were optimal under the performance metrics used (table lookups and number of consistency checks), it is possible that the performance of the algorithms may differ in other implementations.

Although Haralick's result argues for the use of little consistency preprocessing, it should be noted that his results also showed a drastic reduction in the size of the search space for a problem as the amount of consistency preprocessing increased. It is also worth noting that in the many domains which consistency techniques have independently been developed, none of the algorithms compute k-consistency for k>3 and many only compute it for k=2.

It has been suggested that search for solutions in a logic programming system could benefit from the use of some constraint satisfaction preprocessing [Kowalski 1975], [Mackworth 1985]. A proposed mechanism for doing this is reviewed in section 2.3.2 and analysed in terms of Freuder's k-consistency algorithm in section 3.2.
CHAPTER 2: LITERATURE REVIEW

2.2.4.1 Freuder's k-consistency Algorithm

The notation and details of Freuder's k-consistency algorithm are presented here so that the algorithm can be compared to the consistency algorithm for connection graphs in section 3.2.

In a constraint satisfaction problem (CSP), we are given a set of variables \( V = \{X_1, \ldots, X_n\} \) which may take on values from a set of discrete and finite domains \( D_1, \ldots, D_n \) respectively. Given also is a set \( C \) of k-ary constraints amongst the variables. The problem is to synthesize the global relation over \( V \). Every element of that relation is a solution to the CSP and every solution to the CSP is an element of the global relation. Freuder's algorithm will compute the global relation for an arbitrary CSP [Havens 1984].

The terminology and details of the algorithm are now reviewed. Let \( I = \{1, \ldots, n\} \), the index set of the elements of \( V \). Let \( J \) be any non-empty subset of \( I \). \( X_J \) is the indexed set of variables \( \{X_j\}_{j \in J} \). This allows the reference to an arbitrary subset of \( V \) with a single name.

A value \( a_j \) in \( D_j \) is called an instantiation of \( X_j \) and an instantiation of a set of variables \( X_J \) is denoted by \( a_J \): an indexed set of values \( \{a_j\}_{j \in J} \). A constraint over a set of variables \( X_J \) is denoted \( C_J \) and is given as a set of instantiations of \( X_J \). A constraint \( C_J \) then is a subset of the cross product of the domains \( D_J \). The goal of Freuder's algorithm then is to discover the elements of the constraint \( C_I \).

An instantiation \( a_J \) is said to satisfy a constraint \( C_J \) iff \( a_J \in C_J \). An instantiation \( a_J \) satisfies a constraint \( C_H \) where \( J \supseteq H \) if the set \( \{a_j \in a_J\}_{j \in H} \) is a member of the constraint \( C_H \). That is, if \( a_J \) is a projection of \( C_J \) onto \( C_H \). An
instantiation \( a_J \) satisfies constraint \( C_H \) where \( J \subset H \) if there is an \( a_H \) in \( C_H \) such that \( a_J = \{ a_j \in a_H \}_{j \in J} \). That is, \( a_J \) is a projection of \( C_H \) onto \( C_J \) [Havens 1984].

The order \( n \) constraint expression \( C \) is the conjunction of all constraints \( C_J \) where \( I \supseteq J \) (Freuder denotes \( C = \wedge C_J \)). Not all such constraints are necessarily specified but they can all be assumed to exist. Such a non-constraint \( C_J \) can always be initially specified as the cross product of the domains of the variables \( X_J \).

To locally propagate a constraint \( C_J \) to a neighbor \( C_H \), remove from \( C_H \) any \( a_H \) which does not satisfy \( C_J \). Global propagation of a constraint \( C_J \) through a neighbor \( C_H \) is defined as local propagation of \( C_J \) through \( C_H \) and if any \( a_H \) was removed from \( C_H \), global propagation of \( C_H \) through all of its neighbors except \( C_J \). To propagate \( C_J \), globally propagate it through all of its neighbors. In the \( k \)-consistency algorithm, a constraint graph is constructed with nodes corresponding to constraints and thus the notion of propagation can apply to nodes as well.

Freuder's [1978] \( k \)-consistency algorithm is reproduced in Figure 5.
Given $C = \bigwedge C_J$ for every $J$ subset of $I$, the algorithm is defined inductively:

step 1) Construct a constraint graph with nodes $N_J$ corresponding to constraints $C_J$ in the given constraint expression for all $J$ subset of $I$ where $|J| = 1$.

step $k+1$) for all $J$ subset of $I$ of cardinality $k+1$:
   Add the node $N_J$ corresponding to the constraint $C_J$ to the graph. Link $N_J$ to all $N_H$ such that $H$ is a cardinality $K$ subset of $J$.
   Locally propagate to $N_J$ from all of its neighbors. Propagate $N_J$.

Figure 5: Freuder's k-consistency algorithm.

If no constraint $C_J$ is specified, it can be synthesized from existing constraints or from the domains of the variables $X_J$. Freuder showed that if the algorithm is run for $n$ steps that the node $N_I$ corresponds to the global constraint $C_I$ defined by the order $n$ constraint expression $C$. The elements $a_i$ in $N_I$ in the final network are exactly the set of solutions to the constraint satisfaction problem. When the algorithm is run for $k$ steps where $k<n$, it produces a constraint graph that is $k$-consistent (see above). Hence the algorithm can be used to find solutions or as a preprocessor to reduce the space which must subsequently be searched [Freuder 1978].

2.3 Connection Graph Research

In this section some of the research pertaining to connection graphs is reviewed. Some findings relevant to the proof procedure itself are presented first. The focus of this section however, is on the search procedures and the use of
consistency techniques in several existing implementations of the connection graph method as presented in sections 2.3.1 and 2.3.2.

Several researchers have described variations and improvements to the connection graph proof procedure as originally presented by Kowalski. Bruynooghe [1976] showed that step 3bii) of the proof procedure can lead to the reintroduction of links which have already been removed. He showed that this problem can be avoided by admitting 'internal' links between potentially complementary literals within clauses during graph construction and the removal of step 3bii) in the processing phase. Bruynooghe described a method for bringing the structure sharing approach of Boyer [1972] into a connection graph implementation. The use of structure sharing in a connection graph implementation is discussed in 3.3.3. Bruynooghe also describes an algorithm for determining the compatibility of two unifiers.

Bibel [1981] made a major contribution to the connection graph proof procedure by proving it to be sound and complete for the ground case. He also proved that several preprocessing operations on connection graphs are 'proper' in that they do not exclude any correct solutions or introduce any incorrect solutions. Some of these preprocessing operations will be discussed in section 2.3.1.

Several researchers have implemented theorem provers based on connection graphs [Bibel 1981], [Blausius 1984], [Bruynooghe 1976], [Chang 1979], [Sickel 1976] and [Stickel 1982]. Most of these systems focus on search methods in connection graphs and will be reviewed below.
2.3.1 Search in Connection Graphs

Search strategy in connection graphs is determined by the order in which links in the graph are selected for processing in step 3a) of the proof procedure. Determining that order is similar to ordering the generation of resolvents in other resolution systems. Simple search strategies such as unit preference [Nilsson 1971] or SL-resolution used by other resolution systems are easily simulated in connection graphs by appropriate link selection algorithms [Kowalski 1975]. The connection graph representation however, does not restrict the search to such simple global procedures. Some examples of more flexible search strategies made possible by the connection graph data structure are given below.

2.3.1.1 Graph Reductions

One approach common to many of the connection graph systems is to process or preprocess the connection graph in order to reduce it before actually searching. The rationale is that any reductions in the size of the graph (either the number of links or literals) will greatly reduce the size of the exponential space which must subsequently be searched. A reduction operation is said to be proper if: i) its application reduces the number of literals or links in the connection graph and; ii) it can be performed quickly (polynomial time with a small exponent) for any connection graph [Bibel 1981]. Graph reduction operations can be performed once on the original connection graph or as an integral part of the proof procedure, as an additional step in the processing of the graph.

One such reduction operation is the removal of pure clauses in step 3cii) of Kowalski's proof procedure. If there is a pure literal (that is, one with no links) in
CHAPTER 2: LITERATURE REVIEW

the graph, the clause containing that literal can be removed along with any links to the other literals in that clause. This reduction operation is performed on the original connection graph and after every resolution step. The removal of pure clauses is an example of the type of reduction operation which takes advantage of the structure of the graph itself to assist the general search procedure.

Bibel [1981] showed the removal of pure clauses to be proper. He also showed that if clause C is pure in connection graph G and graph G' is the graph G without the clause C then a refutation can be found in G iff it can be found in G'. The same properties were demonstrated for several other standard theorem proving reductions as well as for a newly discovered one. These reduction operations include tautology removal, clause subsumption, graph splitting and the removal of certain types of cycles in the graph. It is beyond the scope of this thesis to present the details of these operations.

Blasius [1984] pointed out that one benefit of using the reduction operations is the potential for the 'snowballing effect' when they are used in conjunction. For example, a link removed by one reduction operation could make a clause pure, causing the deletion of more literals and links which may in turn cause the deletion of more links, and so on. However, this effect has not formally been described for single reduction operations much less for several interacting operations and hence the benefits are hard to quantify. Furthermore, Blasius reported that the unrestricted use of all of the reduction operations leads to an inconsistent and even incomplete proof procedure!

Another way to reduce a connection graph is through the resolution step itself. The search strategy can attempt to choose links in such a way that the connection graph becomes simpler or smaller after the resolution is performed.
This approach is illustrated in Figure 6 and Figure 7. In Figure 6a, the marked link joins two literals neither of which is connected to any other literal in the graph. In Figure 6b the resolution has been performed and the resolvent added to the graph according to step 3a) of the connection graph proof procedure. Notice that both original parent clauses are now pure since they each contain a pure literal. In Figure 6c the pure clauses have been deleted according to step 3c) of the proof procedure. The net size of the connection graph has been reduced. This will always be the case when a link is selected for resolution and the literals connected by that link are not connected to any other literals in the graph.

Another situation in which the link selection will reduce the graph is illustrated in Figure 7. If the literal in a unit clause is connected to some other literal which is in turn not connected to any other in the graph, then choosing the link between

---

**Figure 6**: Reducing a connection graph using link selections. In a) the marked link is the only link attached to the literals it connects. Resolution on that link occurs at 1 and the resolvent is added to the graph in b). At 2, the pure clauses are removed from the graph leaving c), the simplified version of a).
those two literals will result in the connection graph being reduced by one link and one literal. In Figure 7a the marked link is such a link. In Figure 7b the resolution has been performed and the resolvent added to the graph. Notice that one of the parents is now pure. In Figure 7c the pure clause has been removed, leaving a graph that is one link and one literal smaller than that in Figure 7a. Bibel [1984] showed that both link selection methods were proper since they cause a net reduction in the size of the graph and the associated computations are minimal.

![Figure 7: Reducing a connection graph using link selections. In a) the marked link is selected for resolution. Resolution on that link occurs at 1 and the resolvent is added to the graph in b). At 2, the pure clauses are removed from the graph leaving c), the simplified version of a).](image)

Like the graph reduction operations, consistency techniques can reduce the complexity of a connection graph by removing links which can be shown not to be usable in any refutation. The use of consistency techniques in connection graphs will be discussed in section 2.3.2.
CHAPTER 2: LITERATURE REVIEW

In review, graph reduction operations can aid search in a connection graph by reducing the complexity of the graph, thus making large reductions in the exponential space which must be subsequently searched. A connection graph can be reduced by identifying and removing structures within the graph which can not participate in any solution, by selecting links for resolution which are guaranteed to cause a net reduction in the size of the graph or by removing links with some form of consistency check. Reduction operations can be performed as a preprocessing step or as an integral part of the proof procedure.

2.3.1.2 Planning Refutations

Several researchers have taken an alternate approach to searching in connection graphs. Since connection graphs explicitly represent every possible resolution within a set of clauses, it is possible to look ahead in the graph at potential paths to a refutation without actually performing any resolutions. Kowalski [1975] calls this "taking advantage of the ungenerated portion of the search space". Such a strategy can avoid performing resolutions where no refutation path exists and instead try to perform resolutions on what is likely to be the shortest path to a refutation. Several existing connection graph implementations which incorporate this technique in their search strategies are reviewed below.

Although Kowalski did not implement a connection graph proof procedure, he described a technique for using the ungenerated part of the search space for calculating the lower bound on the complexity of the simplest refutation involving a given clause. A simple estimate of this lower bound is the number of literals in
the clause. In Figure 8, this estimate is 3 for clause A and 2 for clause B. This strategy assumes that each literal can be resolved away in one step. That is, each literal is connected to a unit clause. The connection graph data structure however, allows for better estimation of this lower bound. The simple strategy above corresponds to a one-level look-ahead in the graph. A two-level look-ahead estimate involves finding the smallest clause to which each literal is attached and calculating the sum of the minimums over the literals in the clause. In Figure 8, using a two-level look-ahead, the estimate for clause A remains 3 (since all literals are indeed connected to unit clauses) and for clause B it becomes 4. The estimate can be improved by an n-level look-ahead. The information gained by this type of estimate can be used to direct the search strategy as to which clause appears most likely to lead to a quick refutation.

---

Figure 8: An example connection graph. Clause A contains 3 literals and could lead to a refutation in 3 resolution steps. Clause B contains only two literals but will require at least 4 resolution steps to lead to a refutation.
While Kowalski's approach to planning the refutation involves looking a few steps along possible refutation paths, Antoniou [1983] look for complete unit refutations in the graph. Before performing any resolution, the connection graph is checked for the existence of a terminator configuration (also called a refutation tree). A refutation tree is a subgraph which contains no cycles, in which every literal is connected to exactly one other literal, and in which the unifiers of all the links are compatible. Figure 9 contains two examples of terminator configurations. It is known that if a set of clauses is unit refutable then the connection graph will contain a refutation tree. Antoniou [1983] described an efficient algorithm for finding a terminator condition in a connection graph as a pre-theorem proving step. They reported that their approach proved a particular theorem by generating 190 unit clauses as compared to another reported solution which deduced 22,000 clauses before finding a proof [Antoniou 1983].

Figure 9: Example terminator configurations. a) is a refutation tree provided the unifiers are compatible. In b) the shaded portion of the connection graph is a refutation tree if the labeled unifiers are compatible.
Chapter 2: Literature Review

Exploring the ungenerated search space was taken to its extreme by Chang [1979]. They too used the connection graph representation of a problem to look for paths from a goal to unit clauses. These are called refutation plans. They are only plans since no unification or compatibility checking is done while they are being constructed. Actually, because loops may occur in the graph, many refutation plans share common paths in the graph. Therefore, the connection graph is used to construct a context-free grammar which in turn generates refutation plans. Finally, the refutation plans are considered one at a time until one of them proves to be an actual refutation. A plan becomes a refutation if all of the unifiers from the connection graph on the path that generated the plan are compatible. The reported benefit of this approach is that it avoids attempts to search along redundant paths. In particular, they avoided considering permuted orders of a given search path. Their strategy however, is essentially a generate and test approach which considers large portions of the search space that even a simple backtracking algorithm would avoid.

Sickel [1976] described a similar system to that of Chang. Sickel created proof schemata which are similar to the context free grammars for generating refutation plans. The major difference is that Sickel introduced and used the notions of horizontal and vertical consistency. Vertical consistency requires that the unifiers along any path or subpath within a proof schema be compatible. Horizontal consistency requires that the unifiers for separate subgoals within a clause be compatible as well. Horizontal consistency is not implemented in the system. More will be discussed about the notion of consistency in connection graphs in section 2.3.2.
2.3.1.3 Heuristic Search

Because links can be selected for processing in any order, a connection graph search strategy is completely flexible. Thus the choice of which link to choose for resolution at any stage of the procedure can be made on the basis of any heuristic ordering scheme. For example, in the discussion above it was noted that the selection of certain types of links would always lead to a net reduction in the size of the connection graph. Thus, graph size reduction can be used as a heuristic to guide search.

Since the order of link selection is flexible, it is possible to use strategies which are not committed to either a top down or bottom up approach. For this reason cooperative searches like that described in section 3.1 can be incorporated in a connection graph proof procedure. This type of search can be driven by some local criteria such as the structure of the graph, the results obtained to date or the degree to which a given goal has been solved. In the absence of such information, the search can defer to a global strategy such as backtracking. This combined approach is pursued in this thesis. The particular strategy used and the results of its use are described in section 3.3.2.

Blasius [1984] stated that a goal of their theorem proving system was that it should "display an active and directed behavior in its striving for a proof, rather than the passive combinatorial search through very large search spaces". In attempting to achieve this goal, they have tested a great many search heuristics. For example, they experimented with link selection heuristics which would a) minimize the complexity of the graph after resolution, b) create the resolvent with the fewest literals, c) create the resolvent with the terms of the least complexity, and d) minimize the average size of clauses in the resulting graph. For many of
their heuristics however, they found that the expense involved in the associated computations outweighed any benefit gained by using the heuristic. They found, based on empirical results, that the most important heuristic was graph size reduction, much as Kowalski suggested in his original paper.

Stickel [1982] described a link scheduler which gives numerical precedence to each of the links in the graph based on heuristics. The highest scoring link is chosen for processing at each cycle of the system. The scheduler in that system “can act on such important facts as the number of links attached to an atom [literal]”. Details of the scheduler were not described.

Meta-knowledge (such as filters or recommendation lists) can also be used to guide search in connection graphs. This knowledge might be used to suggest which of a set of alternative goals it might be best to pursue under given circumstances. However, meta-knowledge will not necessarily share the same representation as the problems in the system. Thus the incorporation of meta-knowledge is more dependent on the system implementation than on the problem representation. The use of meta-knowledge for logic programming then, does not benefit particularly from the use of the connection graph representation but rather from the way in which the connection graphs are implemented. In section 3.3.2 the use of meta-knowledge to guide search in an object oriented connection graph implementation is discussed.

2.3.2 Constraint Satisfaction in Connection Graphs

Kowalski described a way in which the constraint satisfaction approach can be brought into the connection graph proof procedure. He described a
CHAPTER 2: LITERATURE REVIEW

filtering algorithm which is similar in spirit to the consistency algorithms described above and based on Waltz's [1972] picture interpretation algorithm.

Kowalski's filtering algorithm for connection graphs removes links which cannot be used in any refutation, that is, links which are locally inconsistent. If a link to some literal in a clause is incompatible with all of the links to some other literal in that same clause, the link can be removed from the graph since no refutation involving that link could subsequently resolve away the other literal. In Figure 10 for example, the unifier \( c/x, g(b)/y \) associated with link 1 is not compatible with any link to the literal \( Q(x, y) \)\(^\dagger\) and thus can be removed from the graph. Any link which is not compatible with at least one link to every other literal in the same clause will be called inconsistent. Kowalski's filtering algorithm removes any inconsistent links from the graph at every cycle of the connection graph proof procedure. The removal of links can cause other links to become inconsistent and thus cause further deletions. The deletions will propagate until every link is compatible with at least one link to every other literal in the same clause. This behaviour is illustrated in Figure 10.

\(^\dagger\)See the definition of incompatible links in section 2.1.2.
Figure 10: The deletion of incompatible links: Link 1 can be removed since it is not compatible with any link to the middle literal. Link 2 can also be removed since it is not compatible with any link to the rightmost literal. After the removal of link 2, link 3 is no longer compatible with any link to the middle literal and is therefore removed. Link 4 is not compatible with any link to the middle literal thus it too can be removed.

Kowalski's link filtering algorithm for connection graphs is presented in Figure 11. This algorithm can be made more efficient by repeating the compatibility checks in step 2 for only those links which may have been affected by the deletions in step 1. Also, it could be improved by keeping track of previous compatibility checks. However, as the algorithm stands, it demonstrates that constraint satisfaction techniques can be used as a graph reduction operation in the connection graph proof procedure. At issue is the question of whether or not it is a proper reduction operation or whether it is too expensive to be of practical use. This issue is addressed in 3.2.
CHAPTER 2: LITERATURE REVIEW

After the construction of the connection graph and after every cycle of the procedure:
1) remove any inconsistent link from the graph
2) repeat step 1 until there are no deletions

Figure 11: Kowalski's link filter for connection graphs.

Logic programming has been suggested as an application for constraint satisfaction techniques [Mackworth 1985], and at first glance it seems that the link filter is a good example of its usefulness. However, Kowalski showed that the consistency test for a given link is subsumed by the operation of resolving that link and the removal of pure clauses from the graph. By step 3bii) of the connection graph proof procedure†, the resolvent will inherit only those links which are compatible with the selected link. If a link is chosen which is not compatible with at least one link to every other literal in its clause, then the resolvent will necessarily contain a pure literal and will consequently be removed in step 3c) after the resolution. The net effect will be the removal of the inconsistent link from the graph. Since the main value of constraint satisfaction is as a preprocessing operation [Mackworth 1985], this result indicates that there is in fact no benefit to including this constraint test in connection graphs. The use of consistency techniques in connection graphs and Kowalski's finding are examined in detail in section 3.2 of this thesis.

†See section 2.1.
CHAPTER 2: LITERATURE REVIEW

2.4 Implementation Philosophy

In review, the main goals of this thesis are to investigate the use of local control of search and the use of consistency techniques in a simple connection graph implementation. The philosophy of object oriented programming meshes well with a connection graph implementation in general and with these goals in particular. The objective of this section is to review some of the salient concepts and philosophies of object oriented programming in order to prepare the reader for the discussion of the implementation of connection graphs which follows in section 3.3.1 and for the discussion of the connection graph search strategies in section 3.3.2.

2.4.1 Object Oriented Programming

Pascoe [1986] gave a succinct overview of the elements of object oriented programming. Goldberg [1983] provided detailed discussion of the principles and usage of Smalltalk-80, a prototypical object oriented language. These sources can be consulted for a more detailed overview of object oriented programming.

The objects in an object oriented programming language are the conceptual entities of the domain with which the program deals. For example, integer, real, and complex numbers are some of the conceptual entities in the domain of arithmetic. They are thus represented as classes of objects in an object oriented arithmetic system. Such an arithmetic system is described in Abelson [1985]. A by product of using objects or conceptual entities is that it forces programmers to consider the problem at a high conceptual level and identify common structures in the representation of the problem.
CHAPTER 2: LITERATURE REVIEW

In most object oriented programming languages, there are two types of objects: classes and instances. Class objects describe the common attributes of a group of objects. For example, all circles have a center and a radius. Instances are identifiable elements of a class. Instances of a class differ in the specific values of their attributes. Two instances of the class 'circle' for example can have the same center but differ in the value of their radii.

Objects have internal states represented by the values of their instance variables (attribute names). Their internal state is private in the sense that no object can perform operations on, or change the instance variables of any object other than itself. Each object has access to methods for performing operations on its own internal state. Methods are common to all instances of a class of objects. The methods are activated by messages received by an object and are required to compute a response to that message. In doing so, a method may send further messages to other objects and/or perform operations on the object's internal state. Every object then, provides only a message interface to the rest of the objects in the system. This approach forces the programmer to create modular code which is easily modified [Snyder 1986]. The communication between objects in the form of messages and responses, form the basis for computation in an object oriented language.

Inheritance is another integral aspect of object oriented programming. Inheritance allows for hierarchical classification of objects - a powerful data abstraction technique. Subclasses share the properties of their parent class (instance variables and methods) and they impose some specialization on the superclass (additional instance variables and methods). Inheritance however, is the most controversial aspect of object oriented programming. Smalltalk-80
CHAPTER 2: LITERATURE REVIEW

requires that classes have at most one superclass [Goldberg 1983]. This can lead to duplication of code when an object needs methods from two superclasses but is only permitted to inherit from one. Thinglab [Borning 1981] and Flavors [Moon 1986] allow multiple superclasses. This is a problem if there are name conflicts in the instance variables or the methods in any two superclasses. Another issue is whether methods should be inherited when a new class is created, a problem if the method is later changed in the superclass. Alternately the superclass hierarchy could be searched at runtime when a message is received which requires the use of a superclass's method, obviously a less efficient approach. Problems with inheritance are at the heart of current object oriented programming language development [Lieberman 1986], [Mittal 1986], [Snyder 1986].

Another aspect of inheritance which does not present such problems is that of inheriting an internal state from an existing instance to a newly created instance. This is known as prototyping and is quite well understood. Prototyping is used as the basis for inheritance in the Actors language [Lieberman 1986] and is available in other object oriented languages [Goldberg 1983].

Concurrency in object oriented programming is a relatively new area of study. Shapiro [1983] described how Concurrent Prolog can be used as an object oriented language, giving rise to a powerful new programming technique called incomplete message streams. The technique involves multiple objects (usually in a producer/consumer relationship) sharing a logic variable. The variable acts as a communication stream between the objects. The consumer can not instantiate the variable. As the producer partially instantiates the variable, the consumer becomes aware and can invoke a method depending on the partial instantiation of
the variable. This is a simple form of message passing except that it makes message paths explicit and provides a built-in method for dealing with synchronization of concurrent processes in an object oriented programming language.

Most of the concepts of object oriented programming have straightforward application in the implementation of a connection graph system. Furthermore, it is an appealing approach to implement a distributed search technique by using message passing between a group of cooperating and competing objects. For these reasons, the connection graph implementation used in this thesis is based on the object oriented programming paradigm.
Chapter 3

The focus of this chapter is methods for improving the procedural adequacy of Horn clauses through the use of the connection graphs. In section 3.1, we show a flexible search suitable for a specific knowledge representation task which can be brought into the connection graph framework. In section 3.2, the suitability of constraint satisfaction techniques for connection graphs is explored. Finally, an object oriented implementation of the connection graph proof procedure is described in section 3.3.

3.1 Flexible Search in Connection Graphs

One of Mackworth's [1987] criteria for a procedurally adequate knowledge representation system is that it be able to use knowledge in a flexible manner. The search strategies for connection graphs discussed in section 2.3 were aimed at improving the efficiency of theorem proving in connection graphs rather than for search in knowledge representation. In this section we show how the principles of Havens' hybrid search strategy for visual recognition can be used in connection graph search.
3.1.1 Matching

In the matching phase of Havens' search, cues in the data are matched against instances of models in the knowledge base. This phase of the search is strictly bottom-up. Facts about the components of a particular instance are gathered passively in a data-driven fashion. In connection graphs, the data objects are the known facts. That is, ground clauses with empty bodies. These will be referred to as data clauses. In order to drive bottom-up search in connection graphs, only links associated with data clauses are chosen for resolution. Literals in data clauses will resolve with the literals in the bodies of other clauses in the KB. Thus the subgoals of any particular clause are solved passively by the general search mechanism. However, bottom-up search cannot be used exclusively for reasons mentioned above.

The central idea of Havens' approach is to use data-driven (bottom-up) search while there is insufficient reason to expect any particular object to appear in an image. Then when the data-driven search begins to point towards a particular object being present, the focus of the search changes to confirm the presence of that object. It is reasonable to use goal-driven (top-down) search for the confirmation process since there is a reasonable expectation that the goal will be achieved.

Top-down search is instigated by an instance of a model when some of its expectations† have been met by data cues in the passive phase of the search. The instance has the capability to execute a special local search procedure upon recognition of any of its components. The local procedure first determines if

†see definition below.
sufficient evidence has been collected to justify active search for the rest of the components and if so how that search should proceed. This type of search can be used in a logic programming system by associating local procedures with each subgoal of each clause plus a default bottom-up search strategy. When a particular subgoal is solved via passive bottom-up search, it activates its associated local procedure which can then instigate top-down search by immediately pursuing the other subgoals in the clause using the standard left to right approach or using some recommendation list. This falls into the realm of using meta-knowledge in search and is discussed further in section 3.3.1.

In Havens' technique, it was assumed that the components would be recognized in some order. This makes it possible to know in advance, the context in which a particular component will be found. This in turn makes it possible to attach very precise local search procedures to the model indicating how the search should continue when a particular component is recognized. In visual recognition however, it may be unreasonable to assume that components will be recognized in any particular order given that the data is itself unordered and the search is at first data-driven. Thus it is also unreasonable that the local search procedures can be very precise, since the conditions under which a component is recognized are not predictable. This argues for activating top-down search based on general criteria rather than on special local conditions. It also suggests using a general confirmation search rather than a specific one linked to the recognition of a particular component.

In the hybrid approach "Top-down search is best viewed as useful for confirming the last details of an instance after enough evidence has been collected to be confident of success". But if there are no local procedures to determine when
to shift to top-down search, there must be some general procedure which can determine when that shift should be made for a given goal. Thus it is important to establish some way of measuring the degree to which a goal has been solved. In logic programs, there are several ways to do this. The simplest is to determine the ratio of solved to unsolved subgoals and to switch the direction of the search when the ratio exceeds some given value for a particular clause. This method could be extended if the subgoals are weighted in some way such that search will more readily shift to top-down when the more important subgoals are solved. The weighting could be given by the user or determined automatically in relation to the complexity of terms in the subgoals. Other factors which can be used to determine when to switch to top-down search include the degree of instantiation of the variables in the remaining sub-goals, or the number of links between the remaining sub-goals and the ground literals in the data base.

In connection graphs, the overall bottom-up approach is implemented by choosing to resolve against data clauses only. Top-down search can be implemented in the usual way with a goal stack. When a shift to top-down search is made, the appropriate goal(s) are placed on the goal stack and link selections from those goals supersede the general approach. If the goal stack becomes empty, the general bottom-up strategy regains control. In addition, the connection graph representation readily provides the information needed to decide when to switch to top-down search. A simple implementation of the hybrid search technique in connection graphs is described in greater detail in section 3.3.2.
3.1.2 Completion

In the completion phase of Havens' search, instances of models which are fully recognized become data cues. Since the original data cues are given and the recognized objects are inferred, they are called concrete and abstract cues respectively. By using a hierarchy of cues and models, the recognition system does not have to invoke high-level models based on low-level cues. By ensuring that an instance is completed before it is used as an abstract cue, the system avoids making hypotheses based on incorrect information.

Completion is achieved automatically in the connection graph search described above. The default bottom-up mechanism selects only links to data clauses. A clause which has not yet had all of its sub-goals solved is not a data clause and will thus not be considered for use in bottom up search. However, once all of the literals in the body of a clause have been resolved (either by bottom-up or top-down methods) the head will become a data clause and the general strategy will recognize it as being eligible for use in bottom-up search. In other words, when a clause is completed by the solution of all of the sub-goals in its body, it becomes a data clause which can be used to solve subgoals in the bodies of other clauses.

3.1.3 Expectations

The expectations for an instance of a model are the possible ways in which an unmatched components of the instance can still be matched by the data. Any data cue which matches an instance must necessarily fall within the expectations of that instance. Expectations are of central importance to the search strategy. In
the top-down phase of search, the expectations provide a boundary within which
the search must take place. Thus, better expectations bound the search more
tightly. In bottom-up search they determine which data cues match a partially
instantiated object. Thus, better expectations will allow fewer false matches. In
addition, the degree to which an instance's expectations have been fulfilled can be
used to determine the point at which an instance is sufficiently instantiated to
invoke top-down search.

However, computing the expectations for a particular instance may be a
difficult task. The expectations for the uninstantiated portion of an object are
constrained by the instantiated portions and thus must be recomputed whenever
the object becomes further instantiated. The recomputation of expectations is an
expensive process "perhaps entailing an encyclopaedic tour of the knowledge
base" [Havens 1983]. To circumvent the problem of recomputing expectations
regularly, Havens suggested the use of static and dynamic expectations. Static
expectations are pre-compiled for the every component of every model and do not
change as an instance of a model becomes instantiated. However, it is possible
that erroneous compositions will be allowed when static expectations are used
[Havens 1983]. Dynamic expectations must be recomputed whenever the state of
the instance changes. In order to avoid this, Havens suggests that dynamic
expectations only be used in conjunction with a contextual mechanism. In this
way he is able to avoid computing the expectations for an uninstantiated
component of an object until the instantiated portion of the object provides an
appropriate context. This usage assumes that the components will be found in
some order, a questionable assumption in visual recognition.
A third approach is to weakly recompute expectations. The simplest way to do this is to ensure that all expectations are consistent with the currently established constraints but not require that expectations for uninstantiated portions of an object be mutually consistent. In this way the recomputation of expectations can be performed locally. In addition, no erroneous compositions will be allowed, nor does the system require ordered input. However, the set of expectations for the uninstantiated portion of the model may not be minimal and therefore search may not be as efficient as if expectations were fully computed.

In connection graphs, the expectations of a particular clause are determined by the ways in which the logic variables in that clause can be instantiated. The set of unifiers associated with the links to each clause explicitly represent all the ways in which the logic variables can be instantiated in a given problem. Thus the links associated with a clause represent the expectations of the clause. When resolution takes place between a literal in the body of a clause and some ground literal, a new, more instantiated version of the clause appears in the resolvent. The variable bindings established in the resolution may restrict the way in which the other variables can be bound. In other words, the expectations of the clause are constrained by the resolution. Only the links which are compatible with the resolved link are inherited by the resolvent. Thus, the expectations of the new version of the clause are consistent with the new constraints established by the resolution.

Computing expectations in a knowledge representation system is strongly related to constraint satisfaction. In Havens' system, as an object becomes further instantiated the expectations are constrained by the instantiated portion of that object. Constraint satisfaction algorithms can be used to recompute the
expectations of the remaining portions of the object by removing expectations which are not consistent with internal constraints and the values established for the instantiated portion. The expense of computing better expectations increases rapidly with the use of stronger constraint satisfaction techniques. On the other hand, tighter expectations produce tighter bounds on search, providing efficiency gains. The use of constraint satisfaction in connection graphs will be discussed further in 3.2.

Havens recursive recognition cycle is summarized in Figure 12 which is reproduced from Havens [1983].

---

Figure 12: Recursive recognition cycle.
3.1.4 Summary

Flexible search is an important aspect of procedural adequacy in a knowledge representation system. Havens' hybrid search for visual recognition is such a flexible search strategy. In standard logic programming systems, the automatic control is too inflexible to allow this type of search even with the extra-logical control primitives available in some systems. In this section we have shown how the principles of Havens' search can be naturally brought into a connection graph system. The incorporation of this hybrid search in a simple connection graph implementation is described in section 3.3.2.

3.2 Constraint Satisfaction in Connection Graphs

The use of the Horn clause subset of full first order logic for knowledge representation is a conscious decision to reduce the descriptive capacity of a KR formalism in exchange for increased procedural adequacy. However, if this approach is taken to the extreme the result is a formalism like the database form where inference is very efficient but representational capacity is too restricted. Mackworth [1987] argued that “many interesting tasks seem to be inherently exponential [so] the strategy of weakening descriptive adequacy to achieve efficient (polynomial) algorithms is not available”. He suggests that an alternative is to use efficient approximation algorithms which may not provide solutions to a problem but may be used to eliminate many of the non-solutions. Constraint satisfaction algorithms fall into this category. In this section, the use of
constraint satisfaction algorithms in the connection graph representation of logic programs is examined in detail.

The operation of Kowalski's link filtering algorithm on the connection graph representation of constraint satisfaction problems is analysed by comparing results obtained for it to results for the well known k-consistency algorithm. To do this, we first introduce a new algorithm, called the M-F algorithm, which is an extensively modified version of Freuder's k-consistency algorithm. We show that the M-F algorithm is analogous to Kowalski's link filter except that it operates in the constraint graph domain. We then provide some theoretic results for the M-F algorithm (and by analogy for the link filter) so that we are able to understand the nature of the link filter in terms of k-consistency. In particular, we show that given a constraint satisfaction problem in which the global constraint is partially specified, the M-F algorithm finds all solutions to the constraint satisfaction problem. We also show that the level of k-consistency computed by the M-F algorithm depends on the arities of the given constraints. By analogy the link filtering algorithm computes the equivalent of k-consistency for k=n on some constraint satisfaction problems and may compute the equivalent of k-consistency for k>2 on others. Thus, the link filtering algorithm is not a proper reduction operation even for the relatively small subset of connection graph problems which are also constraint satisfaction problems. In addition, it is shown that the link filter does not provide the well characterized intermediate results of the k-consistency algorithm. To conclude the section, a variation on the link filtering algorithm is proposed which makes the algorithm perform a proper reduction operation but which necessarily reduces the power of Kowalski's original algorithm.
3.2.1 An Example

Before pursuing the analysis of the link filtering algorithm, we present a motivating example which demonstrates the similarity between constraint satisfaction and link filtering. Constraint satisfaction problems can easily be represented as simple logic programming problems - and hence as connection graphs. Figure 13 shows the constraint graph, connection graph and logical representations of a simple constraint satisfaction problem.
Notice that all clauses except the goal clause in the logic program representation are ground unit clauses. Also, there are no cycles in the connection graph and all of the logic variables occurring in unifiers in the connection graph are bound to constant terms. The operation of the k-consistency algorithm for \(k=2\) on the constraint graph in Figure 13a is compared with the operation of Kowalski’s filtering algorithm on the corresponding connection graph in Figure 13b.

The link filtering algorithm removes any inconsistent links from the connection graph in Figure 13b. Consider the links to the literal \(P_x(x)\). The link labeled \(a/x\) is compatible with both links to the literal \(P_y(y)\). If two literals do not relate the same variable(s), then any links to those literals are compatible (see the definition of link compatibility in 2.1.2). The link \(a/x\) is also compatible with the link \(a/x, c/y\) to the literal \(P(x, y)\). Similarly, the link \(b/x\) and both links to the literal \(P_y(y)\) are compatible with at least one link to every other literal in the clause. However, the link labeled \(b/x, c/y\) to literal \(P(x, y)\) is not compatible with any link to literal \(P_y(y)\) and is thus removed. Similarly the links labeled \(c/x, d/y\) and \(a/x, c/y\) are be removed. This causes the links \(b/x\) to the literal \(P_x(x)\) and \(d/y\) to \(P_y(y)\) to become inconsistent and thus they are removed during propagation. Many of the unit clauses in the graph have become pure as a result of the link deletions and are thus also removed from the graph resulting in the connection graph in Figure 14b.

At step \(k=1\) of Freuder’s algorithm a constraint graph is constructed containing only the unary constraints. At step \(k=2\) the binary constraint \(P\) is added resulting in the graph in Figure 13a. No constraints are synthesized since all unary and binary constraints are given. As the node for the constraint
relation P is added to the graph, the elements ac bc and cd are removed since they do not satisfy \( Px \) and \( Py \) together. During the propagation phase, the element b is removed from \( Px \) and c is removed from \( Py \) since they no longer satisfy the adjacent constraints. The resulting graph is presented in Figure 14a.

![Diagram](image)

**Figure 14:** a) 2-consistent version of Figure 13a and b) is Figure 13b after the link filtering operations. Analogous reductions have been made to both graphs.

From this example, it appears that the operations performed by Kowalski's link filtering algorithm on the connection graph representation of a simple constraint satisfaction problem are analogous to those performed by the k-consistency algorithm for \( k=2 \) on the constraint graph. This relationship is further explored in the following sections.
3.2.2 The M-F Algorithm

The terminology relevant to this section was introduced in section 2.2.4.1. Recall that the order n constraint expression C for a constraint satisfaction problem is conjunction of the constraints C_J for all I \supseteq J such that |I| = n. An element a_I of the global constraint C_I is said to satisfy C if it satisfies every constraint C_J in C. Recall also that a graph is said to be k-consistent iff given any instantiation of any (k-1) variables satisfying all the direct constraints between those variables, there exists an instantiation of any k'th variable in the constraint graph such that the k values taken together satisfy all the constraints among the k variables.

The M-F algorithm (Figure 15) is presented below. It is an extensively modified version of Freuder's algorithm which does not synthesize constraints. In section 3.2.3 it is shown that the M-F algorithm and Kowalski's link filtering algorithm perform analogous consistency preprocessing operations for standard constraint satisfaction problems. The operation of the M-F algorithm is then compared to Freuder's k-consistency algorithm in order to establish which aspects of the k-consistency algorithm which it does and does not capture. This allows us to understand the operation of the link filter in terms of k-consistency.

Nodes corresponding to a constraint C_J in the constraint graph constructed by the M-F algorithm are denoted R_J to differentiate them from the nodes in a constraint graph constructed by the k-consistency algorithm. In the M-F algorithm, when a node R_J corresponding to constraint C_J is added to the graph, it is linked to every R_H where H \cap J is non-empty. No constraints are synthesized
by the M-F algorithm. That is, if constraint $C_J$ is not given, no node $R_J$ is added to the constraint graph.

Since two nodes $R_H$ and $R_J$ are joined by the M-F algorithm whenever $H \cap J$ is non-empty, it is necessary to extend the definition of satisfiability. In the M-F algorithm an instantiation $a_J$ of constraint $C_J$ is said to satisfy an adjacent constraint $C_H$ if there is an $a_H$ such that $\{a_j \in a_H\}_{j \in (J \cap H)} = \{a_j \in a_J\}_{j \in (J \cap H)}$. That is, the instantiations of the variables common to $X_J$ and $X_H$ are the same in $a_J$ and some $a_H$. Note that this definition of satisfiability is equivalent to Freuder's in the special cases when $J \supseteq H$ ($J \cap H = H$) or $H \supset J$ ($J \cap H = J$).

---

step 1) Graph construction: Starting with an empty graph, for each given constraint $C_J$, add a node $R_J$ corresponding to the constraint $C_J$ to the graph and link $R_J$ to all $R_H$ such that $H \cap J$ is non-empty.

step 2) Constraint propagation: For each $R_J$, delete any $a_J$ which does not satisfy a neighbouring constraint.

step 3) Repeat step 2 until there are no deletions.

---

Figure 15: The M-F algorithm.

---

3.2.3 Connection Graphs and k-consistency

In this section we make an analogy between the operation Kowalski's link filter and the operation of the M-F algorithm on standard constraint satisfaction problems.
The clausal representation of a constraint \( C_J \) is a ground predicate \( P_J \). For each \( a_J \) in \( C_J \) there is one ground unit clause \( P_J(a_J) \) in the predicate definition. The constraint expression represented by the set of given constraints in a constraint satisfaction problem is a conjunct of constraint predicates \( \wedge P_J(X_J) \). This expression will henceforth be referred to as the goal clause. Figure 13c is an example of the clausal representation of a constraint satisfaction problem and Figure 13b is the connection graph constructed from that clausal representation.

There is a direct analogy between the connection graph representation of a constraint satisfaction problem and the corresponding constraint graph constructed by the M-F algorithm. In the constraint graph representation, the node \( R_J \) corresponding to a given constraint \( C_J \) contains one element \( a_J \) for each element \( a_J \) in \( C_J \). In the connection graph representation there is a link between the literal \( P_J(X_J) \) in the goal clause and each ground literal \( P_J(aj) \) in the predicate definition. The unifier associated with each link is necessarily of the form \( a_{j1}/X_{j1}, \ldots, a_{j2}/X_{j2}, \ldots, a_{jm}/X_{jm} \) where \( |J| = m \), that is, the instantiation \( a_J \) of \( X_J \). Thus there is a one-to-one correspondence between the elements of the nodes in the constraint graph and the links in the connection graph representation of the same problem.

The links in the constraint graph join any two nodes \( R_J \) and \( R_H \) where \( J \cap H \) is non-empty. These are not explicitly represented in the connection graph but can easily be constructed by comparing the variables occurring in the literals in the goal clause.

Recall the definition of incompatible unifiers from section 2.1.2. A link to some literal in a clause is said to be consistent with respect to a second literal in
the same clause if that link is compatible with some link to the second literal. The link filter operates by removing links which are not consistent with respect to every literal in the goal clause.

In the following theorem we show that the M-F algorithm and Kowalski's link filter perform analogous operations on their respective representations of standard constraint satisfaction problems. In order to do this we first prove a proposition which asserts that the criteria used by the algorithms for deleting analogous structures are equivalent. That is, when an element $a_J$ is fails to satisfy a neighbouring node $R_H$ in the constraint graph, the link corresponding to $a_J$ is inconsistent with respect to the literal $P_H(X_H)$ in the connection graph representation of the same problem and vice versa. The subsequent proof of Theorem 1 establishes the analogous behaviour of the two algorithms for standard constraint satisfaction problems.

**Theorem 1:** The link filtering algorithm and the M-F algorithm perform analogous deletion operations on their respective representations of a given constraint satisfaction problem. More precisely, the element $a_J$ is deleted from the node $R_J$ by the M-F algorithm iff the link between $P_J(X_J)$ and $P_J(a_J)$ is deleted by the link filtering algorithm in the respective representations of the same constraint satisfaction problem.

**Proposition 1.1:** A link between $P_J(X_J)$ and $P_J(a_J)$ in the connection graph representation of a constraint satisfaction problem is inconsistent with respect to some literal $P_H(X_H)$ in the goal clause iff the element $a_J \in R_J$ does not satisfy some neighboring constraint $R_H$ in the constraint graph representation of the same problem.
Proof: It is necessary to show that 1) if the link between \( P_j(X_j) \) and \( P_j(a_j) \) is inconsistent with respect to some literal \( P_H(X_H) \) in the goal clause, then the element \( a_j \in R_j \) does not satisfy some neighboring constraint \( R_H \) and 2) the converse.

Part 1) We prove the contrapositive, that is, if \( a_j \in R_j \) satisfies a neighbouring constraint \( R_H \), then link between \( P_j(X_j) \) and \( P_j(a_j) \) is consistent with respect to some literal \( P_H(X_H) \). Assume that the element \( a_j \in R_j \) satisfies a neighbouring constraint \( R_H \). Then by definition there exists an \( a_H \in R_H \) such that \( \{a_j \in a_H\}_{j \in (J \cap H)} = \{a_j \in a_J\}_{j \in (J \cap H)} \). That is, the instantiations of the variables common to \( X_j \) and \( X_H \) are identical in \( a_j \) and \( a_H \). In the connection graph representation of the problem, \( a_j \) represented by the link between \( P_j(X_j) \) and \( P_j(a_j) \) and \( a_H \) is represented by link between \( P_H(X_H) \) and \( P_H(a_H) \). Let \( \theta_1 \) and \( \theta_2 \) respectively be the unifiers associated with those links. Since the instantiations of the variables common to \( X_j \) and \( X_H \) are identical:

\[
\theta_1 = \{a_1/X_1, \ldots, a_n/X_n, a_{1_1}/X_{1_1}, \ldots, a_{1_m}/X_{1_1} \}
\]

\[
\theta_2 = \{a_1/X_1, \ldots, a_n/X_n, a_{2_1}/X_{2_1}, \ldots, a_{2_m}/X_{2_2} \}
\]

where \( \{X_1, \ldots, X_n\} = X_J \cap X_H \) and no \( X_{1_1} \) is the same as any \( X_{2j} \). Note that all variables in \( \theta_1 \) and \( \theta_2 \) are necessarily instantiated to constant terms in a connection graph representation of a constraint satisfaction problem because all constraint definitions are ground. From \( \theta_1 \) and \( \theta_2 \) we construct the expressions:

\[
E_1 = (X_1, \ldots, X_n, X_{1_1}, \ldots, X_{1_1}, \ldots, X_n, X_{2_1}, \ldots, X_{2_m})
\]

\[
E_2 = (a_1, \ldots, a_n, a_{1_1}, \ldots, a_{1_1}, \ldots, a_n, a_{2_1}, \ldots, a_{2_m})
\]

\( E_1 \) and \( E_2 \) must be unifiable since any variable which occurs twice in \( E_1 \) is unified with the same constant from \( E_2 \) in both cases. Any variable which occurs only once in \( E_1 \) is unified with a constant term from \( E_2 \). Therefore, the link corresponding to \( a_j \) is compatible with a
link to the literal \( P_H(X_H) \) and hence it is consistent with respect to the literal \( P_H(X_H) \).

Thus, if the link between \( P_J(X_J) \) and \( P_J(a_J) \) is inconsistent with respect to some literal \( P_H(X_H) \) in the goal clause, then the element \( a_J \in R_J \) does not satisfy some neighboring constraint \( R_H \) in the respective representations of the same constraint satisfaction problem.

**Part 2** Proof of the converse is similar. Again proving the contrapositive, assuming that the link corresponding to \( a_J \) is compatible with some link to the literal \( P_H(X_H) \) leads to the conclusion that the element \( a_J \) satisfies a neighbouring node \( R_H \) in the constraint graph representation of the problem.

**Proof of Theorem 1:** Proposition 1.1 shows that the the criteria used by the two algorithms for deleting corresponding elements in the analogous representations of the same problem are identical. Since both algorithms perform analogous constraint tests and use identical constraint propagation, it follows that both algorithms perform analogous deletion operations on their respective representations of the same constraint satisfaction problem.

With this result established, we continue by providing some theoretic results for the M-F algorithm (and by analogy for the link filter). In the following theorem and its corollaries, we establish the relationship between the M-F algorithm and the k-consistency algorithm. Our goal is to characterize the types of constraint satisfaction problem which can be solved by the M-F algorithm and describe the type of problem for which the M-F algorithm computes intermediate degrees of k-consistency. We also describe the aspects of the k-consistency algorithm which are not achieved by the M-F algorithm. In this way we are able
to characterize the behaviour of the link filter for standard constraint satisfaction problems in terms of the well understood k-consistency algorithm.

**Theorem 2**: In a constraint satisfaction problem in which the n'ary constraint $C_i$ is partially specified, the node $R_I$ in the constraint graph constructed by the M-F algorithm contains, after propagation, exactly those elements $a_I$ which occur in $C_I$ and which satisfy the order n constraint expression $C$, thus computing k-consistency for $k=n$.

**Proof**: It is necessary to show 1) that all $a_I$ occurring in $C_I$ which satisfy the order n constraint expression $C$ are in $R_I$ after constraint propagation and 2) that after constraint propagation all $a_I \in R_I$ also occur in $C_I$ and satisfy the order n constraint expression $C$.

**Part 1**  An element $a_I \in R_I$ can only be deleted if it fails to satisfy some node $R_J$ after some finite number of steps of constraint propagation, thus the proof is by induction.

*Base case*: If $a_I$ satisfies $C$, it must satisfy all given constraints, so it must satisfy all nodes $R_J$ in the initial constraint graph.

*Inductive hypothesis*: If an element $a_I$ satisfies $C$, it satisfies all nodes $R_J$ after $i$ steps of constraint propagation.

*Induction*: Assume that $a_I$ satisfies $C$ but that it does not satisfy some node $R_J$ after $i+1$ steps of constraint propagation. This must be due to the earlier removal of some single element $a_J$ in $R_J$ for which $a_I$ satisfied $R_J$. Thus, $a_I$ shares the same instantiation for the variables $X_J$ as does $a_J$. Let the step at which $a_J$ was removed be step $h$, $h \leq i$. The element $a_J$ must have been removed because it did not satisfy some neighbouring node $R_H$. Let $K$ be $J \cap H$ ($K \neq \emptyset$ since $R_J$ neighbours $R_H$). So, $a_J$ was removed because no element in $R_H$ contained the same instantiations of the variables $X_K$ as did $a_J$ at step
CHAPTER 3

h. But \( a_i \) contains the same instantiation of \( X_K \) as does \( a_j \) since \( X_I \triangleright X_J \triangleright X_K \). Thus no element in \( R_H \) contained the same instantiations of the variables \( X_K \) as did \( a_i \) at step \( h \) and since \( K \) is non-empty, \( a_i \) could not satisfy \( R_H \) at step \( h \), contradicting the inductive hypothesis.

Since \( R_I \) initially contains the same elements as \( C_I \), and by the inductive proof above, all \( a_i \) occurring in \( C_I \) which satisfy the order \( n \) constraint expression \( C \) are in \( R_I \) after constraint propagation.

Part 2) Since \( R_I \) initially contains the same elements as \( C_I \) and no elements are added to \( R_I \), it follows that all elements in \( R_I \) after propagation also occur in \( C_I \). Any \( a_i \in R_I \) trivially satisfies any non-constraints in \( C \). If an \( a_i \) does not satisfy a given constraint \( C_J \) then there is no \( a_j \in C_J \) such that \( a_j = \{ a_i \in a_j \}_{i \in J} \). But during constraint propagation, the M-F algorithm would necessarily delete such an \( a_i \) from \( R_I \) for it would not satisfy the neighbouring node \( R_J \). Thus every \( a_i \in R_I \) satisfies every \( C_J \) in \( C \). Thus, after constraint propagation all \( a_i \in R_I \) also occur in \( C_I \) and satisfy the order \( n \) constraint expression \( C \). ■ End of Theorem 2.

Corollary 2.1: If a constraint \( C_J \), \(| J | = k \) is specified, then the M-F algorithm computes \( k \)-consistency among the variables \( X_J \).

Proof: If one considers the given constraint \( C_J \) and all the given constraints \( C_H \) such that \( J \triangleright H \) as a separate sub-problem with \( C_J \) as the global constraint, then by result 2 the M-F algorithm computes \( k \)-consistency among the variables \( X_J \). The additional constraints in the larger problem can not cause inconsistent elements to be added to the constraint \( C_J \). ■

Corollary 2.2: If all possible constraints \( C_J \) such that \(| J | = k \) are specified, the M-F algorithm computes \( k \)-consistency for the constraint satisfaction problem.

71
Proof: If all constraints $C_j$ such that $|J| = k$ are specified then by corollary 2.1 the M-F algorithm computes $k$-consistency within each $k$-tuple of variables in the constraint satisfaction problem. By definition then, the M-F algorithm computes $k$-consistency for the entire problem. ■

These results and corollaries about the M-F algorithm apply also to Kowalski's link filter by the analogy established in Theorem 1. We now use the results to analyse the link filtering algorithm in terms of Freuder's $k$-consistency algorithm.

The first and perhaps most critical issue is that consistency algorithms are intended to be fast preprocessing operations which will operate on data structure in order to reduce subsequent search. It is therefore prerequisite that such operations be proper in the sense that they can be performed in polynomial time. Kowalski's link filtering algorithm computes $k$-consistency in connection graph representations of some constraint satisfaction problems (Theorem 2), and high degrees of partial consistency in others (Corollary 2.2). The operation performed by the link filter are proper in the sense that it generally reduces the size of the graph and that no solutions are lost. However, the operation is not proper in that it can not always be performed in polynomial time.

The complexity Freuder's $k$-consistency algorithm is too great to make that algorithm of use for consistency preprocessing just as is the case for the link filter. However in the case of Freuder's algorithm, useful intermediate results can be obtained by running the algorithm for a limited number of steps. In particular, it can be used to compute $k$-consistency for $k = 1, \ldots, n$ (where $n$ is the number of variables in the problem). For small values of $k$, the consistency
preprocessing done by this algorithm are in fact proper reduction operations [Mackworth 1985a]. This is corroborated by the fact that many AI researchers have independently developed and used algorithms which essentially compute 2-consistency and 3-consistency in idiomatic representations of constraint satisfaction problems. In several existing systems ([Havens 1985], [Mackworth 1977], [Montanari 1974], [Waltz 1972]), constraint preprocessing is used but in none of them is k-consistency computed for k>3. Haralick's [1980] empirical result showed that the overhead involved in computing full arc consistency may not even be worthwhile in many constraint satisfaction problems. Therefore, a second major problem with the link filter is that does not guarantee these intermediate degrees of consistency except under the special conditions noted above. Also, when the M-F algorithm does guarantees k-consistency for k=m in a particular problem, it does not guarantee strong k-consistency. That is, it guarantees k-consistency for k=m but not for k=i where i=1,...,m-1. Thus, the link filter does not generally provide the important intermediate results produced by the k-consistency algorithm and other consistency algorithms.

With these problems in mind, we present some approaches to revising the link filtering algorithm in section 3.2.4. Implications of these approaches for consistency preprocessing in constraint graphs and logic programs are discussed in section 4.2.
3.2.4 Revising the Link Filtering Algorithm

A revised link filtering algorithm should attempt to make use of the constraint information in all of given constraints, yet it should not compute the same degree of consistency which Kowalski's computes.

**Approach 1:** One method is to restrict the propagation phase of the link filtering algorithm. For example, each link could be tested once for compatibility with the other literals in the goal clause but no link would be re-tested for consistency after the first deletions have taken place. This approach is too *ad-hoc* in that the ordering of literals in the goal clause could have a dramatic effect on the outcome of the processing. It would thus be impossible to specify exactly what the algorithm was computing without reference to consistency procedure. In addition it would be difficult to assess the performance of such an algorithms since it would perform differently on alternate formulations of the same problem.

**Approach 2:** The simplest modification which reduces the strength of the link filtering algorithm would be to restrict the consistency checking to those links to literals of some fixed arity. In fact, if links to only unary and binary literals were checked against each other for consistency, then the link filtering algorithm computes 2-consistency in the connection graph representation of a constraint satisfaction problem. However, it seems unreasonable that in general problems will be specified with a sufficient number of unary and binary constraints to make such preprocessing useful. In addition, none of the constraint information from constraints of degree > 2 would be used by the link filter at all.

**Approach 3:** In order to make use of all of the given constraints, the link filter could be revised to compare links of all given constraints with those links to
the binary and unary constraints only. The motivation for such an approach is that it considerably reduces the upper bound on the amount of computation done in order to eliminate a link based on consistency. In particular, suppose the removal of links to the literal $P_J(X_J)$ with respect to the literal $P_H(X_H)$ is being considered. Let $K = H \cap J$ and $k = |K|$. Suppose that the domains of all the variables in the problem are of size $m$. There are $m^k$ possible instantiations of the variables $X_K$ in both $P_J$ and $P_H$. In the link filtering algorithm therefore, $m^{2k}$ compatibility checks may be made (each of which compares the unifiers of $k$ variables) in testing all the links of $P_J(X_J)$ for consistency with respect to $P_H(X_H)$. The restriction proposed in this approach limits $k$ to being 1 or 2 making any reductions to the graph proper in both senses. In cases where more computation may be necessary to remove inconsistent links, the algorithm does not expend any effort attempting to do so. This approach also guarantees that the link filter will compute 2-consistency.

These modifications to Kowalski's link filtering algorithms are presented as alternative methods for applying constraint satisfaction techniques in connection graphs without the drawbacks encountered by the original link filter. Like the consistency techniques used in other systems, it will not always be the case that the link filter or its alternates will reduce the computation in the solution of constraint satisfaction problems. The merits of consistency techniques for connection graphs would have to be assessed for assorted problem types and a set of performance metrics would have to be developed. In addition, it would be necessary to characterize the behaviour of any connection graph consistency preprocessing operation for general connection graph problems rather than the restricted subset which make up the standard constraint satisfaction problems.
These algorithms have not been incorporated in the connection graph implementation discussed in other sections of this thesis.

### 3.3 Implementation

In this section, an object oriented connection graph implementation is described. The method of implementing a connection graph system is outlined in section 3.3.1. This is followed by a description of search strategy implementation in object oriented connection graphs in section 3.3.2. Finally, some of the details of the implementation are discussed in section 3.3.3.

#### 3.3.1 Object Oriented Connection Graphs

Object oriented programming has its roots in the SIMULA-67 programming language [Goldberg 1983]. That language was designed for simulating physical systems. Since then, physical system simulation has remained one of the main applications of object oriented programming [Abelson 1985], [Borning 1981], [Goldberg 1983], [Sussman 1980]. In a physical system simulation, the physical objects are modeled by computational objects and system actions are modeled by symbolic actions on the computational objects.

The connection graph proof procedure closely parallels a physical system. The nodes and links correspond to physical objects. Selecting a link, creating a resolvent, and reducing the graph can be thought of as physical operations on the graph. Exploring the search space corresponds to traversing the edges on the graph, again easily viewed as a physical process.
An object oriented implementation of a connection graph system takes advantage of some of the best understood aspects of object oriented programming. The objects in a connection graph are the components of the graph: the nodes (literals), the node partitions (clauses), the links and the unifying substitutions (mgus) which are associated with the links. In addition, the logical terms which occur in literals and mgus are also objects as is the connection graph itself. A particular connection graph is composed of instances of each of these classes of objects. The internal state of each of the classes is defined with a few instance variables. For example, the class literal is defined with two internal state variables, one is the sign (+/-) of the literal the other is a logical term. Logical terms are also objects. The internal state of a logical term is either a logical variable or a list of arguments (which are themselves logical terms) plus a predicate or functor name.

It is straightforward to represent the components of a connection graphs as objects with internal state. However, the operations on the connection graphs must also be implemented. These operations include constructing the graph, resolving on a selected link and adding the resolvent to the graph. The selection of links for resolution and graph reduction operations are considered as part of the search process and can vary depending on the particular application or problem being solved. Therefore, the search operations are considered separately in section 3.3.2.

In an object oriented language computations are performed by methods associated with the objects. The methods are invoked by the reception of messages from other objects. The methods for a particular task may invoke other methods or local function in order to complete the specific details of the task.
The methods for constructing a connection graph are associated with the connection graph object itself. This operation is performed by the initialize method of a connection graph object which requires as an input argument a set of clauses to be represented. This method simply adds each clause in the input set to the graph using the connect-clause method. Connect-clause attempts unification between each literal in the input clause and the literals already in the graph and creates a link between any potentially complementary pair. The connect-clause method does not actually know how to unify two literals or create a new link, rather it knows which two literals to unify and if they are unifiable, that a link must be created. The actual acts of performing the unification and creating new links are delegated to other objects by way of messages.

The task of selecting a link between two potentially complementary literals will be covered in the next section. Once a link has been selected through whatever means, resolution takes place and the resolvent is added to the graph. Resolution itself is trivial in connection graphs since the unification between the linked literals has already been performed during graph construction and the unifying substitution is associated with the link. The construction of a resolvent and the subsequent connection of the resolvent to the rest of the graph are handled by the resolve method associated with the link class. When a link receives a resolve message, it causes a new clause to be created and copies the remaining literals from each of the resolving clauses into the resolvent. The extent to which any copying is actually performed depends on the internal representation for literals being used in the implementation. If a structure sharing approach [Boyer 1972] is used, little copying is needed but other problems are encountered. (See the discussion of structure sharing in section 3.3.3). It is important to note that
whatever the internal implementation of the literals, the *resolve* method for links will be the same.

In order to connect the resolvent each literal must be linked to every potentially complementary literal in the graph. Since the literals in the resolvent are specialized instances of the literals in the two resolving clauses, all of the potentially complementary literals are identified by the links to the original literals. In other words, the links to the resolvent are specialized instances of the links to the resolving clauses. Thus it is not necessary to make a comprehensive tour of the connection graph or attempt any further unifications in order to link each new resolvent into the graph. Thus the task of connecting the newly formed resolvent to the graph is reduced to the task of determining which links are to be inherited from the resolving clauses. By step 3b) of the connection graph proof procedure, only those links which are compatible with the unifier on the selected link are inherited by the resolvent.

In an object oriented implementation the operation of inheriting instances of a existing objects is particularly simple. The original object is simply sent a *new* message which returns an new instance of the object. Any changes can then be made to the new instance without affecting the original. The main operation involved in forming a new resolvent and connecting it to the graph is the inheritance of literals and links by the new resolvent. This can be performed by sending *new* messages to the appropriate literals and links in the resolving clauses. The test for link compatibility is performed within the link before a new instance is created.

This section has provided an overview of an object oriented implementation of connection graphs. It demonstrates that the structures and operations of in the
connection graph proof procedure can be naturally represented in the object oriented style. In the next sections further details of the implementation are described. For the curious reader, the code for the implementation is attached in the Appendix.

### 3.3.2 Object Oriented Search

The task of search in connection graphs falls into two categories. First there is the selection of links which determine how the solution space is to be searched. Second are the preprocessing operations which can be used to reduce the graph and consequently make large reductions to the solution space. The implementation of both types of operations are amenable to the object oriented style of computation via message passing.

Link selection can be based on any of a number of factors. Several strategies were described in section 2.3 and the object oriented implementation of some of them will be discussed here. Many of the search techniques involve adding instance variables to one or more of the object classes in the system so that control can be kept at the appropriate level.

#### 3.3.2.1 Backtracking

Depth-first, left-to-right search with automatic backtracking is a global search strategy which uses little information from the connection graph itself. In order to implement this strategy the connection graph object is augmented by an instance variable which keeps a goal stack. A stack is in turn an object which
Chapter 3

behaves in the usual way by accepting push and pop messages\(^\dagger\). Further, one of the input clauses must be designated as the initial goal and is placed on the goal stack to begin. In the implementation top-down search was chosen as the default search strategy. Thus, the methods which implement top-down search were named \textit{d-search}. A \textit{d-search} method is associated with the graph, clause and literal objects. In the connection graph, the \textit{d-search} method simply pops a goal clause off the stack and relays the message to that clause. Upon receiving the \textit{d-search} message, the goal clause picks its first (leftmost) literal and again relays the message. When a literal receives the message, it selects one of the links attached to itself (there must be one or else the clause would be pure) and sends it the \textit{resolve} message. The new resolvent is recursively returned to the connection graph where it is pushed on to the goal stack. If the resolution does not cause the original goal clause to become pure through the removal of the selected link, it is replaced in its original position on the stack. Since the resolved link is removed from the connection graph, any subsequent retries of the original goal will not attempt to solve the goal in the same way.

In top-down search, the connection graph representation saves time in that there is no search involved in finding candidates to resolve against a particular goal. On the other hand, the clauses on the goal stack are not represented as compactly as the backtrack points in standard logic programming implementations. In addition, top-down search does not take advantage of any other features of the connection graph representation. It does not make any use of graph reduction techniques nor does it look at the structure of the graph or the problem in order to determine the next resolution to be performed.

\(^\dagger\)See the definition of the stack object in the Appendix.
3.3.2.2 Local Guidance

Local guidance of the search procedure in a connection graph can be based on local properties of the graph. The local methods of the clause and literal objects would have access to precisely this sort of information. Simple methods which evaluate the internal structure of a clause or literal can therefore be used to provide guiding information to the global search strategy. Consider the example of estimating lower bounds for the number of resolutions required to find a refutation involving a particular clause. The clause and literal objects are augmented with methods for estimating this lower bound. The estimate-refutation-length (e-r-l) method takes a single argument for the depth of the estimate. For depth one, the e-r-l method associated with a clause returns the number of literals in the clause. This assumes that every literal can be resolved against some unit clause. Otherwise, the e-r-l method associated with a clause sends an e-r-l message to each of the literals in that clause and returns the sum of the answer. For depth \( n \), the e-r-l method associated with a literal sends an e-r-l message to each linked clause for depth \( n-1 \) and returns the minimum of the responses. With the e-r-l methods in place the connection graph can estimate length of a refutation involving any or all of its clauses to any depth.

Kowalski [1976] stated that the efficiency of search guided by lower bounds estimate improves significantly with the improved estimates of the lower bound. For any particular application it remains to be determined how deep to look in an exponential space before the benefit of the improved estimate in lower bounds is no longer worth the expense of computing that estimate.
CHAPTER 3

This search heuristic is an example of the way in which communicating objects can be used to drive the search. This type of method is simple to implement, local and easy to extend or modify by simply adding or redefining a single method for a single class of objects. One extension to this approach is that rather than estimating the refutation length, the method estimates the complexity of the refutation involving a clause. The complexity measure takes into account the number of terms and depth of nesting of the terms in the literals being unified at each stage of the refutation. The connection graph representation makes the information about the complexity of a refutation available. The object oriented implementation provides a natural way of collecting the information.

3.3.2.3 Hybrid Search

A simple version of the hybrid search described in section 3.1 above was also implemented. The method \textit{h-search} (hybrid-search) attached to the connection graph object drives the search. It maintains a prioritized stack of clauses used in top-down search and a list of data clauses used in bottom-up search. The priority for using a clause in top-down search is established by the degree to which it has already been solved. In the implementation this is measured simply by the ratio of literals which have been resolved to the number not yet resolved. A clause will not be used for top-down search until that ratio exceeds one (half the literals in the clause body have been resolved).

The objective of the \textit{h-search} method is to determine the order in which links are selected for resolution. The data-driven phase of the hybrid search is the default strategy used when there are no goals being actively pursued. In the
implementation, if there are no clauses on the top-down clause stack then an arbitrary data clause is selected. The data clause is sent a \textit{b-u-search} (bottom-up-search) message. When a clause receives a \textit{b-u-search} message it selects an arbitrary link for resolution, sends the \textit{resolve} message to that link and returns the resulting resolvent.

If there are goals on the top-down priority list, the clause with the highest priority is selected and sent a \textit{t-d-search} message. The clause chooses the leftmost literal in its body which in turn selects a link in the same manner as in the top-down search described above. This link selection approach implements the \textit{matching} phase of Havens' search.

If the resolvent returned by either approach has no more literals in its body, it becomes a data clause and is added to the list of bottom-up clauses, thus implementing the \textit{completion} phase of the search.

Finally, a resolvent only inherits links which are compatible with the link chosen for resolution. This restricts the possible bindings of the rest of the variables in the resolvent to be consistent with all of the variables so far instantiated. As discussed in section 3.1.3, this is a reasonable way to implement the \textit{expectation} phase of the hybrid search.

However, without a thorough and detailed implementation of the connection graph method such as the one used by Blausius [1984], it would be impossible to analyse the merits of alternate approaches to switching to top-down search. It is clear however, that the connection graph method allows the natural incorporation of this hybrid search strategy.
3.3.3 Implementation Details

This implementation closely follows the suggestions in [Bruynooghe 1976]. In this section, some of the interesting aspects of the implementation of the connection graph method which were encountered during the development of our system are discussed. The code for the implementation is attached in the Appendix.

3.3.3.1 Structure Sharing

Structure sharing [Boyer 1972] provides a way of representing clauses in a constant amount of space independent of the number of literals in the clause and the depth of function nesting of the terms within the literals. The central idea of structure sharing is that rather than computing a resolvent explicitly by substituting the unifier of the two clauses over all of the literals in the resolvent, the resolvent can be represented implicitly by a tuple which indicates the left parent clause, the literal resolved upon in the left parent clause, the right parent clause, the literal resolved upon in the right parent clause, and the unifying substitution of the two literals. Boyer [1972] estimated that structure sharing is 50 to 100 times more efficient in usage of space than a standard list notation for clauses. In addition, the creation of new resolvents is faster since no substitution instances of the literals in the resolvents have to be computed during resolution. The only drawback to a structure sharing implementation is in ascertaining the actual bindings of the arguments of some literal in a resolvent. In a structure sharing approach this requires a recursive lookup procedure whereas the values are immediately available in an implementation which does not use structure
sharing. The lookup operation is however very efficient in the structure sharing scheme [Boyer 1972].

Bruynooghe [1976] described how structure sharing can be incorporated into a connection graph implementation. However, in the implementation used in this thesis several problems with the use of structure sharing in connection graphs were discovered. Foremost is the fact that the resolvents in a connection graphs could not be represented in constant size. This is because each literal in the resolvent must contain links to each of the potentially complimentary literals (at least one) in the connection graph. Hence the space requirement for representing a clause is dependent on the number of literals it contains. It is not sufficient in a connection graph for a resolvent merely to point to the parent clauses because not all of the links to each of the parent clauses are necessarily inherited by the resolvent. It should be noted that in spite of the fact that the space requirement for representing a clause is no longer constant, it is still much better than that required to represent the clause explicitly since the structure within the literals (nested functions) is still shared.

The second problem with using structure sharing in connection graphs also stems from the presence of links in the graph. In a connection graph system, when a particular resolution is chosen and the resolvent is created, the resolvent must inherit instances of all or some of the links attached to each of the parent clauses. This operation connects the resolvent to the rest of the graph by adding links between the resolvent and every other clause in the graph against which it can in turn be resolved. There is no analogous operation in standard resolution systems since they do not explicitly represent all possible resolutions between clauses. During the inheritance operation each link which is to be
inherited from a parent clause to a new resolvent must first be tested for compatibility with the unifier which was used in creating the resolvent. The test for compatibility of a link and a unifier requires many lookups in a structure sharing implementation. In fact, since all of the variables occurring in a clause will also occur in many of the links to that clause, it is quite possible that all the bindings in the clause will be looked up one or more times in determining which links are to be inherited by the resolvent. Furthermore, this can occur each time a particular clause is used as a resolvent. Thus, one of the main benefits of structure sharing is lost in the connection graph system.

This second problem is compounded if any amount of constraint satisfaction is used. Compatibility tests are at the heart of constraint satisfaction techniques in connection graphs and will result in repetitive binding lookups.

A final problem with structure sharing is that clauses which are no longer active in the connection graph cannot be deleted because they contain variable binding information which may be needed by the active clauses. This is because newly formed resolvents use their ancestors in order to create a unique variable binding views [Bruynooghe 1976]. This is clearly an inefficient use of space.

The implementation in this thesis uses a structure sharing approach to implementing connection graphs. The object oriented style of programming is quite useful in this approach. Clauses sharing instances of the same literal simply point to the same literal and keep an internal state variable which described the binding environment particular to that clause. This as opposed to clauses actually pointing to separate instances of the literal which differ only in the bindings of their terms.
CHAPTER 3

The problems mentioned above cast some doubt on the usefulness of standard structure sharing in a connection graph implementation. It is outside the focus of this thesis however to determine the impact of using structure sharing in a connection graph system to any greater extent. Note that since the implementation is object oriented, switching from a structure sharing approach entails only local changes to a few methods in the clause and literal objects.

3.3.3.2 Links and Clauses

It was noted by Bruynooghe [1976] that the links between two literals are essentially unresolved clauses. Links and clauses have very similar internal state representations. Both need to know the literals and clauses which are joined by the link and which create the resolvent. Both links and clauses also need to know the unifier of the two parent literals. The object oriented implementation was able to take advantage of this insight, representing clauses and links as a single type of object with a state variable indicating which mode a particular instance was being used in. This approach simplified the creation of resolvents during the operation of proof procedure. When a link accepts a resolve message, it has only to inherit the appropriate literals and links from the parent clause and switch an internal variable to become a clause.

3.3.3.3 Object Oriented Language Implementation

The object oriented language used for the development the connection graph system is called Objects. It was jointly developed by William Havens and myself in order to look at some of the issues in object oriented programming.
CHAPTER 3

Objects is written in MacScheme† [SM 1986] and incorporates many of the ideas suggested in [Abelson 1985]. This section provides an overview of the Objects language including some of the implementation decisions as well as an outline of object definitions. This is to enable the reader to consult the code for the connection graph implementation.

An object definition in Objects has the following general form:

\[
\text{(Objectdef } <\text{object-name}> \\
\quad \text{(supers } <\text{superclass}>* \\
\quad \text{(variables } (<\text{var-name}> <\text{initial-value}>* \\
\quad \text{(methods } (<\text{method-name}> <\text{method-procedure}>* \\
\quad \text{(localfns } <\text{localfn}>*))
\]

The Objectdef macro for \(<\text{object-name}> = \text{foo}\) will return an object named foo. Henceforth messages can be sent to foo and/or new instances of foo can be created.

Supers is a (possibly empty) list of objects which serve as superclasses of the object being defined. In Objects, every object is assumed to be a specialization of all of its superclasses.

Variables is a list of the internal state variables and their initial values. These variables may be atomic values or may be used point to components of the main object.

Each method-name in the methods list identifies a message type which the object can respond to. The method-procedure is a MacScheme function which

†MacScheme is a trademark of Semantic Microsystems.
CHAPTER 3

computes the response to the message. This function may also send further messages or use the object's local functions.

Localfns is a list of functions which are only available to the class of objects being defined. These are generally used to perform actions on the internal state of the object and may be used by one or more of the methods of the object.

A message is sent as follows:

\[ \text{(-> object message arg}_1\ldots\text{arg}_n) \quad \text{where } n \geq 0. \]

All arguments to -> are evaluated.

Every object in Objects is provided with a core of methods which are primarily of use for debugging and for internal operations such as inheritance. One such method however is the :new method which implements prototyping. In Objects, any object receiving a :new message creates and returns a copy of itself. In this way any object can serve as a prototype. This is of great use in the implementation of connection graphs where new instances of literals and links are being created after each resolution.

The implementation of multiple inheritance in Objects leaves much to be desired. Those objects specified in the supers section of a definition contribute all of their instance variables and methods to the object being defined. However, the language does not provide any primitives for resolving method or variable name clashes during inheritance. Inheritance is also expensive in terms of space. Since method inheritance is done at compile time, there can be multiple copies of a method throughout the class/superclass inheritance hierarchy. Unfortunately, the use of static scoping in MacScheme makes it impossible to do run-time method lookups. However, the use of inheritance was not of paramount
importance for the implementation of connection graphs and therefore a minimal implementation was sufficient for our purposes.

To this point the *Objects* language provides a compiler which transforms object definitions into functions which are invoked to create closures representing objects. It also provides a message sending function and a rudimentary debugger for examining the internal state of objects during computation.
Chapter 4: Discussion

4.1 Flexible Search in Connection Graphs

The focus of the review in section 2.2 was on the procedural aspects of the logic program or Horn clause subset of FOL. It was noted that many of the procedural enhancements of logic programming implementations were designed to make programs run more efficiently. However, the goals of an efficient programming language and the goals of KR formalism are sometimes at odds with one another.

One of Mackworth's [1987] criteria for a procedurally adequate visual knowledge representation system is that it be able to use knowledge in a flexible manner. "An ideal image-based system would regard all of its potential information sources as inputs or outputs depending on the availability of information. The system would allow control flow from image to scene (image analysis) or from scene to image (image synthesis)". Since logic representations are generally ambivalent with regards to input/output classification of information sources, they are inherently flexible with regards to synthesis and analysis. However, this flexibility is lost in logic programming implementations in order to achieve greater one-way efficiency of logic programs. By using appropriate link selection algorithms, it is possible to implement data driven
search in connection graphs whether the data is in the form of complex (scene domain) objects or simple visual cues (image domain). Thus a connection graph implementation retains the flexibility of the logic representation.

Another aspect of flexibility in KR is the ability to change focus of search dynamically. The hybrid search described above is an example of how this can be done in connection graphs. In Prolog, the global search strategy is fixed and cannot be changed for entire programs let alone while the search is in progress.

On the other hand, the connection graph implementation of automatic backtrack search is less efficient in terms of space usage than the standard implementations. Also, the extra-logical control primitives available in many logic programming languages rely on the data structures used specifically in their implementations and thus cannot be naturally incorporated in connection graphs.

### 4.2 Approximation Techniques in Connection Graphs

Some tasks in knowledge representation are too difficult to be solved by polynomial algorithms. In these cases it may be possible enhance the procedural efficiency of the knowledge representation system by using efficient approximation algorithms. Approximation algorithms will not generally find the solutions to a problem but will rather eliminate non-solutions thus reducing the size of the solution space which must subsequently be searched.

In connection graphs, approximation algorithms will reduce the size of the graph by removing redundant or inconsistent links and/or clauses. Bibel's connection graph reduction operations and the link filtering algorithm discussed
above are such approximation algorithms. Aside from reducing the size of the connection graph, approximation algorithms must also be efficient. The combination of these two aspects makes for a proper approximation algorithm. Bibel's graph reduction algorithms have been shown to be proper. However, Kowalski's link filtering does not always run in polynomial time and is thus not proper. Some possible refinements to his algorithm which make it proper (albeit weaker), were discussed above. In addition, it was shown above that Kowalski's algorithm does not provide the useful intermediate results which are provided by Freuder's k-consistency algorithm when run under resource limitation. In particular, the link filter does not guarantee the computation of 2- or 3-consistency for arbitrary constraint satisfaction problems. These intermediate results have been found to be useful by several AI researchers.

It is important to determine the complexity of approximation algorithms before they are used. The Markgraph Karl Refutation Procedure [Blasius 1984] is a connection graph based automatic theorem proving system. In the system several types of graph reduction operations are used in order to prevent the system from performing any unnecessary resolutions. The system performs exceedingly well in that it generally requires only a fraction of the resolutions required by other automatic theorem proving systems to find proofs. However, the reduction techniques it uses are not proper, leading to vast resource usage in order to make its preprocessing graph reductions. In trying to avoid the standard theorem proving approach of very fast search through an exponential space, this approach virtually solves problems with non-inferential preprocessing operations and follows with very short and often perfect searches.
Like the incorporation of flexible search techniques, the use of approximation techniques is a standard way to enhance the procedural efficiency of a knowledge representation system. Standard logic programming implementations have been designed with the focus on efficient execution of programs rather than knowledge representation. Thus, none have incorporated constraint satisfaction techniques or other approximation algorithms which have their primary benefits in knowledge representation systems.

The connection graph data structure lends itself naturally to implementation of consistency techniques. The connection graph method proceeds by removing links and simplifying the graph. Link filtering and graph reduction operations also simplify the graph. The approximation algorithms therefore cooperate with the overall goal of the connection graph method. The data structures used in standard logic programming implementations are not amenable to the incorporation of this type of operation.

4.2.1 Extending Constraint Satisfaction in Connection Graphs

In previous discussion of consistency techniques in connection graphs it was assumed that the connection graphs represented only standard constraint satisfaction problems. Connection graphs however, can be used to represent a much larger set of problems. In particular, the variable domains need not be finite. Nor is it the case that the constraints need be finite. In fact, constraints may be defined in terms of other predicates or in terms of themselves. These types of problems can not be solved by Freuder's k-consistency algorithm simply
because his algorithm may consider all the elements in all constraint relations during propagation of constraints.

In the connection graph representation of such problems, the unifiers associated with the links to literals in the goal clause need not be ground. The variables in the unifiers may be bound to ground or non-ground terms depending on the definition of the constraints involved. In essence, a variable bound to a non-ground term represents the fact that that variable may have an undetermined number of potential instantiations. By using unification, the compatibility of non-ground instantiations of variables in separate unifiers can be determined. Thus, it is possible to use the link filter for consistency preprocessing for more than just the standard constraint satisfaction problems. However, the compatibility test for unifiers involving non-ground terms is more complex than if the unifiers are ground.

Mackworth's [1985a] complexity analyses for consistency algorithms assume finite variable domains. Davis [1985] considered the use of constraint satisfaction techniques for problems with infinite variable domains and he contends that the analysis of such problems "depends critically on the nature of the constraints used". In connection graphs, variable domains may be infinite and there is little restriction on the nature of constraint relations used.

The benefit of using constraint satisfaction on arbitrary connection graph problems remain the same: the removal of inconsistent links reduces the size of the graph and therefore the size of the search space. The cost of using constraint satisfaction algorithms for this type of problem remains to be analysed.
4.3 Object Oriented Programming

From the previous description in section 3.3, it is clear that the connection graph method is well suited to an object oriented implementation. The components of the graph are naturally described as classes of objects with small amounts of local storage. Operations on connection graphs are naturally described in the message passing paradigm of object oriented programming.

The flexibility of search in connection graphs stems from the nondeterminism in the link selection phase of the proof procedure. Thus it is the connection graph representation which makes it possible to use special search strategies in a logic program representation. However, it is the object oriented implementation of connection graphs which makes it possible to implement the various strategies with minimal effort. Objects have the ability to examine their own state and initiate actions based on their findings. Thus, active search can be directed by objects at the appropriate conceptual level rather than at the global level.

An object can also gather information from related objects via message passing. Thus, a search procedure can base control decisions on sets of associated objects rather than on single objects. For example, a connection graph search procedure such as the e-r-l method described above, can explore the connection graph structure around a given link in order to determine the priority of that link for resolution.

The modularity of object oriented programs makes the correction, modification and extension of code straightforward. In an experimental system like the one developed for connection graphs, this is extremely beneficial. In
particular, the extension of the raw connection graph system to include assorted search strategies was simple. Only a few methods and state variables had to be added to a few classes in order to bring each new search strategy into the system. None of the existing code had to be modified. Changes and corrections were also limited to single methods or small groups of mutually dependent methods.

It was noted above that the notion of inheritance of methods and instance variables in class-superclass hierarchies is a controversial one in object oriented programming languages. However, there is no hierarchy of objects in connection graphs and thus the controversy surrounding inheritance has no bearing on a connection graph implementation. Prototypical inheritance is better understood and is implemented in the *Objects* language. It proved to be useful in the connection graph implementation for the inheritance of links after resolution.

### 4.4 Conclusions

The connection graph representation of logic programs facilitate the introduction of flexible search and constraint satisfaction techniques into a logic programming implementation. Both of these techniques extend the procedural adequacy of the logic program formalism over standard logic programming implementations without sacrificing any descriptive capacity. The use of several types of flexible search techniques in connection graphs were demonstrated in this thesis, focusing on the incorporation of the principles of Havens' hybrid search technique in connection graphs.

The second focus of this thesis has been the use of constraint satisfaction in connection graphs. Constraint satisfaction algorithms are generally used to do a
limited amount of preprocessing in order to reduce the subsequent search. It has been shown that constraint satisfaction techniques can be used in connection graphs although the algorithm originally proposed by Kowalski is too complex to be of general use. It was also shown that Kowalski's algorithm is not guaranteed to provide any of the intermediate degrees of k-consistency which are provided by Freuder's k-consistency algorithm and which are often used in other AI tasks.

Object oriented programming proved useful for the connection graph implementation as well as the incorporation and extension of various search techniques in connection graphs.

4.5 Future Work

There are several directions in which the work in this thesis can be extended. The connection graph implementation can be extended to include some constraint satisfaction algorithms. It would be interesting to collect empirical evidence for the benefits of using constraints in some standard logic programming problems as well as KR problems represented as logic programs. The use of constraint satisfaction over partially instantiated variables is also worthy of consideration both empirically and from a theoretical standpoint.

It is simple to restrict the subset of FOL represented by a connection graph by restricting the types of clauses in the inputs. Since connection graphs are very flexible with regards to search, they might provide a good tool to test KR forms other than Horn clauses.

Finally, many of the connection graph search techniques and reduction algorithms discussed above seem to lend themselves naturally to parallel
implementations, especially in an object oriented system. Thus, the benefits of parallelism in a connection graph implementation should be considered.
Appendix: Connection Graph Implementation

This appendix contains the code used to implement connection graphs and various associated search strategies. The code is written in the *Objects* language which is an object oriented extension to MacScheme. An object definition schema for the Objects language is given in section 3.3.3 above.

Connection Graph Object Definition

; Definition of the Connection Graph (CG) object.
; This version of the file the CG definition includes methods and
; local storage for search.

(Objectdef cg
 (supers object)
 (variables (clauses ()))
 (search-type ()))
 (t-d-clauses ())
 (b-u-clauses ())
 (search-stack ()))
 (methods (make-cg (make-cg (1st args)))
 (initialize (initialize (1st args)))
 (default-search-init (d-s-init))
 (default-search (d-search))
 (hybrid-search-init (h-s-init))
 (hybrid-search (h-search))
 (resolve-link (resolve-link (1st args)))
 (add-clause (add-clause (1st args)))
 (remove-clause (remove-clause (1st args)))
 (done (display "Refutation Complete") (newline))
 (get-rep (get-rep)))

(localfns

; make a new connection graph
 (define make-cg
 (lambda (i-c)
 (-> (*self* ':new) 'initialize i-c)))

; initialize sets up the internal state of a new CG from a
; set of input clauses.
 (define initialize
APPENDIX: CONNECTION GRAPH IMPLEMENTATION

(lambda (input-clauses)
  (set! clauses ())
  (mapc (lambda (input-clause)
    (let ((new-clause (make-clause input-clause *self*))
      (connect-clause new-clause clauses)
      (connect-to-self new-clause)
      (add-clause new-clause))
    input-clauses)
  *self*))
)

; connect a new clause into the connection graph
(define connect-clause
  (lambda (new-clause clauses)
    (mapc (lambda (clause)
      (mapc (lambda (literal)
        (connect-literal literal clause))
      clauses)
    clauses)
  )
)

; connect a literal in one clause to the literals in another
(define connect-literal
  (lambda (lit1 clausel)
    (mapc (lambda (lit2)
      (-> clause 'make-link lit1 lit2 *self*)
    )
    clausel )
  )
)

; make any internal links in a clause
(define connect-to-self
  (lambda (new-clause)
    (cts (-> new-clause 'get-literals)))
)

(define cts
  (lambda (literals)
    (cond ((<? (length literals) 2))
      (else (cts1 (car literals) (cdr literals))
      (cts (cdr literals))))
)

(define cts1
  (lambda (lit1 literals)
    (mapc (lambda (lit2)
      (-> clause 'make-link lit1 lit2 *self*)
    )
    literals)))

; use the default search. For now this is top-down, left to right.
(define d-s-init
  (lambda ()
    (set! search-type 'default)
    (set! search-stack (-> stack ':new))
    (-> search-stack 'push (1st clauses)))
)

(define d-search
  (lambda ()
    )
)
(let ((goal-clause (-> search-stack 'top)))
  (cond ((eq? goal-clause 'empty)
      (cerror "problem: Search stack empty: no solution"))
      (else (let ((resolvent (-> goal-clause 'd-search))
                   (if resolvent
                      (-> search-stack 'push resolvent)))))))

(define h-s-init
  (lambda ()
    (set! search-type 'hybrid)
    (set! t-d-clauses (-> sorted-a-list ':new))
    (set! b-u-clauses (-> queue ':new))
    (mapc (lambda (clause)
            (and (-> clause 'b-u-clause?)
                 (-> b-u-clauses 'add clause))
            clauses))))

; hybrid search
(define h-search
  (lambda ()
    (let ((choice (-> t-d-clauses 'top))
           (resolvent ())
           (priority ()))
      (cond ((or (eq? choice 'empty) (<? (rest choice) 1))
         ; bottom up search if no t-d search is appropriate
         (set! choice (-> b-u-clauses 'pop))
         (set! resolvent (-> choice 'b-u-search))
         ; put the clause back on the list of b-u-clauses
         (and (-> choice 'b-u-clause?)
              (-> b-u-clauses 'add choice)))
      (else
        ; t-d search if there is a clause with priority>=1
        (set! choice (-> t-d-clauses 'top))
        (set! resolvent (-> (1st choice) 't-d-search)))
    ; if there is a resolvent, put it with the appropriate group
    (and
     resolvent
     (cond ((-> resolvent 'b-u-clause?)
            (-> b-u-clauses 'add resolvent))
            (else (set! priority (-> resolvent 't-d-priority))
                   (if priority
                      (-> t-d-clauses 'add resolvent priority)))))))

; allows user to guide search by hand
(define resolve-link
  (lambda (link)
    (if (null? clauses)
        "no solution"
        (-> link 'resolve))))

(define add-clause
  (lambda (new-clause) (set! clauses (cons new-clause clauses))))

; remove a clause from the connection graph
(define remove-clause
  (lambda (new-clause) (set! clauses (cons new-clause clauses))))
APPENDIX: CONNECTION GRAPH IMPLEMENTATION

(lambda (clause)
  (set! clauses (remove clause clauses))
  (case search-type
    ((hybrid) (-> t-d-clauses 'delete clause)
      (-> b-u-clauses 'delete clause))
    ((default) (-> search-stack 'remove clause)
      (else #!true))))

; get an output representation of the clauses in the graph
(define get-rep
  (lambda ()
    (cons *self*
      (map (lambda (clause)
          (-> clause 'get-rep)
            clauses)))))

Clause Object Definition

; Clause object for connection graphs.
; The Clause object is also used to represent connection graph links.
(Objectdef clause
  (supers object)
  (variables (literals ()
    (age ()
      (right-parent ()
        (right-parent-lit ()
          (left-parent ()
            (left-parent-lit ()
              (bndenv ()
                (mi-1p 1)
                (max-index 1)
                (link #!false)
                (active #!true)
                (owner ())
              )
            ))
          ))
        ))
      )
    )
  (methods (make-clause (make-clause (1st args) (2nd args)))
    (init-clause (init-clause (1st args) (2nd args)))
    (make-link (make-link (1st args) (2nd args) (3rd args)))
    (d-search (d-search))
    (b-u-search (d-search))
    (t-d-search (t-d-search))
    (b-u-clause? (and active (b-u-clause?)))
    (t-d-priority (and active (t-d-priority)))
    (estimate-refutation-length (e-r-l (1st args) (rest args)))
    (resolve (and link (resolve)))
    (inherit-clause (and (not link) (apply inherit-clause args)))
    (inherit-link (and link (apply inherit-link args)))
  )
)
(:inherit-link (and link (apply :inherit-link args)))
(get-max-index max-index)
(get-age age)
(get-literals literals) ; and hope nobody messes with it
(get-left-parent left-parent)
(get-left-parent-lit left-parent-lit)
(get-right-parent right-parent)
(get-right-parent-lit right-parent-lit)
(lookup (lookup (1st args) (2nd args)))
(bind (forward-to bndenv))
(delete (and active (delete)))
(get-rep (get-rep))

(localfns

; make a new clause
(define make-clause
  (lambda (i-c cg)
    (-> (-> clause ':new) 'init-clause i-c cg)))

; create a cg clause from an input clause
(define init-clause
  (lambda (input-clause cg)
    (set! owner cg)
    (set! literals '())
    (set! age 0)
    (set! right-parent '())
    (set! right-parent-lit '())
    (set! left-parent '())
    (set! left-parent-lit '())
    (set! mi-lp 1)
    (set! max-index 1)
    (set! link #!false)
    (set! literals
      (map (lambda (input-literal)
            (-> literal 'make-literal input-literal 1 *self*)
            input-clause))
      *self*))

; try to make a link between two literals
(define make-link
  (lambda (lpl rpl cg)
    (cond ((and (eq? (-> lpl 'get-pred-name) (-> rpl 'get-pred-name))
                  (not (eq? (-> lpl 'get-sign) (-> rpl 'get-sign))))
      (set! owner cg)
      (set! age 0)
      (set! link #!true)
      (set! right-parent-lit rpl)
      (set! right-parent (-> rpl 'get-parent-clause))
      (set! left-parent-lit lpl)
      (set! left-parent (-> lpl 'get-parent-clause))
      (set! mi-lp (-> left-parent 'get-max-index))
      (set! max-index (+ mi-lp (-> right-parent 'get-max-index))))

105
APPENDIX: CONNECTION GRAPH IMPLEMENTATION

(cond (bndenv (-> bndenv 'reset))
  (else (set! bndenv (-> substitution ':new))))
(cond ((-> unifier
  'unify (list (cons (-> lpl 'get-term)
                  (-> lpl 'get-index)))
  (list (cons (-> rpl 'get-term)
            (+ mi-lp
             (-> rpl 'get-index))))
  *self*)
  (let ((new-link (-> *self* ':new))
        (-> lpl 'add-link new-link)
        (-> rpl 'add-link new-link)
        (set! bndenv (-> substitution ':new))
        new-link))
  (else #!false)))
(else #!false)))

; Make a default link selection for the clause and resolve it.
(define d-search
 (lambda ()
  (-> (1st literals) 'd-search)))

; Make link selection for top down search from this clause
(define t-d-search
 (lambda ()
  (t-d-search-1 literals)))
(define t-d-search-1
 (lambda (lits)
  (cond ((null? lits) #!false)
        ((eq? (-> (1st lits) 'get-sign) '-)
         (-> (1st lits) 'd-search))
        (else (t-d-search-1 (rest lits))))))

; Determine if the clause should be used for bottom-up search.
(define b-u-clause?
 (lambda ()
  (and (eq? (length literals) 1)
       (eq? (-> (1st literals) 'get-sign) '+))))

; Establish the priority for the use of a clause in top-down search.
; For now this is simply the ratio of literals currently in a
; clause to the literals which have been resolved away.
(define t-d-priority
 (lambda ()
  (and (>? age 0) (/ age (length literals))))))

; Estimate the minimum length of a refutation involving this clause
(define e-r-1
 (lambda (depth lits)
  (cond ((<=? depth 1) (- (length literals) (length lits)))
        (else (do ((sum 0)
                   (ls literals (cdr ls))
                   ((null? ls) sum)
                   (cond ((member (1st ls) lits)
                           (e-r-1 (-> lpl 'add-link new-link)
                                 (-> rpl 'add-link new-link)
                                 (set! bndenv (-> substitution ':new))
                                 new-link))
                          (else #!false))))
        (else #!false)))

106
APPENDIX: CONNECTION GRAPH IMPLEMENTATION

```
; change an unresolved link to a resolvent
(define resolve
  (lambda ()
    (set! literals
      ; inherit literals from the left parent
      (-> left-parent 'inherit-clause *self* 0 left-parent-lit))
    (cond ((and literals (eq? (car literals) '*pure*))
      (else
       (set! literals
         ; inherit literals from the right parent
         (append
          (-> right-parent
            'inherit-clause *self* ml-lp right-parent-lit)
          literals))))
    ; compute the age of the resolvent
    (cond ((eq? (-> left-parent-lit 'get-sign) '-)
      (set! age (+ 1 (-> left-parent 'get-age))))
      (else (set! age (+ 1 (-> right-parent 'get-age))))))
    ; remove the link from the graph
    (-> left-parent-lit 'remove-link *self*)
    (-> right-parent-lit 'remove-link *self*)
    (set! link #!false)
    ; add the resolvent to the graph return the resolvent itself
    (cond ((and literals (eq? (car literals) '*pure*))
      (set! literals (rest literals))
      (set! *self* 'delete)
      #!false)
      (literals
       (-> owner 'add-clause *self*) *self*)
      (else (-> owner 'done) #!false))))

; new versions of the literals in this clause (except the one which
; was resolved upon) are to be passed on to a newly resolved clause
; stop if any literal is pure (has no links).
(define inherit-clause
  (lambda (resolvent index-update resolved-lit)
    (let ((new-lits ()) (new-lit ()) (done ()))
      (do ((lits literals (rest lits)))
        (done new-lits)
        (cond ((null? (rest lits)) (set! done #!true)))
        (cond ((eq? (car lits) resolved-lit))
          (else
           (set! new-lit
             (-> (car lits) 'inherit resolvent index-update))
             (cond (new-lit (set! new-lits (cons new-lit new-lits)))
               (else (set! new-lits (cons '*pure* new-lits))
                 (set! done #!true))))))

; a new version of this link is to be added to a new resolvent
(define inherit-link
  (lambda (cinh linh cres new-lpl)
    (-> (-> *self* ':new) ':inherit-link cinh linh cres new-lpl)))
```
APPENDIX: CONNECTION GRAPH IMPLEMENTATION

; see section 7 of Bruynooghe for description.
; cinh is clause containing literal being inherited
; cres is the resolvent clause
; new-lpl is the left-parent-literal of the inherited link and
; is necessarily in the resolvent.
(define :inherit-link
  (lambda (cinh linh cres new-lpl)
    (let ((low-update 0)
           (high-update 0)
           (new-rpl ()))
      ;; 4 cases
      (cond ((and (eq? linh left-parent-lit)
                  (eq? cinh (-> cres 'get-left-parent)))
             (set! low-update 0)
             (set! high-update (-> (-> cres 'get-right-parent)
                                    'get-max-index))
             (set! new-rpl right-parent-lit))
            ((and (eq? linh left-parent-lit)
                  (eq? cinh (-> cres 'get-right-parent)))
             (set! low-update (-> (-> cres 'get-left-parent)
                                  'get-max-index))
             (set! high-update (-> (-> cres 'get-left-parent)
                                   'get-max-index))
             (set! new-rpl right-parent-lit))
            ((and (eq? linh left-parent-lit)
                  (eq? cinh (-> cres 'get-left-parent)))
             (set! low-update (-> cres 'get-max-index))
             (set! high-update (- mi-lp)
                               (-> (-> cres 'get-left-parent)
                                   'get-max-index))
             (set! new-rpl left-parent-lit))
            ((and (eq? linh right-parent-lit)
                  (eq? cinh (-> cres 'get-right-parent)))
             (set! low-update (-> cres 'get-max-index))
             (set! high-update (- mi-lp (-> (-> cres 'get-left-parent)
                                           'get-max-index))
                               (-> cres 'get-left-parent)
                               'get-max-index))
             (set! new-rpl left-parent-lit))
            (else (cerror "None of the four cases apply, schiersten")))
      (let ((tms (-> bndenv 'get-updated-bindings
                          mi-lp low-update high-update)))
        (set! right-parent-lit new-rpl)
        (set! right-parent (-> new-rpl 'get-parent-clause))
        (set! left-parent-lit new-lpl)
        (set! left-parent cres)
        (set! mi-lp (-> left-parent 'get-max-index))
        (set! max-index (+ mi-lp (-> right-parent 'get-max-index)))
        (set! bndenv (-> substitution ':new))
        (cond ((-> unifier 'unify (1st tms) (rest tms) *self*)
               (-> new-lpl 'add-link *self*)
               (-> new-rpl 'add-link *self*)
               *self*)
               (else #!false))))))

; lookup a variable binding in the view of this clause
(define lookup
  (lambda (var index)
    (cond ((eq? max-index 1) #!false)
          ((-> bndenv 'lookup var index))
          ...)))
APPENDIX: CONNECTION GRAPH IMPLEMENTATION

```
else
  (if (<=? index mi-lp)
    (-> left-parent 'lookup var index)
    (let ((binding (-> right-parent 'lookup var
                        (- index mi-lp)))
          (cond (binding (cons (car binding)
                            (+ mi-lp (cdr binding))))
                (else '#false))))))))

; deletion depends on whether the clause is representing a
; resolved clause or an unresolved clause (a link)
(define delete
  (lambda ()
    (set! active '#false)
    ; for links, just inform both ends of the link
    (cond (link (-> right-parent-lit 'remove-link *self*)
              (-> left-parent-lit 'remove-link *self*)
              ; for resolved clauses, delete all the literals. This is
              ; done when one of the literals becomes pure.
              (else (-> owner 'remove-clause *self*)
                     (mapc (lambda (literal)
                             (-> literal 'delete))
                            literals)
                    (set! literals '()))))))

; get an output representation for the clause
(define get-rep
  (lambda ()
    (cond ((eq? max-index 1)
           (cons *self*
                 (map (lambda (literal)
                        (-> literal 'get-rep))
                      literals)))(else (list *self*
                              'left-parent-clause:
                              left-parent
                              'right-parent-clause:
                              right-parent
                              'binding-environment:
                              (-> bndenv 'get-rep)
                              'literals:
                              (map (lambda (literal)
                                     (-> literal 'get-rep))
                                   literals)))))))
```
APPENDIX: CONNECTION GRAPH IMPLEMENTATION

Literal Object Definition

; A literal in the connection graph has nearly constant size.
; It points to a term representing its internal structure and
; a list of links which represent its connectivity in the CG.

; This version includes a method for default search. The default is
; simply to select the leftmost link to a literal for resolution.

(Objectdef literal
  (supers object)
  (variables (sign ())
    (term ())
    (index 0)
    (links ())
    (parent-clause ())
    (active #!true))
  (methods (make-literal
                (make-literal (1st args) (2nd args) (3rd args))))
    (initialize
      (initialize (1st args) (2nd args) (3rd args)))
    (d-search (d-search))
    (estimate-refutation-length (e-r-l (1st args)))
    (get-sign sign)
    (get-term term)
    (get-index index)
    (get-pred-name (-> term 'get-constructor))
    (get-parent-clause parent-clause)
    (get-number-links (length links))
    (add-link (set! links (cons (1st args) links)))
    (remove-link (and active (remove-link (1st args))))
    (inherit (inherit (1st args) (2nd args)))
    (:inherit (:inherit (1st args) (2nd args) (3rd args)))
    (delete (and active (delete)))
    (get-rep (get-rep))
    (else (forward-to term)))

(localfns

 ; make a new literal from a term object
 (define make-literal
  (lambda (i-l ind pc)
    (-> (-> literal ':new) 'initialize i-l ind pc)))

 ; initialize the literal
 (define initialize
  (lambda (lit ind pc)
    (set! sign (1st lit))

110
\begin{verbatim}
(set! term (2nd lit))
(set! index ind)
(set! parent-clause pc)
(*self*))

; select a default link for resolution
(define d-search
  (lambda ()
    (-> (1st links) 'resolve)))

; estimate the min number of resolutions to refute this literal
(define e-r-l
  (lambda (depth)
    (let ((ref-length 1000)
          (linked-cl ())
          (linked-lit ()))
      (mapc (lambda (link)
              (set! linked-lit (-> link 'get-left-parent-lit))
              (cond
               ((eq? *self* linked-lit)
                (set! linked-lit (-> link 'get-right-parent-lit))
                (set! linked-cl (-> link 'get-right-parent))))
              (else
                (set! linked-cl (-> link 'get-left-parent))))
            (let ((r-1 (-> linked-cl 'estimate-refutation-length
                            (- depth 1)
                            linked-lit)))
            (if (>? ref-length r-1) (set! ref-length r-1)))
            (links)
            (+ 1 ref-length))))

; remove a link to this literal
(define remove-link
  (lambda (link)
    (set! links (remove link links))
    ; if the deletion makes the literal pure,
    ; delete the clause containing the literal.
    (cond (links #!true)
          (else (set! active #!false)
                 (-> parent-clause 'delete))))))

(define inherit
  (lambda (n-p i-u)
    (-> (-> *self* ':new) ':inherit n-p i-u *self*))

(define :inherit
  (lambda (new-parent index-update linh)
    (let ((cinh parent-clause) (new-links ()))
      (set! parent-clause new-parent)
      (set! index (+ index index-update))
      (mapc (lambda (link)
              (let ((new-link (-> link 'inherit-link
cinh
linh
\end{verbatim}

111
APPENDIX: CONNECTION GRAPH IMPLEMENTATION

(parent-clause
 *self*))
   (cond (new-link
           *self*)
         (set! new-links (cons new-link new-links)))))
   links)
   (set! links new-links)
 ; return nil if the new literal is pure
   (cond (links *self*)
         (else ()))))

; delete the literal
(define delete
  (lambda ()
    (set! active #!false)
    (mapc (lambda (link) (-> link 'delete)) links)))

; get a representation of the literal (suitable for output)
(define get-rep
  (lambda ()
    (list *self* index (cons sign (-> term 'get-rep))
          (cons 'links: links))))

Term Object Definition

; The term object.
; We use this representation for predicate and function terms.

(ObjectDef term
  (supers object)
  (variables (arguments ()))
    (arity 0)
    (name ())
    (var-list ())
    (varflag #!false))
  (methods (make-term (make-term (1st args)))
           (initialize (initialize (1st args) (2nd args)))
           (var? varflag)
           (expr? (not varflag))
           (get-arity arity)
           (get-var-name (and varflag name))
           (get-constructor (and (not varflag) name))
           (get-arguments (and (not varflag) arguments))
           (get-rep (get-rep))
           (make-term (make-term (1st args)))
           (initialize (initialize (1st args) (2nd args))))
  (localfns

112
APPENDIX: CONNECTION GRAPH IMPLEMENTATION

; get a new term to represent a particular expression
(define make-term
  (lambda (expr)
    (cond ((and (atom? expr)
      (eq? 63 (1st (string->list (symbol->string expr))))))
      ; the expression is a variable, check a term exists
      (let ((temp (assq expr var-list)))
        (cond (temp (cdr temp))
          (else (let ((new-term (-> term ':new))
            (-> new-term 'initialize expr #!true)
            (set! var-list
              (cons (cons expr new-term) var-list))
            new-term)))))
      ; the expression is not a variable
      (else (-> (-> term ':new) 'initialize expr #!false))))

; initialize-term initializes a term object from an expression
(define initialize
  (lambda (expr vf)
    (set! varflag vf)
    (cond ((atom? expr)
      (set! name expr)
      *self*)
      (else
        (cond ((not (atom? (1st expr)))
          (objecterror
            "first position of term is not atomic: "
            (1st expr))))
        (set! name (1st expr))
        (set! arguments
          (map (lambda (argument)
              (-> term 'make-term argument))
            (rest expr))
        (set! arity (length arguments))
        *self*))))

; get a representation of this term (suitable for output)
(define get-rep
  (lambda ()
    (cond (varflag name)
      ((eq? arity 0) name)
      (else (cons name
        (map (lambda (argument)
          (-> argument 'get-rep))
            arguments)))))))

113
Unifier Object Definition

; copyright Ted Moens 1986

; the unifier object for connection graphs

(ObjectDef unifier
  (supers object)
  (methods (unify (unify (1st args) (2nd args) (3rd args)))))

(localfns

; unify - unify two sets of terms with respect to a binding
; environment.
; This algorithm does not incorporate the "occurs check"

(define unify
  (lambda (tms1 tms2 env)
    (cond ((and (null? tms1) (null? tms2)))
      ((or (null? tms1) (null? tms2)) #f)
      (else
        (let ((tl (1st (1st tms1)))
              (il (rest (1st tms1)))
              (t2 (1st (1st tms2)))
              (i2 (rest (1st tms2))))
          (cond ((-> tl 'var?)
                (let ((binding (-> env 'lookup tl il)))
                  (cond (binding
                                 (unify (cons binding (rest tms1)) tms2 env))
                    (else
                      (-> env 'bind tl il t2 i2)
                      (unify (rest tms1) (rest tms2) env))))))
          ((-> t2 'var?)
           (unify tms2 tms1 env))
          ((and (eq? (-> t1 'get-constructor)
                     (-> t2 'get-constructor))
               (eq? (-> t1 'get-arity)
                    (-> t2 'get-arity)))
           (unify
            (append! (rest tms1)
                      (map (lambda (tm)
                             (cons tm il))
                             (-> t1 'get-arguments)))
            (append! (rest tms2)
                     (map (lambda (tm)
                            (cons tm i2))
                            (-> t2 'get-arguments)))
           env))

114
APPENDIX: CONNECTION GRAPH IMPLEMENTATION

Substitution Object Definition

; the substitution object contains a variable substitution.
; it can be used to represent a binding environment or
; a unifier of two terms.

(ObjectDef substitution
  (supers object)
  (variables (vars ()) (trms ()))
  (methods (make-substitution (-> *self* ':new))
    (reset (set! vars '()) (set! trms '()) #!true)
    (lookup (lookup (1st args) (2nd args) vars trms))
    (bind (bind (1st args) (2nd args) (3rd args) (4th args)))
    (get-updated-bindings
     (get-u-b (1st args) (2nd args) (3rd args)))
    (get-rep (list vars trms)))

(localfnfs

; bind - binds a variable to a term in the environment
(define bind
  (lambda (var index1 trm index2)
    (set! vars (cons (cons var index1) vars))
    (set! trms (cons (cons trm index2) trms))))

; lookup - lookup the value of a variable in a binding environment.
; returns the binding of the variable or #!false if unbound.
(define lookup
  (lambda (var index vs ts)
    (cond ((null? vs) #!false)
      (else (cond ((eq? index (cdar vs))
        (cond ((eq? (-> var 'get-var-name)
          (-> (caar vs) 'get-var-name))
            (car ts))
          (else
            (lookup var index (cdr vs) (cdr ts))))))))

; return a copy of the bindings in this substitution with the
; indexes updated
(define get-u-b
  (lambda (cutoff low-update high-update)
    (cons (update vars cutoff low-update high-update)
      (update trms cutoff low-update high-update))))
APPENDIX: CONNECTION GRAPH IMPLEMENTATION

(define update
  (lambda (a-list cutoff low-update high-update)
    (map (lambda (el)
           (let ((index (cdr el)))
             (cond ((<=? index cutoff)
                   (cons (car el)
                         (+ low-update index)))
                   (else
                    (cons (car el)
                          (+ high-update index))))))
     a-list))))

Connection Graph Input Processor

; This function converts input clauses to a list of term objects.
; The clause: \{~P(x,f(a)) Q(y) R(f(g(h(a x) y) b))\}
; Would be input as: ((~ (P ?x (f a))) (Q ?y) (R (f (g (h a ?x) ?y) b)))
; This will be converted into a list of term objects:
; ((- term.1) (+ term.2) (+ term.3))
; where each term.i represents a literal in an input clause.
; Of course, each term.i can itself be composed of several term objects.

(define convert-cg-input
  (lambda (clause-list)
    (map (lambda (lit-list)
           (map (lambda (literal)
                  (map (lambda (literal)
                        (cond ((eq? (1st literal) '\~)
                              (list '\- (-> term 'make-term (rest literal))))
                              (else
                               (list '+' (-> term 'make-term literal)))))
                       lit-list)))
        clause-list)))

116
Stack Object Definition

; A stack object defined for the Objects language.

(objectdef stack
(supers ()))
(variables (s ()))
(methods (push (set! s (cons (1st args) s))
#t)
(pop (cond (s (let ((ret (1st s)))
(set! s (rest s))
ret))
(else 'empty)))
(top (cond (s (1st s))
(else 'empty)))
(remove (remove (1st args) s))
(contents s)
(empty? (null? s))
(size (length s))))

Queue Object Definition

; A stack object defined for the Objects language.

(objectdef queue
(supers ()))
(variables (q ()))
(methods (add (set! q (append q (cons (1st args) ())))
#t)
(delete (set! q (remove (1st args) q)) #t)
(pop (cond (q (let ((ret (1st q)))
(set! q (rest q))
ret))
(else 'empty)))
(top (cond (q (1st q))
(else 'empty)))
(remove (remove (1st args) q))
(contents q)
(empty? (null? q))

APPENDIX: CONNECTION GRAPH IMPLEMENTATION

(sorted-a-list)

; a-list object which is sorted by the value field

(objectdef sorted-a-list
  (supers ()))
  (variables (a-list ()))
  (methods (add (set! a-list (insert (1st args)
    (2nd args)
    (delete (1st args) a-list))) #!true)
    (delete (set! a-list (delete (1st args) a-list)) #!true)
    (pop (cond (a-list (let ((ret (1st a-list)))
      (set! a-list (rest a-list))
      ret))
      (else 'empty)))
    (top (cond (a-list (1st a-list))
      (else 'empty)))
    (contents a-list)
    (empty? (null? a-list))
    (size (length a-list)))

(localfns

  (define insert
    (lambda (x v l)
      (cond ((null? l) (cons (cons x v) ()))
        ((>? v (rest (1st l))) (cons (cons x v) l))
        (else (cons (1st l) (insert x v (rest l)))))))

  (define delete
    (lambda (x l)
      (let ((rem (assq x l)))
        (cond (rem (remove rem l))
          (else l)))))))
Bibliography


122