DEPICTION AND DOMAINS IN
VISUAL KNOWLEDGE REPRESENTATION

By

GLADYS MAGALI WONG
B.Sc. Honours, University of British Columbia, 1984

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
September 1986
© Gladys Wong, 1986
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 September 8, 1986
Abstract

Systems need knowledge to behave intelligently in a complex environment. This thesis presents a formalism for characterizing the knowledge in a world while providing a framework that can be utilized efficiently in computation. Thus Domain Theory is developed as a knowledge representation scheme.

Under Domain Theory, the knowledge in a world is divided into domains that are interrelated through the relation of representation. This theory is evaluated as appropriate for knowledge representation using descriptive and procedural adequacy criteria.

Domain Theory is applied to produce a working system called Depicts. Depicts is written in Prolog which is well suited for implementing Domain Theory. Given a graphical representation, Depicts returns a relational description and, conversely, given a relational description, Depicts returns a graphical representation.
# Contents

Abstract

Contents

List of Figures

Acknowledgement

1 Introduction

2 Framework of Field
   2.1 Picture Grammars
   2.2 Origins of Domain Theory
   2.3 Prolog
      2.3.1 Prolog for Knowledge Representation
      2.3.2 Prolog in Computer Graphics
      2.3.3 Prolog and Graphics Standards

3 Domain Theory
   3.1 Definitions
   3.2 Domain Theory Described
      3.2.1 Domain
      3.2.2 Relation of Representation
      3.2.3 Mapping
      3.2.4 Equivalence Classes
      3.2.5 Formation of Composite Objects
      3.2.6 An Example of the Formation of a Composite Object
   3.3 Descriptive Adequacy
   3.4 Procedural Adequacy
      3.4.1 Flexibility
4 Why Prolog?  39
   4.1 Overview of Prolog ................................................. 40
   4.2 Descriptive Adequacy .............................................. 42
   4.3 Procedural Adequacy ............................................... 44
      4.3.1 Flexibility ..................................................... 46
   4.4 Problems in Prolog ................................................ 49
5 Description of Implementation  52
   5.1 Image Domain and Scene Domain ..................................... 53
   5.2 Data Structures .................................................... 55
   5.3 Tools ............................................................... 57
   5.4 Environment ....................................................... 60
      5.4.1 The Editor ................................................... 60
      5.4.2 The Interpreter ............................................... 65
   5.5 Applications ....................................................... 66
      5.5.1 Computer Graphics ........................................... 66
      5.5.2 Expert systems ............................................... 69
      5.5.3 Databases ..................................................... 71
6 Evaluation and Possible Future Extensions  74
   6.1 Evaluation ........................................................ 74
   6.2 Possible Future Extension to Depicts ............................ 76
References  78
A An Example of a Composite Object  83
B A Programming Example  85
C A Sample Session with Depicts  87
# List of Figures

3.1 Schematic Representation of Domain Theory ........................................ 23
3.2 Mapping of $R'$ between $G$ and $L$ .................................................. 27
3.3 Mapping of $R''$ between $G$ and $N$ ............................................... 29
3.4 Mapping of $R$ among $G$, $L$ and $N$ ............................................... 30

5.1 Schematic Diagram of Data Structure ............................................... 58
5.2 Depicts System Diagram ................................................................. 60
5.3 Depicts Display on a Workstation .................................................... 62
5.4 Hierarchy of Command Menus ........................................................... 63

C.1 Partial Tree Diagram of British Columbia Trees ............................... 90
C.2 Partial Venn Diagram of British Columbia Trees ............................... 91
Acknowledgement

I would like to thank Dr. A. K. Mackworth, my supervisor, for guiding me in my research while giving me freedom to explore. He has given me a renewed appreciation for Artificial Intelligence.

I am very grateful to my parents and siblings for their invaluable help.

And to my friends at the department of Computer Science, a big thank you for the profitable discussions, the helpful suggestions, and all the good times.
Chapter 1

Introduction

Artificial Intelligence is defined as the design of systems that represent, use, and acquire knowledge to perceive, reason, act and communicate (Mackworth, 1985b). It follows that there is a need to represent what systems must know in order for them to behave intelligently in a complex environment. A core issue in Artificial Intelligence is defining a set of conventions for an adequate knowledge representation.

A good knowledge representation should capture a situation in the world so that the representation corresponds to the actual situation in the world. Furthermore, the knowledge should be organized in such a way that an intelligent machine can come to new conclusions about its environment by formally manipulating these descriptions (Brachman and Levesque, 1985).

The problem is finding a knowledge representation that captures the knowledge in the world and that can be used efficiently in computation. In this thesis, we present a scheme for knowledge representation called Domain Theory which divides the knowl-
CHAPTER 1. INTRODUCTION

edge in the world into domains. The elements in the domains are related through a relation of representation. We also show why Domain Theory is adequate for knowledge representation.

We have applied Domain Theory specifically to the fields of computational vision and computer graphics to produce a working system which we have called Depicts. In general, a computational vision system analyzes an image to produce a description of what it "sees" in an image. In contrast, a computer graphics system synthesizes an image to illustrate graphically what it "understands" in a description. Depicts is flexible in the sense that it can perform both analysis and synthesis.

When Depicts is given a graphical representation, it will act like a vision system and it will produce as output a description of the graphical representation. But when Depicts is given a description of a scene, it will act like a graphics system and it will produce as output a picture of the scene. After all, both computational vision and computer graphics are firstly and ultimately interested in pictures (Nake and Rosenfeld, 1972) or images.

Prolog is used as the programming language for implementing Domain Theory in Depicts. Prolog was chosen because Horn Clauses lend themselves to describing Domain Theory in a natural way. Furthermore, a variable in Prolog can serve either as input or as output depending on its initial value at the time of the procedure call. This Prolog feature is useful to Depicts since it allows for true bidirectional knowledge
flow. As a result, Depicts can be neutral with respect to analysis and synthesis.

Chapter 2 defines the criteria used for evaluating a knowledge representation. These criteria are used to judge picture grammar systems which are functionally similar to Depicts. The origins of Domain Theory are presented next. In the last part of this chapter examples of systems written in Prolog are reviewed.

Chapter 3 describes Domain Theory. It shows why Domain Theory is adequate for representing knowledge.

Chapter 4 gives a brief overview of the programming language Prolog. It then presents the reasons why Prolog is suitable for implementing Domain Theory.

Chapter 5 explains how Domain Theory has been used in computer graphics and computational vision with the system Depicts. The actual implementation of Depicts is described in terms of the tools used to develop Depicts, and the environment created by Depicts. Finally, possible applications of Depicts are suggested.

Chapter 6 evaluates Domain Theory for knowledge representation, Prolog as a programming language for implementing Domain Theory, and the system Depicts. This chapter also points to possible work that might be pursued in the future.
Chapter 2  
Framework of Field  

Havens and Mackworth (1983) advocate that *descriptive* and *procedural* adequacy criteria are necessary for evaluating a knowledge representation. It is necessary that knowledge be effectively organized, and efficiently used in computation.

A knowledge representation KR is descriptively adequate if *all* the information in a world W is representable through KR. KR should define the *elements of discourse* and the *constraints/rules* on these elements. The structure of KR needs to be finite so that it will be machine computable. Hence, KR must be *generative* in order to be finite and yet be capable of characterizing an arbitrary amount of information (Havens, 1985).

Descriptive adequacy criteria are used for judging how the knowledge is *represented*. They seek to evaluate how well a situation in the world is captured using a knowledge representation scheme. In contrast, procedural adequacy criteria are used for judging how the knowledge is *used*.

KR is procedurally adequate if it provides a framework that can be used efficiently
in time and space by a machine. Flexibility of use of the available information is another characteristic of a procedurally adequate knowledge representation: there should be freedom in the flow of knowledge depending on the knowledge sources. Other aspects of procedural adequacy are control facilities provided by the knowledge representation, and the ease of reprogramming a system. A typical example of reprogramming is changing a system so that instead of performing image synthesis, it will perform image analysis (Mackworth, 1983b).

As expected, the descriptive and procedural adequacy criteria of knowledge representation are apparently conflicting. There is a tradeoff so that favouring descriptive adequacy might mean sacrificing procedural adequacy, and vice versa. For instance, a knowledge representation might be able to capture the knowledge in the world in an unusually complete manner and thus be descriptively adequate. Nonetheless, this knowledge representation might not be useful if it is procedurally inadequate because it cannot be embodied for a computationally effective algorithm to use.

Traditionally, people writing application programs have concentrated almost entirely on the efficiency of their programs and as a result, little attention has been paid to flexibility and descriptive adequacy. Those doing theoretical work in knowledge representation often emphasize descriptive adequacy disregarding the practicality needed of any useful knowledge representation. We realize that we cannot find a knowledge representation that is perfectly adequate descriptively and procedurally; however, we
seek to optimize both criteria: this is the *raison d'être* of Domain Theory.

*Depicts* implements Domain Theory for knowledge representation. Given a graphical representation, *Depicts* returns a relational description and conversely, given a relational description, *Depicts* returns a graphical representation. We say, therefore, that *Depicts* can perform both image synthesis and image analysis. *Depicts* is neutral with respect to the *analysis/synthesis question*.

In section 1 we review systems of the 1960's and early 1970's that are functionally similar to *Depicts*, in that they seek to perform both image synthesis and image analysis. These systems too, are neutral with respect to the analysis/synthesis question; however, since they are based on picture grammars, they suffer from a poor descriptive and procedural knowledge representation. The representational problems of picture grammars are so serious that the linguistic approach to picture analysis has been abandoned altogether, in favour of a more relational knowledge representation.

Section 2 explains work by Clowes and Stanton, and their idea of having a picture domain and a scene domain with a mapping relation between the two domains. Mackworth extends and formalizes this idea.

Section 3 describes some of the work done in Prolog. Although far from being exhaustive, this survey will help illustrate the emergence of Prolog in the fields of artificial intelligence and computer graphics.
2.1 Picture Grammars

Minsky (1961) presented a paper called "Steps Toward Artificial Intelligence", where he described a knowledge representation for pictures. He suggested that any pattern fragment could have relationships with other fragments and at the same time be itself composed of other fragments.

Kirsch (1964) was not satisfied with Minsky's suggestions for representing pictures because they did not lead to "a syntactic theory having the clarity of a generative grammar such as was exhibited for English text". He claimed that the syntactic structure of text and of pictures could be described in similar ways. So, he suggested that text and pictures be analyzed in the same way, namely with generative grammars. Kirsch and others like Ledley (1964) began to use this linguistic approach to picture representation.

Before the 1960's "image analysis" meant classifying pictures into categories. The linguistic approach brought about structural description of pictures with the so-called picture description languages. Typically, a class of pictorial objects would have a picture grammar, various images would be parsed using this grammar, and the resulting parse trees would provide a structural description of the images.

Shaw (1970) describes his picture description language, PDL, that consists of picture primitives, each with a head and a tail. The picture elements are then described
using these picture primitives which are connected at the head or at the tail.

Unfortunately, like most picture description languages, PDL has only a few relations which can be described among the picture elements. These relations are usually concatenation or connectivity relations. In addition, the pattern composition range of relationships is restricted considerably. As a result, these relations are inadequate for two-dimensional images. Picture description languages are not descriptively adequate since they cannot represent the image domain properly.

The analysis process directed by the picture grammar is called picture parsing. Shaw emphasizes that the same picture description language can be used for both analysis and generation purposes. However, Shaw does not discuss how his PDL can be used for generating pictures given a certain parsing process. He only mentioned that the same "description formalism" can be used. Procedurally it is not clear that picture parsing can be reversed into picture generation. However, interestingly enough, descriptively we can say that grammars are neutral for analysis and synthesis.

George (1972) presents a graphical meta system using a picture description language. He writes that even though most linguistic picture processing systems emphasize the analysis of pictures and not their generation, his model allows for "generation and recognition to coexist within the same application". There is symmetry between generation and recognition in George's system; nevertheless, as with all linguistically based systems, the mapping between the description and the image is embedded in
the semantics of the parse tree and not in the grammar. Descriptively this presents a problem because there is a lack of formalization.

There are serious, unresolved problems with this linguistic approach to picture analysis. First of all, it is not clear what the image primitives should be. Secondly, the relations which exist in the picture between the primitives are not clearly represented. In fact, these relations are only implicitly expressed in the way the symbols of the grammar are ordered.

Among the other systems that provided the user with commands for analyzing and synthesizing drawings was O’Callaghan’s interactive system (1972), which used a picture language to produce descriptions from line drawings. These systems lacked the ability to let the user define his own interpretation for line drawing. O’Callaghan admitted that if the current interpretation of drawings were to be changed, then considerable modification would be required, “particularly to the organization and content of the data structure”. It is interesting to note that O’Callaghan did recognize the notion of mapping but he simply stated that extensive work would be required to incorporate it.

2.2 Origins of Domain Theory

Clowes (1969) was concerned with characterizing a language in terms of primitive relationships as opposed to primitive atoms (as was done in picture grammars). He
stated that it is impossible to represent a picture as a set of independent parameters. By proposing two domains with a mapping relation between them, Clowes (1970) was departing from Chomsky's syntactic theory. He called the mapping relation the relation of representation and the two domains the picture domain and the scene domain.

Clowes noted that the relation between picture fragments and scene fragments is a one-to-many mapping because there are several interpretations for a single picture. Thus he described the interpretation problem as recovering one or more scene structures from the corresponding single picture.

Not satisfied with picture grammars, Stanton (1970) wrote his own system called RAMOS. In RAMOS, a situation is described as a complex of objects, their relationships and their attributes. A situation in one domain can represent a situation in another domain with a relation of representation between them. Since the relation of representation is between situations, it is possible for an object to map to an object, for a relationship to map to a relationship, or for an object to map to a relationship. Several situations are bound together by the relation of representation to form definitions and these definitions are subdivided into several domains. The framework is completed by defining the relations between the situations in the picture domain and those in the problem domain.

Stanton recognized that the analysis/synthesis claim was very appealing but he could not accept the way it was developed by others with their picture grammars.
CHAPTER 2. FRAMEWORK OF FIELD

Therefore in RAMOS, he combined descriptive structures of pictures with some primitive operations over a data base. Relations among image objects were expressed as combinations of the primitive operations in a way which anticipated the use of logic programming as Prolog (Browse, 1982).

Stanton (1972) also identified the need to invert predicates among image descriptions but he did not give a concrete answer to this issue. To him the recovery procedure dealt with information in the problem domain, and generating a picture would be a question of inverting the recovery procedure. It would not have been difficult to reverse the recovery procedure if it had consisted simply of rewrite rules; unfortunately, this did not generalize to all cases.

Parenthetically, the term relation of representation was first used by Kūng in his book Ontology and the Logistic Analysis of Language (1967). He wrote about the relation of representation that exists between the "symbols of a text and the reality described by it". He described the relation of representation of predicate signs of logistic languages where he dealt with "the semantic relation of these signs to the reality designated by them: what do the signs represent?"

Mackworth (1983a) presents Domain Theory as a form of knowledge representation and he applies it specifically to computational vision. He explains how perception involved three types of domains: the domains of possible worlds, possible images, and possible projections. He then applies this general view of perception to vision in various
ways factoring the world domain and the projection domain into subdomains. He describes how the relation of representation or depiction relation (Mackworth, 1983b) between these domains constrains the elements of the domains.

In his paper on “The Correspondence Continuum”, Smith (1986) deals with a theory of correspondence that is able to support an indefinite continuum of circumstantially dependent relations. These relations are between states of affairs in the domains. His correspondence theory emphasizes descriptive adequacy “providing an informationally rich account of the structure of the relation” but procedurally, the theory remains “entirely silent on any question of computing this relation”. Smith’s approach is much more theoretical than ours. In this thesis, we do describe a theory, however we also have a practical working system that uses Domain Theory for knowledge representation.

MacLennan (1981) introduces relational programming, that is, a style of programming for manipulating entire relations rather than individual data. He describes a set of operations on relations such as intersection, union, and composition. In his paper, relational programming is not discussed for knowledge representation but more as an introduction of relational calculus aspects into programming. MacLennan feels that an implementation of relational programming is somewhat premature since the effectiveness of relational programming has not yet been determined precisely. MacLennan’s work is relevant to this thesis as we deal with relations and operations on them.
CHAPTER 2. FRAMEWORK OF FIELD

In the next chapter, we develop on the idea of dividing the knowledge of the world into domains and relations of representation the way Clowes, Stanton and Mackworth did. However, we go one step further and show that it is possible to generalize all the elements of the domains to a single type of object. Unlike Stanton, we do not distinguish between various types of elements.

2.3 Prolog

Prolog (Kowalski, 1982) stands for Programming in logic. It is a programming language that was developed in the mid-seventies. Prolog has received much attention in the last few years because the Japanese chose it for their core programming language in the Fifth Generation Computer Systems Project. This 10-year endeavour is very ambitious and promises to produce revolutionary hardware and software. In this project, logic programming is “envisioned as the link between the fields of software engineering, database systems, computer architecture, and knowledge engineering”. (Cuadrado and Cuadrado, 1985).

2.3.1 Prolog for Knowledge Representation

Prolog has not been used extensively for knowledge representation in artificial intelligence. This is somewhat surprising considering that Prolog is based on logic and that formal logic has been used frequently in knowledge representation. Even when not positively acclaimed, logic has at least been alluded to in many of the papers on
knowledge representation (Brachman and Levesque, 1985).

One example where Prolog is used for knowledge representation in computational vision is the system of Cuadrado and Cuadrado (1985). They describe a vision system that uses a *frame-based knowledge representation* to handle all the components in the high-level processing of a vision hierarchy. This frame-based system is implemented in Prolog and includes such features as inheritance and demons. The system classifies houses according to a very small, fixed number of models. The system then makes recommendations as to how some sample houses should be modified so that they will conform more closely to the models. As with many systems, the main concern is procedural efficiency – attention is given neither to descriptive adequacy nor to flexibility of knowledge flow.

### 2.3.2 Prolog in Computer Graphics

Davis (1982) uses Prolog as a design tool for formal problem specifications in software engineering. She suggests that the user specify his problem in Prolog, and that he *run* his problem specifications, namely his Prolog program, when he wants the answer to a query. Running the specifications directly contrasts with other methods of problem specification. With Prolog, the designer of the specifications may pose the questions directly.

Davis gives as an example the Prolog specification of a high-level interface to a
flexible display. She shows how the user can test his design visually. Since the user can run the Prolog specifications, when he wants to know how some pictures will look, then he simply needs to construct his pictures and look at the result. He does not need to formalize his questions the way he would if he were to use other approaches such as algebraic axioms. As a result, even an inexperienced user can manipulate images interactively.

Eggert and Chow (1983) state that logic programming's applications in knowledge-based systems will require heavy usage of images for effective human-machine interfaces, and future graphics workstations will need simple but efficient methods for describing images. Logic programming methods provide a promising way for novice users to manipulate images cleanly and concisely. They seem to fill a certain gap in current picture specification languages.

Eggert and Chow describe their interactive graphics logic programming system that performs image synthesis. They use this system to display the contents of an infinite tree up to the resolution of the output device. They do this by exploiting the fact that GLOG (a version of C-Prolog with graphics capabilities) lacks an occur check in the unification algorithm. The paper does not mention anything about the system's possible flexibility.

Eggert and Chow emphasize the importance of a logic programming system that

---

1cf. section 4.4 for a description of the occur check
performs both image analysis and image synthesis: it would be a powerful tool for building future knowledge based systems. Their system only performs image synthesis. They leave the harder part, analysis, as a research issue which is being explored by them.

Franklin (1986) writes how he uses Prolog for geometric and graphics implementations. He has implemented several algorithms in Prolog including a subset of GKS, convex hull finding, planar graph traversal, recognizing groupings of objects, and boolean combinations of polygons using multiple precision rational numbers. He concludes that although not perfect, Prolog is a powerful tool for increasing the efficiency of graphical and geometrical algorithms. This allows for harder problems to be solved in a given time.

Gonzales, Williams and Aitchison (1984) evaluate Prolog for writing CAD tools such as two-dimensional drafting packages, three-dimensional geometric modelling packages, etc. They conclude that Prolog is more concise, more readable and clearer than Pascal, and also more efficient on the compilers they used. Furthermore, they report that reliable programs could be developed faster using Prolog than using Pascal.

Franklin, and Gonzales et al. use Prolog to describe knowledge procedurally. Emphasis is placed on producing algorithms that are efficient; even though this is important, there is also a need for descriptive adequacy.
2.3.3 Prolog and Graphics Standards

An important goal in the field of computer graphics is to produce device independent graphics packages. In other words, graphics routines should be executable on all machines and should not depend on "features" particular only to certain devices. When it was recognized in the mid-seventies that there was a need for device-independent graphics packages, the ACM SIGGRAPH Committee produced a Core Graphics System (abbreviated to Core) which is a proposed standard for graphics programming. The ISO set forth the Graphical Kernel System (or GKS for short) which later became the first International Standard in computer graphics and is now gaining acceptance in practice (International Organization for Standardization, 1985).

Both Core and GKS consist of functions for the manipulation of data structures in graphics and for the management of graphical devices. Traditionally, Core functions and GKS functions could only be called from procedural languages like FORTRAN, PASCAL and C. However, in the last few years, people have written interfaces for the low level Core and GKS graphics routines so that they can be called in Prolog. There seems to be a greater awareness of the need for extending the Prolog capabilities towards graphical applications (Hübner and Markov, 1986).

GKS-Prolog (Hübner and Markov, 1986) is an integration of GKS into Prolog. In GKS-Prolog, most GKS functions have been defined as built-in predicates. The user calls the graphics functions the same way he calls any other Prolog predicate. See
(Pereira, 1982) and Nichols' GKS Prolog (1985) are two other versions of Prolog that incorporate GKS graphics routines.

In theory, Prolog programs written in GKS-Prolog, SeeLog, or Nichols' GKS Prolog should be portable since they use graphics routines that should be based on standardized graphics functions. However, we feel that in practice they are not portable. The calling sequences are too different and the effects of the graphics predicates are not the same. In subsection 5.5.1 we describe the idea of image domain independence which is a step beyond device independence in computer graphics.
Chapter 3
Domain Theory

The first section of this chapter defines the terms necessary to describe Domain Theory.

The second section presents Domain Theory. The last two sections show how Domain Theory is descriptively and procedurally adequate.

3.1 Definitions

We have adopted the notation and definitions of set theory used in Stanat and McAllister (1977) and in Stoll (1979), with a few extensions to some of the original definitions.

Definition: An ordered \( n \)-tuple is a sequence of \( n \) objects which is symbolized by \(< w_1, w_2, w_3, \ldots, w_n >\). For \( n = 2 \), the ordered \( n \)-tuple is called an ordered pair. An ordered pair of \( x \) and \( y \) denoted by \(< x, y >\) is uniquely determined by \( x \) and \( y \). Moreover, if \(< x, y > = < u, v >\), then \( x = u \) and \( y = v \).

Definition: Let \( A \) and \( B \) be sets. The Cartesian product or cross product of the sets \( A \) and \( B \) is denoted by \( A \times B \), and it is defined as a set of ordered pairs, \(< x, y >\),
such that $x$ is in $A$ and $y$ is in $B$; symbolically: $A \times B = \{ <x, y> \mid x \in A \text{ and } y \in B \}$.

**Definition**: Let $A$ and $B$ be sets. A *binary relation* $R$ *from $A$ to $B$* is a subset of $A \times B$. $R$ is a set of ordered pairs and we denote membership by $<x, y> \in R$.

A natural generalization of a binary relation is an *$n$-ary relation* as a set of ordered $n$-tuples. In this section we restrict ourselves to binary relations for didactic reasons; however, it is very simple to extend all the definitions from being binary ($n = 2$) to a higher order ($n > 2$).

From this point on the unqualified term *relation* will denote *binary relation*. Relations that are not binary will be specified by such terms as *$n$-ary relation*. A 0-ary relation is called a 0-place relation.

**Definition**: A *statement* is a declarative sentence capable of being classified as either *true* or *false*. A 2-place predicate, $P(x, y)$, is an expression having the quality that on assignment of values to the variables $x$ and $y$ from appropriate domains a *statement* results.

**Definition**: In the *intuitive principle of abstraction*, a predicate $P(x)$ defines a set $S$ by the convention that the members of $S$ are exactly those elements $s$ such that $P(s)$ is a true statement. Therefore, the set of all ordered pairs that are members of $R$ can be defined as $\{ <x, y> \mid R(x, y) \}$. Thus, $<a, b> \in \{ <x, y> \mid R(x, y) \}$ iff $R(a, b)$ is a true statement. Every relation on a set $S$ corresponds to a predicate with $S$ as the universe of discourse. In this thesis, we will use relation and predicate interchangeably.
CHAPTER 3. DOMAIN THEORY

Definition: Let $R$ be a relation on $A \times B$. The domain of $R$ symbolized by $D_R$ is \{ $x \mid \exists y \; R(x,y)$ \} and the range (or co-domain) of $R$ symbolized by $R_R$ is \{ $y \mid \exists x \; R(x,y)$ \}. Note that $D_R \subseteq A$ and $R_R \subseteq B$.

Example

Let $A$ and $B$ be the set of integers and let $R$ be \{ $< x, y > \mid x \in A \land y \in B \land y = x^2$ \}. The relation $R$ can be described as \{ $..., < -2, 4 >, < -1, 1 >, < 0, 0 >, < 1, 1 >, < 2, 4 >, ...$ \}. The domain of $R$ is the set of integers and the range of $R$ is \{ $0, 1, 4, 9, 16, ...$ \}

Definition: If $x \in A$, then $x$ is an object of the set $A$. Every object is a relation which could be expanded to a set. This allows for an arbitrary nesting of sets. If the object is a constant, then the object is atomic (i.e., a 0-place relation), otherwise the object is composite (i.e., an $n$-ary relation, for $n > 0$).

Definition: Let $A$ and $B$ be sets, let $R$ be a relation on $A \times B$, and let $x \in A$ and $y \in B$. There is a mapping from $A$ to $B$ iff $R(x,y)$ holds. If $x$ and $y$ are composite objects, then the mapping needs to be done recursively at every level of the nesting for $R(x,y)$ to hold.

Definition: Given a relation $R$ from $A$ to $B$, let $D_R$ be the domain of $R$ and let $R_R$ be its range. Let $x_i, x_j \in D_R$ and let $y_k, y_l \in R_R$.

A relation $R$ on $A \times B$ is many-to-one (or functional from $A$ to $B$) if
CHAPTER 3. DOMAIN THEORY

\( R(x_i, y_k) \) and \( R(x_i, y_l) \in R \) implies \( k = l \)

A relation \( R \) on \( A \times B \) is **one-to-many** if

\( R(x_i, y_k) \) and \( R(x_j, y_k) \in R \) implies \( i = j \)

A relation \( R \) on \( A \times B \) is **one-to-one** if both of the above.

**Definition:** If \( R \) is a relation from \( A \) to \( B \), then the **inverse of a relation**, symbolized by \( R^{-1} \), is the relation from \( B \) to \( A \) such that, \( R^{-1}(y, x) \) iff \( R(x, y) \). If the mapping of \( R(x, y) \) is **one-to-one**, then the mapping of \( R^{-1}(y, x) \) is also **one-to-one**.

We say that there is a **bidirectional flow of knowledge** when relation \( R \) is one-to-one. If there are more than two domains and their mapping relation is one-to-one, then there is a **multidirectional flow of knowledge**.

**Definition:** Let \( R \) be a relation on a set \( A \times B \) with domain \( D_R \) and with range \( R_R \). For every \( b \in R_R \), the **equivalence class of \( b \) with respect to \( R \)**, symbolized by \([AB]_b,\) is \( \{ x \mid R(x, b) \} \)

**Properties of Equivalence Classes:**

The equivalence classes are **exhaustive** because their union forms the complete original set: \( D_R = \bigcup_{b \in R_R} [AB]_b \). They are **exclusive** since each element of the set \( D_R \) can belong to exactly one equivalence class: \([AB]_{b_i} \cap [AB]_{b_j} = \emptyset \), where \( i \neq j \).
3.2 Domain Theory Described

Under this theory, the knowledge of a world is divided into structures called domains that are interrelated by a relation.

![Schematic Representation of Domain Theory](image)

Figure 3.1: Schematic Representation of Domain Theory

3.2.1 Domain

A *domain* is a set of objects. We can think of a domain as a cluster of knowledge. Objects in one domain are related to objects in other domains by the corresponding
CHAPTER 3. DOMAIN THEORY

relation between the domains. Figure 3.1 illustrates a collection of domains connected by a relation $R$.

3.2.2 Relation of Representation

As described in section 2.2, the mapping relation that relates objects across domains is called the relation of representation. We can define a procedure using this relation as follows,

Find the set of objects, $x_1, x_2, \ldots, x_n$, from each of the $n$ domains, $D_1, D_2, \ldots, D_n$, respectively, such that the $n$-ary relation $R(x_1, x_2, \ldots, x_n)$ holds.

Furthermore, we can formalize the above procedure to determine the truth value of the formula

$$\exists x_1 \exists x_2 \ldots \exists x_n ((x_1 \in D_1) \land (x_2 \in D_2) \land \ldots \land (x_n \in D_n) \land R(x_1, x_2, \ldots, x_n)) \quad (3.1)$$

We will call the process of solving this formula the computational task of Domain Theory.

A way of viewing the computational task is as a search problem. At the initial state some of the variables $x_i$ may already have values. At the goal state, all the $x_i$ will have values and $R(x_1, x_2, \ldots, x_n)$ will hold. This implies that there is an $n$-tuple such that $<x_1, x_2, \ldots, x_n> \in R$. The operator that transforms a state into the next state is the membership test $<x_1, x_2, \ldots, x_n> \in R$?. The search space consists of all possible $n$-tuples which are defined as elements of $D = D_1 \times D_2 \times \ldots \times D_n$. 
The ideal case is for the relation of representation to be one-to-one, since this guarantees that the solution to the computational task will be unique. In addition, one-to-one relations of representation allow for multidirectional flow of knowledge. The flow of knowledge is not fixed in a certain direction: knowledge flows depending on the availability of information.

Unfortunately, having a one-to-one relation of representation among domains is not the typical case. In fact, the relations of representation are usually many-to-one. This implies that there are many possible solutions to (3.1), that is, there are many ordered n-tuples, \(<x_1, x_2, \ldots, x_n>\), that satisfy \(R\). Furthermore, the inverse of the relation is ambiguous and thus multidirectional knowledge flow is not possible. We say that this case is underconstrained.

3.2.3 Mapping

In section 3.1 we stated that there is a mapping from set \(A\) to set \(B\) if \(R(x, y)\) where \(x \in A\) and \(y \in B\). If \(x\) and \(y\) are composite objects, then \(x\) maps to \(y\) under \(R\) iff \(R(x, y)\) and there are some (possibly empty) relations, \(R^{x_i, y_j}\), such that the elements of \(x\) are mapped to the elements of \(y\) under these relations \(R^{x_i, y_j}\).

There is a difference between \(x\) mapping to \(y\) and the elements of \(x\) mapping to the elements of \(y\). For instance, let the set \(x\) be \(R_1(x', x'')\) and let the set \(y\) be \(R_2(y', y'')\). There is a mapping from \(x\) to \(y\) if \(R(x, y)\) holds, which means that there is a mapping
CHAPTER 3. DOMAIN THEORY

from \( x \) to \( y \) if \( R(R_1(x', x''), R_2(y', y'')) \) and \( R'(x', y') \) and \( R''(x'', y'') \) hold.

The relation of representation \( R \) maps objects across domains. If the objects are composite, then there needs to be relations, \( R^{x_i,y_j} \) that define the mappings between the elements of the various composite objects. If the relations \( R^{x_i,y_j} \) are not explicitly defined, then the elements of the composite objects are implicitly defined under the relation of representation \( R \).

3.2.4 Equivalence Classes

In many-to-one relations, all the many elements in the domain that map to the one element in the range can be grouped together into an equivalence class. An equivalence class is a finite formalism of a representation and yet an equivalence class may contain an infinite number of members in its set. An equivalence class provides a way of describing a set intentionally rather than extensionally \(^1\).

In Figure 3.2, there is a set \( G \) that consists of the graphical representations for digital logic gates. There is a set \( L \) that consists of logical expressions. There is also a many-to-one relation of representation, \( R' \), between the domain \( G \) and the range \( L \).

The domain \( G \) has equivalence classes that correspond to the logical expressions in \( L \). For instance, the logical expression \( y = x' \) can be represented graphically in many (in fact, infinitely many) ways depending on the (odd) number of inverters found in the circuit. Nonetheless, there is a unifying factor to these circuits in that they all map

\(^1\)where all the members of the set need to be listed
to the same element $y = z'$ in $L$. These circuits form an equivalence class of $y = x'$ with respect to $R'$. We will represent this equivalence class as $[GL]_{y=x'}$.

Equivalence classes lend themselves to efficient computation because elements sharing the same meaningful property are put into the same subspace. As a result, the total search space is structured making it possible to eliminate entire subspaces at once instead of having to examine every element in the subspace.

Figure 3.2: Mapping of $R'$ between $G$ and $L$
3.2.5 Formation of Composite Objects

The objects in the domains may be atomic or composite. As defined in section 3.1 composite objects are themselves relations whose arguments are objects. Hence, forming a new composite object is equivalent to defining a new relation. Composition of objects is a necessity because this is how Domain Theory accommodates generativity. Furthermore, being able to nest relations allows for set reduction of equivalence classes so that a many-to-one mapping may be reduced to the ideal case of a one-to-one mapping.

Let \( U(object_1, object_2, \ldots, object_n) \) be the universal relation of representation for a model \( W \). Each object is defined as:

\[
\begin{align*}
< \text{object} > & ::= < \text{atomic object} > \mid < \text{composite object} > \\
< \text{composite object} > & ::= < \text{relation}(object_1, object_2, \ldots, object_n) > \text{ for } n > 0 \\
< \text{atomic object} > & ::= < \text{relation} > \text{ (0-place relation)}
\end{align*}
\]

Composite objects are formed with relations. Often these relations come from \( \text{a priori} \) constraints that define acceptable situations in the world. A constraint is itself a relation, and often this relation takes other relations as its arguments. This does not present a conflict at all since in Domain Theory all objects are relations. Consequently, imposing a relation on relations is the same as imposing a relation on objects.
Chapter 3. Domain Theory

3.2.6 An Example of the Formation of a Composite Object

Here is an example of how a new relation was imposed on existing relations. Take two domains: the domain consisting of the graphical representations for digital logic gates and their components, \( G \), and the domain consisting of logical expressions, \( L \). A typical example of the many-to-one relation of representation, \( R' \), between the two domains is illustrated in Figure 3.2.

We can reduce the size of the equivalence class \([GL]_{y=z}\) by adding more knowledge in terms of a constraint that limits the number of gates allowed in the circuit. \( R'' \) is
this constraint and it is a relation that counts the number of gates found in a circuit. $R''$ maps elements from $G$ to the domain of number of gates per circuit, $N$. Figure 3.3 illustrates the relation $R''$.

Let $[GN]_1$ be the equivalence class that corresponds to 1 in $N$ through the mapping relation $R''$. Adding the constraint $R''$ to the existing relation $R'$ causes a new relation to be formed. This new relation, $R$, is a ternary relation that takes one argument from $L$, one argument from $N$ and one argument from the intersection of the equivalence classes $[GL]_{y=x'}$ and $[GN]_1$. Figure 3.4 illustrates the new relation $R$.

Figure 3.4: Mapping of $R$ among $G$, $L$ and $N
The relation \( R \) among \( G, L \) and \( N \) is one-to-one which is the ideal case. \( R \) has constrained the situation enough so that the computational task has a unique solution.

It is interesting to show how the relation \( R \) operates.

For example, from Figure 3.2 we have

\[
\forall x_1 \exists x_2 ((x_1 \in G) \land (x_2 \in L) \land R'(x_1, x_2))
\]

(3.2)

and from Figure 3.3 we have

\[
\forall x_1 \exists x_3 ((x_1 \in G) \land (x_3 \in N) \land R''(x_1, x_3))
\]

(3.3)

both (3.2) and (3.3) hold simultaneously to form

\[
\forall x_1 \exists x_2 \exists x_3 ((x_1 \in G) \land (x_2 \in L) \land (x_3 \in N) \land R'(x_1, x_2) \land R''(x_1, x_3))
\]

which is equivalent to

\[
\forall x_1 \exists x_2 \exists x_3 ((x_1 \in G) \land (x_2 \in L) \land (x_3 \in N) \land R(x_1, x_2, x_3))
\]

because from Figure 3.4

\[
\forall x_1 \forall x_2 \forall x_3 (R'(x_1, x_2) \land R''(x_1, x_3)) \equiv \forall x_1 \forall x_2 \forall x_3 (R(x_1, x_2, x_3))
\]

In this specific example, \( x_1 \) is unique and there is a one-to-one mapping among the three domains. As a result, we can write

\[
\exists x_1 \exists x_2 \exists x_3 ((x_1 \in G) \land (x_2 \in L) \land (x_3 \in N) \land R(x_1, x_2, x_3))
\]
which is a particular instance of (3.1).

$R$ is a composition of $R'$ and $R''$. Using $R$ instead of $R'$ and $R''$ reduces the search space. Thus, we can also think of $R$ as performing the set operation of intersection on the equivalence classes of $R'$ and $R''$.

When the mapping is many-to-one, it is possible to reduce the size of the corresponding equivalence class by forming composite objects (which are in themselves relations). Ideally, the knowledge added to perform the composition of objects will be sufficient to reduce the equivalence class to a singleton. The resulting mapping will be then one-to-one.

### 3.3 Descriptive Adequacy

In chapter 2 descriptive adequacy was defined as one of the criteria used in evaluating a knowledge representation. In this section we discuss why Domain Theory is descriptively adequate for knowledge representation.

All the knowledge found in a world $W$ is representable by Domain Theory. Domain Theory requires that for a given $W$, the knowledge in $W$ be divided into domains. In other words, all objects found in $W$ are grouped into separate domains. The elements of discourse for this knowledge representation are the objects of the domains. The constraints on the objects are the relations of representation defined across the domains.

This structure of domains and relations of representation is finite. However, there
is no upper bound on the number of elements that may be represented through Domain Theory. Equivalence classes are especially useful in this respect since they group objects together and yet they allow for infinitely many members per class.

Domain Theory’s knowledge representation is generative. It is possible to derive new composite objects. Formation of new composite objects is also useful when incorporating a priori constraints about the nature of acceptable solutions in the world (Mackworth, 1983a). Furthermore, being able to compose new objects is helpful when it is necessary to merge the knowledge corresponding to one situation but represented from two different “points of view”. Usually the merge is necessary to obtain a single, more complete representation of both situations.

Domain Theory is correct for knowledge representation as it is based directly on the knowledge of the world to be represented. Being generative, this knowledge representation is incremental and dynamic. At the same time, it is also finite.

3.4 Procedural Adequacy

Having established that Domain Theory is descriptively adequate, in this section we analyze Domain Theory's procedural adequacy. We show that the knowledge represented through Domain Theory can be used by processes.

The computational task of solving the formula

\[ \exists x_1 \exists x_2 \ldots \exists x_n ((x_1 \in D_1) \land (x_2 \in D_2) \land \ldots \land (x_n \in D_n) \land R(x_1, x_2, \ldots, x_n)) \]  

(3.1)
is a well defined problem. A machine can take this framework and compute a solution to the formula. In fact, several algorithms that can solve this formula already exist.

(3.1) is a boolean constraint satisfaction problem (Mackworth, 1985a). The constraint relation is $R$. The $n$ variables are $x_1, x_2, \ldots, x_n$ and their associated domains are $D_1, D_2, \ldots, D_n$. The set of solutions is the largest subset of $D = D_1 \times D_2 \times \ldots \times D_n$ such that each $n$-tuple in that set satisfies $R$. In (3.1), we are only required to find one member of the solution set.

The generate-and-test algorithm generates $n$-tuples that are elements of $D$, and it tests them to see whether they satisfy $R(x_1, x_2, \ldots, x_n)$. The algorithm searches for an $<x_1, x_2, \ldots, x_n>$ that satisfies $R(x_1, x_2, \ldots, x_n)$. Finding the appropriate $n$-tuple to satisfy $R(x_1, x_2, \ldots, x_n)$ is a blind search through $D$. Since, there is no specific searching strategy, it may take a long time before the correct $n$-tuple is identified (if it exists at all). In fact, the complexity of generate-and-test is exponential since the algorithm does not reduce the search space $D$.

Backtracking algorithms improve on the generate-and-test algorithms. Each variable $x_i$ is assigned a value from its corresponding domain $D_i$. If the resulting $n$-tuple does not satisfy the predicate $R(x_1, x_2, \ldots, x_n)$, then $x_n$ is assigned a new value from $D_n$ and $R(x_1, x_2, \ldots, x_n)$ is evaluated again with this new $n$-tuple. If $x_n$ has been assigned all the values of $D_n$, and $R(x_1, x_2, \ldots, x_n)$ is still false, then $x_{n-1}$ is assigned a new value from $D_{n-1}$, and the resulting $n$-tuple is evalu-
ated in $R(x_1, x_2, \ldots, x_n)$. Following this simple procedure, all the possible $n$-tuples of $D$ are tested in $R(x_1, x_2, \ldots, x_n)$ until the first $< x_1, x_2, \ldots, x_n >$ that satisfies $R(x_1, x_2, \ldots, x_n)$ is found.

Backtracking algorithms are easy to program and require relatively little storage (Bitner and Reingold, 1975). Furthermore, with a single predicate failure it is possible to eliminate an entire subspace of $D$, that is, all the $n$-tuples found in the eliminated subspace are not tested at all.

Unfortunately, backtracking algorithms often suffer from thrashing. When thrashing occurs, some $n$-tuples are needlessly and repeatedly tested. These $n$-tuples differ only in the variables, $x_i$, whose values are irrelevant to the success of the predicate. So-called intelligent backtracking algorithms have been designed to alleviate thrashing in backtracking algorithms (Mackworth, 1985a). These algorithms usually require more storage. Some of these algorithms remember the last failure point so that they can fail back to that point skipping through several intermediate states. Others look ahead to avoid assigning the variables with unacceptable values.

Another way of solving (3.1) is with consistency algorithms (Mackworth, 1977). These algorithms try to correct the thrashing behaviour of backtracking algorithms. They operate on a network graph model that represents variables as nodes, and constraints as arcs, of the graph. Local inconsistent instances are removed by the algorithm when they cannot contribute to any global solution. Removing an inconsistent instance
at one node might cause an inconsistency to occur at some of its neighbouring nodes. The affected neighbouring nodes are subsequently checked for inconsistencies. This process continues until the whole network is consistent. Consistency algorithms are not as simple to program as backtracking algorithms.

Consistency algorithms improve over backtracking by performing a certain amount of local computation so that there is an overall improvement in performance. Unfortunately, there is still no adequate theory that specifies how the nature of the constraints affects the performance of these methods (Mackworth, 1985a).

In order for a knowledge representation to be procedurally adequate, it needs to organize knowledge so that it can be used by a process efficiently. Up to this point in this section we have been showing that the computational task (3.1) can indeed be solved by a machine and we have presented here three algorithms for solving this boolean constraint satisfaction problem. The efficiency of solving the problem depends on the algorithm used. We know that the generate-and-test algorithm is inefficient both in time and space. Backtracking algorithms perform better than generate-and-test algorithms. Of the three algorithms presented here, consistency algorithms are the best.

Even though it is possible for a process to use the knowledge represented by Domain Theory, using this formalism for knowledge representation does not provide control facilities for directing the search. Methods are needed to guide the search and techniques.
CHAPTER 3. DOMAIN THEORY

For instance, should the search be bottom-up, top-down, or hybrid? Domain Theory knowledge representation does not provide any information in this respect. As a result, the search for a solution may be inefficient.

3.4.1 Flexibility

In chapter 2 we mentioned that a knowledge representation that was procedurally adequate also had to display flexibility of use of the available information.

For two domains, Domain Theory defines the computational task as solving the expression

\[ \exists x \exists y ((x \in A) \land (y \in B) \land R(x, y)) \]  

There is flexibility since there are 4 possible specifications for the above formalism:

(i) find \( y \) given \( x \) and \( R \)
(ii) find \( x \) given \( y \) and \( R \)
(iii) find an \( x \) and a \( y \) given \( R \)
(iv) define \( R \) given \( x \)'s and \( y \)'s pairwise such that \( R(x, y) \)

Defining the computational task as in (3.4) shows that Domain Theory allows for bidirectional knowledge flow. For example, let \( x = a \). Then case (i) would be defined as

\[ \exists y ((y \in B) \land R(a, y)) \]

so that data would enter through \( x \) and exit through \( y \). Conversely, let \( y = b \). Then case (ii) would be defined as

\[ \exists x ((x \in A) \land R(x, b)) \]
so that data would enter through $y$ and exit through $x$.

Thus we have shown that Domain Theory does provide flexibility of use of the available knowledge. It permits bidirectional knowledge flow. When there are more than two domains, this same formalism can be extended to achieve multidirectional flow of knowledge.

Ease of reprogramming a system is one of the aspects that count favourably towards a procedurally adequate knowledge representation. In chapter 2 we mentioned the specific example of reprogramming a system so that instead of performing image synthesis, it will perform image analysis. Domain Theory as a knowledge representation scheme allows for ease of reprogramming a system because of the modular way in which knowledge is represented. In fact, domains are like modules in a system. It is possible to modify domain A and leave domain B intact. As long as the relation of representation is redefined, domain B can easily be replaced by domain C. In chapter 5 we describe Depicts that performs both image analysis and synthesis.

In this chapter we have shown that Domain Theory is a descriptively adequate knowledge representation. Procedurally, Domain Theory could incorporate knowledge to guide the search space. Nonetheless, in the other aspects of procedural adequacy, Domain Theory fares well: it has the capability of supporting processes that can use its knowledge framework, it has flexibility of use of the available information, and it provides a modular environment for programming.
Chapter 4

Why Prolog?

This chapter presents the reasons why Prolog was chosen as the programming language for implementing Domain Theory.

Section 1 presents a short overview of the language. Next we show why Prolog is appropriate for implementing Domain Theory: we concentrate in section 2 on the descriptive adequacy and in section 3 on the procedural adequacy.

Subsection 4.3.1 focuses on the flexibility of knowledge flow offered by Prolog. We have placed this subsection under section 3 since flexibility of knowledge flow is one of the considerations in the evaluation of a knowledge representation for procedural adequacy. However, this subsection is rather extensive because we found reversibility in Prolog to be an interesting feature.

Section 5 points out some of the drawbacks of Prolog.
CHAPTER 4. WHY PROLOG?

4.1 Overview of Prolog

Prolog is a programming language that implements the computational paradigm of logic programming and is based on the Horn clause form of logic (Sergot et al, 1986). One of the attractive features of Prolog is that it is based on logic, a field that has been studied extensively and is well understood. Here we give a brief overview of Prolog using the notation of Clocksin and Mellish (1981) and the definitions of Lloyd (1984).

Definition: A constant is a string of characters that names a particular object. A variable is a string of characters that stands for an object. A variable is instantiated when there is an object that the variable stands for. Conversely, a variable is uninstantiated when what the variable stands for is not yet known. Variables begin with a capital letter or an underscore (_). This is how we distinguish constants from variables.

Definition: A term is defined as follows:

(i) a constant is a term
(ii) a variable is a term
(iii) if $f$ is an $n$-ary function and $t_1, t_2, \ldots, t_n$ are terms, then $f(t_1, t_2, \ldots, t_n)$ is a term.

Definition: An $n$-ary predicate is represented as $p(t_1, t_2, \ldots, t_n)$ where each $t_i$ is a term.

Definition: Given that $p, q_1, q_2, \ldots, q_n$ are predicates, a program clause (also called a definite Horn clause) is of the form $p : -q_1, q_2, \ldots, q_n$, meaning that $p$ follows
iff all $q$'s hold. This clausal notation is a shorthand for the formula

$$\forall x_1 \forall x_2 \ldots \forall x_s (q_1 \land q_2 \land \ldots \land q_n \rightarrow p)$$

where the $x_1, x_2, \ldots, x_s$ are all the variables occurring in $q_1, q_2, \ldots, q_n$. $p$ is called the head of the clause and $q_1, q_2, \ldots, q_n$ is called the body of the clause. A unit clause is of the form $p$, that is, it is a program clause with an empty body.

**Definition**: A goal clause (also called a Horn clause query) is of the form, $:- q_1, q_2, \ldots, q_n$. This clausal notation is a shorthand for the formula,

$$\forall y_1 \forall y_2 \ldots \forall y_l (\neg q_1 \lor \ldots \lor \neg q_n)$$

or equivalently

$$\neg (\exists y_1 \exists y_2 \ldots \exists y_l (q_1 \land q_2 \land \ldots \land q_n))$$

where the $y_1, y_2, \ldots, y_l$ are all the variables occurring in $q_1, q_2, \ldots, q_n$.

**Definition**: A Horn clause is either a program clause or a goal clause. From this point on, the unqualified term clause will denote Horn clause. A Prolog program is a finite set of Horn clauses. In a Prolog program, unit clauses are viewed as facts whereas program clauses are considered as rules. Goal clauses are used for querying or invoking the rules and the facts.

**Definition**: Unification is the process of matching. Unification of two predicates occurs when the predicate names are the same and their arguments refer to the same terms. If in one of the predicates there is still an uninstantiated variable, then the unification process binds this variable to the value of the corresponding argument
in the other predicate. Unification satisfies the main purpose of logic programming systems which is to compute bindings.

4.2 Descriptive Adequacy

In chapter 3 we presented Domain Theory for knowledge representation: the knowledge of the world is divided into domains with mapping relations connecting the objects in the domains. We also showed that Domain Theory led to a knowledge representation that was descriptively adequate. In this section we show how Prolog produces proper representations for the objects in the domains and their mapping relations, and thus we can claim that Domain Theory in Prolog is descriptively adequate.

Given the universal relation of representation, \( U(object_1, object_2, \ldots, object_n) \), each \( object_i \) is defined as in subsection 3.2.5:

\[
\begin{align*}
< object > & ::= < atomic \ object > | < composite \ object > \\
< composite \ object > & ::= < relation(object_1, object_2, \ldots, object_n) > \quad \text{for } n > 0 \\
< atomic \ object > & ::= < relation > \quad (0\text{-place relation})
\end{align*}
\]

\( u(object_1, object_2, \ldots, object_n) \) can be used to describe the universal relation of representation in Prolog, where \( u \) is the name of a Prolog predicate and each \( object \) is defined as:
CHAPTER 4. WHY PROLOG?

Note the close similarity between the representation of objects and relations in Domain Theory and in Prolog. Relations are represented as predicates in Prolog. When the relation is n-ary, an n-ary predicate is used. When the relation is 0-place, a Prolog constant or variable is used. Appendix A gives an example of a composite object and shows how it can be analyzed according to the rules given above.

The arguments of a Prolog predicate are always terms; Prolog does not allow for predicates to have predicates as arguments. Therefore, strictly speaking, if a composite object has objectₙ as its argument, and if objectₙ is also composite, then objectₙ will be represented as a function term (as in function(object₁, object₂, ..., objectₙ)) and not as a predicate (as in predicate(object₁, object₂, ..., objectₙ) as defined above). This is not a desirable distinction which we wish to make. We blur the difference between objects that are function symbols and those that are predicate symbols. We feel that this is an implementation issue that can be easily overcome. Whereas the arguments of a Prolog predicate cannot be predicates, they can be functions or lists ¹. We can then use the evaluable predicate call or the predicate univ to apply them to the functions and lists so that the functions and lists will be treated as predicates. call treats its

¹Lists are also functions but it is useful to distinguish here between functions and lists
argument as a goal. The `univ` predicate takes a list and assigns the first element of the list as the predicate name, and the rest of the elements of the list as arguments to the predicate (Clocksin and Mellish, 1981).

Using Prolog for Domain Theory is descriptively adequate. The syntax for relations and Prolog predicates is very similar. In fact, there is a close correspondence between Domain Theory and logic programming as shown here and as discussed in the next section.

### 4.3 Procedural Adequacy

In chapter 3 we showed that many of the criteria of a procedurally adequate knowledge representation were fulfilled by Domain Theory. We defined the computational task for Domain Theory as determining the truth value of the formula

\[ \exists x_1 \exists x_2 \ldots \exists x_n ((x_1 \in D_1) \land (x_2 \in D_2) \land \ldots \land (x_n \in D_n) \land R(x_1, x_2, \ldots, x_n)) \quad (3.1) \]

This equation can be reformulated to

\[ \exists x_1 \exists x_2 \ldots \exists x_n (d_1 \land d_2 \land \ldots \land d_n \land d_{n+1}) \quad (4.1) \]

where

\[ d_i \text{ is } (x_i \in D_i) \]
\[ d_{n+1} \text{ is } R(x_1, x_2, \ldots, x_n) \]

The negation of formula (4.1) is exactly equivalent to a goal clause in Prolog. Therefore the computational task formula (4.1) is expressible in Prolog in the form

\[ : - d_1, d_2, \ldots, d_n, d_{n+1}. \quad (4.2) \]
The bindings to the variables of this goal will be computed during the unification process of Prolog (Lloyd, 1984).

Prolog uses a *depth-first* search strategy when it investigates alternate clauses in trying to satisfy a goal. The mechanism used in guiding this depth first search is *backtracking*. Prolog satisfies each subgoal, $d_i$, in sequential order. If any of the subgoals, $d_i$, fails, then Prolog backtracks to $d_{i-1}$, attempts to reestablish the satisfiability of $d_{i-1}$, and then tries to satisfy $d_i$ again. In section 3.4 we described backtracking as one of the possible algorithms for solving the computational task. We see that Prolog provides us with automatic backtracking; therefore, specifying (4.1) in Prolog not only gives us the exact formulation of the problem, but also provides us with the control mechanism needed to solve the problem.

We said that a procedurally adequate knowledge representation provided a framework that could be used efficiently in time and space. In this section, we have shown that representing the computational task in Prolog provides a framework that can be used by a machine since the formula may be represented as a Prolog goal. Furthermore, the formula is solved by the Prolog backtracking algorithm.

Up to this point, we have ignored the fact that a procedurally adequate knowledge representation supports *efficient* processes. We admit that Prolog's search strategies are not optimal; backtracking is an inefficient algorithm. Thus, we are led to conclude that Prolog does lend itself to representing the computational task of Domain Theory,
however, there is much room for improvement in efficiency.

4.3.1 Flexibility

In most programming languages each parameter is either for input or for output. This distinction is a precompiled restriction that may not be changed during execution time. Data enters through the input parameters (or information sources) and exits through the output parameters (information sinks). In this case the direction of the knowledge flow is fixed.

Prolog is different in this respect. The decision as to whether a parameter is going to be used for input or for output is delayed until the time when the predicate is called. If a parameter is uninstantiated at the time the predicate is called, the parameter is used for output; otherwise, it is used for input. This Prolog feature is useful for implementing Domain Theory so that bidirectional (or multidirectional) knowledge flow can be achieved.

Two pertinent questions arise:

Given a predicate defining $R(x, y)$,
- Can Prolog always find $y$ given $x$?
- Can Prolog always find $x$ given $y$?

Under Domain Theory the answer to both questions is "yes" because the relation $R$ is well defined both as a mathematical relation and as a Prolog predicate. It seems, therefore, that by using Prolog we are getting knowledge inversion for free.
Before analyzing further knowledge inversion in Prolog, we see how Dijkstra, Eppstein, and Shoham approach the issue. Dijkstra (1982) gives a solution to the vector inversion problem which he manually worked out. However, he admits that he was able to solve this problem because his program is deterministic without information loss. He says that the inversion of non-deterministic programs is left as an open question.

Eppstein (1985) treats the program inversion problem heuristically. He gives a procedure for producing the inverse of simple LISP functions. It is not clear, however, whether or not his program can be used to reverse complicated functions. Eppstein does not address the issue of bidirectional flow of knowledge. He is merely concerned with producing the inverse of functions.

Shoham (1984), in his paper on “Knowledge Inversion”, gives an interesting way of computing reversibly in Prolog: given a goal made out of a conjunction, solve the conjuncts in reverse order. Shoham’s solution does not give true bidirectional knowledge flow because the user needs to invoke specifically a predicate called \texttt{invgoal} whenever he wants the inverse of the goal solved. The advantage to having a separate predicate \texttt{invgoal} is that the reverse of extra-logical Prolog features can be computed. For instance, when \texttt{invgoal} encounters \texttt{assert} it substitutes this goal by \texttt{retract}. Another problem with Shoham’s approach is that altering the order in which the subgoals are solved may not always work. Shoham mentions functions that cannot be reversed by his algorithm.
CHAPTER 4. WHY PROLOG?

As we worked on the subject of inversion in Prolog, we came to the conclusion that, for a general predicate \( p(x, y) \), it is not always possible to obtain \( x \) for a given \( y \), or vice versa.

For instance,

\[
\begin{align*}
p(X,Y) & : - q(X, [], Y). \\
q([], Y, Y). \\
q([X\text{\_head} \mid X\text{\_tail}], \text{Temp}, Y) : - \\
& \text{one}(X\text{\_head}, Y\text{\_head}), \\
& q(X\text{\_tail}, [Y\text{\_head} \mid \text{Temp}], Y). \\
\text{one}(p(W), g(V)) & : - \text{one}(W, V). \\
\text{one}(h(W), l(W)).
\end{align*}
\]

will loop infinitely if \( Y \) is bound (and not bound to \( [] \)) while \( X \) is unbound. Therefore, a Prolog programmer needs to be careful in writing his predicates so that the predicates can be solved both forwards and backwards. Furthermore, he needs to avoid Prolog's extra-logical feature (such as \textit{the cut}, \textit{assert} and \textit{retract} (Clocksin and Mellish, 1981)) since these are not usually reversible.

It is possible that a predicate may be solved bidirectionally; nonetheless, we found that Prolog seemed to favour one direction. In other words, most predicates were solved efficiently when executing in one direction but were very inefficient when executed in the other direction.

Even though Prolog cannot solve \textit{all} its predicates bidirectionally, we found that Prolog’s neutrality with respect to information sources/sinks is helpful in implementing Domain Theory. Under Domain Theory, knowledge is represented as relations. Since
CHAPTER 4. WHY PROLOG?

our relations are well defined, we did not encounter the above-mentioned problems of
reversibility with Prolog predicates. If we can define the relation in Domain Theory,
then we can represent it as a Prolog predicate and solve this predicate bidirectionally.

Descriptively, Prolog predicates are appropriate for representing the objects of Do­
main Theory. The computational task of Domain Theory is well described as a Prolog
goal. The task is conveniently solved by Prolog’s backtracking mechanism. Further­
more, Prolog allows all parameters to be either information sinks or sources, thus
offering bidirectional knowledge flow.

4.4 Problems in Prolog

Prolog has proven to be appropriate for implementing Domain Theory; nevertheless,
Prolog is far from being the ideal programming language. Here we present some of our
experiences with Prolog which, as it turns out, will counterbalance the praise given to
the language in the previous sections. On the other hand, we do not want to dwell on
the problems found in Prolog as this is beyond the scope of this thesis.

Cuadrado and Cuadrado (1985) recommend Prolog highly; nevertheless, they write
that in developing an application, they are “forced to write the whole thing as one
giant, flat program”. Prolog has software engineering problems that become more
obvious in large programs. A few versions of Prolog do allow modules but often these
modules are implemented into Prolog by adding extra-logical “features”. Bowen and
Kowalski (1982) have investigated this matter and have suggested a way to modularize Prolog while keeping its foundation on logic.

Two problems in Prolog that result from favouring procedural adequacy in the descriptive/procedural adequacy tradeoff (as described in chapter 2) are the implementation of negation as negation by failure and the omission of the occur check in the unification algorithm.

*Negation by failure* is not true negation as in logic so descriptively this choice of knowledge representation is somewhat lacking; nonetheless, procedurally, negation by failure is an efficient way of providing negation to a system that cannot deduce negative information. Since the Horn clause subset of logic does not have sufficient expressive power, Prolog systems allow negative literals in the bodies of clauses. These negative literals conveniently implement negation as negation by failure (Lloyd, 1984). Therefore, implementing Domain Theory in Prolog implies that negation is implemented as negation by failure.

Domain Theory uses the *closed world assumption* so that information not explicitly present in the database is taken to be false. For instance, if we cannot find \( x \) in a domain in order to use it as an argument to a relation, then we just fail and assume that \( x \) cannot be mapped into \( y \). The closed world assumption and negation by failure are not necessarily equivalent; however, they are compatible when the database is Horn and definite (Shepherdson, 1984). The implementation of Domain Theory in
CHAPTER 4. WHY PROLOG?

Prolog is Horn and definite which fulfills the requirements for compatibility. Since the closed world assumption is a more powerful rule than the negation by failure rule, a research issue is to extend the negation by failure rule to resemble more the closed world assumption (Lloyd, 1984).

The other well known deficiency in Prolog is the omission of the occur check in the unification algorithm. Again, this decision favours procedural adequacy, not descriptive adequacy. The occur check prevents a match of a term against a subterm of itself. It is easy to include the occur check in a unification algorithm. However, the occur check reduces the efficiency of the unification algorithm and increases its computational complexity. Most Prolog implementations omit the check altogether producing descriptive inadequacy since matching a term against its own subterm may cause an infinite loop.

In our experience with C-Prolog version 1.5 (Pereira, 1984), Prolog was slow and demanding in memory. Nevertheless, we expect Prolog's performance to improve within the next few years as Prolog compilers become more efficient and as computing power increases and hardware costs decrease. This will be especially true when the fifth-generation of computers becomes available (Feigenbaum and McCorduck, 1983).
Chapter 5

Description of Implementation

This chapter describes Depicts ¹ which is a working system that implements Domain Theory for knowledge representation. Depicts is written in Prolog and it is applied specifically to the image (picture) domain and the scene (description) domain. Depicts is neutral in respect to the synthesis/analysis question.

Section 1 defines the image and scene domains. Their relation of representation is called the depiction relation (Mackworth, 1985b). Section 2 gives a specific example that is used to explain the Prolog data structures in Depicts. Section 3 mentions the tools used in the development of Depicts. Section 4 describes the environment created by Depicts. Section 5 suggests possible applications for Depicts.

Up to this point we have talked about a theory for knowledge representation. In this chapter we show that it is indeed possible to implement this theory. Furthermore, we show that our implementation Depicts can actually be used as a practical tool.

¹depict is defined in the Merriam-Webster (1974) dictionary as a verb (i): to represent by a picture (ii): to describe in words
The value in having a working system is that of assigning semantics to a theory that would otherwise be purely syntactical. Syntactically knowledge is represented as the expression "line(point(a),point(b))", but semantically this expression represents the physical line: "a —— b".

5.1 Image Domain and Scene Domain

The image domain (or picture domain according to Clowes (1971)) deals with pictorial information. The image domain includes lines, circles, regions, and image characteristics up to the level of Marr’s primal sketch (Havens and Mackworth, 1983). Note that in this thesis we do not consider the image domain to be consisting of 2-dimensional continuous signals. We assume that the image domain is already a graphical representation.

Usually when we see an image, we are not interested in how the lines are connected; we are interested in what the line connections represent. Do the line connections represent an intersection of two roads? Do they represent an edge of a table, or do they represent the roof of a house? Images only carry meaning because they depict scenes (Mackworth, 1983b). We analyze the image domain because we want to learn about the world being imaged, that is, we want to find out what the scene domain is.

From chapters 3 and 4 we know that one-to-one depiction relations between domains have much nicer properties than many-to-one relations. When the relations are one-
to-one, the solution to the computational task is unique and there is also bidirectional flow of knowledge.

Unfortunately the depiction relations from the scene domain to the image domain are usually many-to-one. The scene domain contains knowledge of a three-dimensional world. The image domain contains knowledge of a two-dimensional world, namely, the image. A mapping of information from the scene domain to the image domain implies going from a three-dimensional world to a two-dimensional image. One "dimension" is lost in this process; in fact, the lost information can never be recovered completely. As a result, given an image, it is not possible to determine uniquely the scene that corresponds to the particular image. An image might correspond to an infinite number of scenes.

In Depicts we want bidirectional flow of knowledge between the scene and image domains, which implies that the depiction relation between the domains needs to be one-to-one. This can be accomplished by restricting the scene domain to two-dimensional worlds with corresponding images as diagrams and sketches. Consequently, there can be flexibility of use of the available knowledge (regardless of whether it comes from the scene or the image domain) without any information loss.

Sketches and diagrams convey information in an effective and compact way. They are used widely both for human-human and human-machine communication (Browse, 1982) as in engineering drawings, architectural plans, maps, and the like. In fact, line
drawings have been used by humans for a long time. They have even been found in primitive caves dated thousands of years B.C.. All of these line drawings are meaningful because there is a graphical convention associated with the various parts of the drawings.

The domains used by Depicts are the scene domain, and the image domain consisting of line drawings. In a vision system, data flows from the image domain to the scene domain. In a graphics system, data flows from the scene domain to the image domain. Depicts allows bidirectional flow of knowledge between the image and the scene domain depending on what the information sources are.

5.2 Data Structures

In this section we present a specific example of how knowledge is represented in Depicts. The problem is one of illustrating a certain classification. For instance, we would like a graphical representation of a dichotomous identification key for the British Columbia Tree Species.

When we talk about a classification, we talk about classes, and about classes being subclasses of other classes. The scene domain will then have objects that are classes, and the relation among these objects is that of an object being a subclass of another object.

Suppose that we want to illustrate this classification with n-ary trees. The picture
will then have lines connecting two points. In the image domain the objects are points, and the relation among them is a line joining two endpoints.

In order to map subclass into line in this manner, we need to establish a closer relationship between these two relations. Class\( Y \) is a subclass of class\( X \) if in the diagram, node\( X \) is the father of node\( Y \). We also want to say that node\( X \) is the father of node\( Y \) if point\( Y \) is joined by a line to point\( X \). So, we create an intermediate domain that is in between the scene domain and the image domain. The composite objects for each of the three domains are

Scene Domain : subclass(class\( Y \),class\( X \)).

Intermediate Domain : father(node\( X \),node\( Y \)).

Image Domain : line(point\( Y \),point\( X \)).

These composite objects are arguments to the depiction relation. Let us call the depiction relation depicts. The data structure for depicts between the scene domain and intermediate domain is

\[
\text{depicts(subclass(class}(Y),\text{class}(X)),\text{father}(\text{node}(X),\text{node}(Y))).
\] (5.1)

and between the intermediate domain and the image domain

\[
\text{depicts(father}(\text{node}(X),\text{node}(Y)),\text{line}(\text{point}(Y),\text{point}(X))).
\]

As an alternative to illustrating the classification with a tree, we can use a Venn diagram; the image domain will then consist of circles that are contained within other
circles. Here are the composite objects for the same scene domain and the new image domain of Venn diagrams

Scene Domain: \( \text{subclass}(\text{class}(Y), \text{class}(X)) \).

Intermediate Domain: \( \text{subset}(\text{set}(X), \text{set}(Y)) \).

Image Domain: \( \text{contains}(\text{circle}(Y), \text{circle}(X)) \).

The Depicts interpreter uses this data structure to perform the correct mappings across domains. Appendix A analyzes clause (5.1) according to the Backus Naur Form rules given in section 4.2. Figure 5.1 illustrates the depiction relation data structure between the scene domain of subclasses, and the image domains of contains and of lines.

5.3 Tools

Depicts has been tested on the SUN-2/5 Workstation\(^2\) with 2 Megabytes of memory under Unix 4.2 BSD\(^3\). The output modules were also tested on a VAX-11/780\(^4\) under Unix 4.2 BSD.

Depicts has been implemented as device independent as possible. When Depicts is executed on a machine other than a SUN Workstation (or when the user specifies hardcopy when Depicts is being executed on a SUN Workstation), the output which

---

2SUN Workstation is a trademark of Sun Microsystems Incorporated

3Unix is a trademark of AT & T Bell Laboratories

4VAX is a trademark of Digital Equipment Corporation
Figure 5.1: Schematic Diagram of Data Structure
is in the form of plotting commands is directed to a file. (The plotting commands are library commands at the Laboratory for Computational Vision, University of British Columbia. They are a superset of the standard Unix plotting commands.) This file is then used to produce graphs which can be drawn on a Telidon terminal\textsuperscript{5}, a Raster Technologies Model One/25\textsuperscript{6}, a LaserWriter\textsuperscript{7}, or an Imprint-10 laser printer\textsuperscript{8}.

\textbf{Depicts} runs on a SUN workstation using GProlog (Brachman, 1985) which is an enhanced version of C-Prolog (Pereira, 1984) that incorporates most of the graphics functions found in the SunCore library. The SunCore library consists of SUN Microsystem's (1985) implementation of the Core standard for graphics programming (cf. subsection 2.3.3).

The GProlog graphics functions are called as Prolog predicates:

\begin{verbatim}
plgraphics(< SunCore function name >)
or
plgraphics(< SunCore function name > (< arg_1 >, ..., < arg_n >))
\end{verbatim}

or, more conveniently, by prefixing the function name with a sharp sign, as in

\begin{verbatim}
# < SunCore function name >
or
# < SunCore function name > (< arg_1 >, ..., < arg_n >)
\end{verbatim}

\textsuperscript{5}Telidon is a trademark of Microtel
\textsuperscript{6}Model One/25 is a trademark of Raster Technologies Incorporated
\textsuperscript{7}LaserWriter is a trademark of Apple Computer Incorporated
\textsuperscript{8}Imprint is a trademark of Imagen Corporation
5.4 Environment

In this section we will describe the environment created when Depicts is run. Figure 5.2 gives a system diagram of Depicts. The major two components are the editor and the interpreter. These will be discussed in the following two subsections.

5.4.1 The Editor

The editor supports the communication between Depicts and the user. Furthermore, the editor makes the user interface between the machine and the person "user friendly"
and as a result, the end user of Depicts need not be an experienced programmer.

The three major subdivisions of the editor are for input, output, and command processing. The input and output divisions are themselves segmented into the various domains. Sometimes input is in the form of words (as in the scene domain) and at other times it is in graphical form (as in the image domain). Similarly the output could be an English-like description (as in the scene domain) or it could be a graphical diagram (as in the image domain). As implied by its name, the command processor processes commands redirecting the flow of control. If there is a command that needs more data, the command processor will seek to obtain more information from the user. Alternatively, the command processor might simply need to call the interpreter to perform a necessary action.

In short, the editor takes the user input, (be it graphical or in words) and prepares the correct data structure for the interpreter to use. It then invokes the interpreter. Finally, it receives the interpreter’s answers (graphical or in words) and displays them on a screen or prints them to produce a hardcopy.

Depicts has been tested with the scene domain classes and the image domains tree and Venn diagram. These domains were explained in detail in section 5.2 and the sample session in appendix C uses these domains. We need to emphasize that Depicts is not limited to classes, trees and Venn diagrams. At the present time Depicts is set up for these three domains and so, instead of describing the editor in general terms,
we will give a brief description of the editor using these domains.

Figure 5.3 illustrates how the screen is set up when Depicts is executed on a SUN Workstation \(^9\). The lower part of the screen is used for graphics while the upper part is used for conversation between the user and Depicts.

The editor presents the choice of commands in the form of menus. The user selects a command by using a pointing device and as a confirmation of his selection, the chosen command is flashed on the screen. (The pointing device we use is the mouse that is attached to the SUN Workstation).

---

\(^9\)SUN Workstation is a trademark of Sun Microsystems Incorporated
All the menus of commands can be grouped into a hierarchy as illustrated in Figure 5.4. The top menu of the hierarchy is the first menu to appear on the screen. After a command is selected, a second menu may appear. This latter menu belongs to the level below in the hierarchy. To go up one level, the user needs to quit the current menu.

When the user selects classes in the first menu presented to him, he indicates that he wants to converse with Depicts in a non-graphical manner, that is, he wants
to interact at the scene domain level. He can update the scene domain knowledge base since Depicts has the ability to query. Alternatively, the user may find out what is stored in the scene domain knowledge base because Depicts can describe in words the contents of a knowledge base.

By pointing at classes $\leftrightarrow$ tree, the user is indicating that he wants to move from the classes domain to the tree domain. This command may look simple to the user but it is significant; it causes the command processor to invoke the interpreter so that a mapping between the classes and the tree domains is performed.

The user may enter the image domain tree to interact with Depicts graphically. Depicts makes it simple for the user to draw a tree by providing him with the command line that is used for drawing lines and the command text for labelling the nodes of the tree. The user may redraw the tree at anytime or he can ask for a pretty layout of the tree. The algorithm used for the layout of the tree is an extended version of Vaucher's "Pretty-Printing of Trees" (1980).

The command window is used for choosing a new view of the diagram. The user can reduce or enlarge the diagram. There is also freedom to move down, move up, move right, or move left in the present screen. Another option provided by Depicts is to create a hardcopy of the diagram.

If the user chooses venn diagram as his image domain, the graphical primitive available is circle with the corresponding text command to label the circles.
The user can manipulate the diagram at will, displaying various parts with the use of the `window` command. `hardcopy` will redirect the output to a printer.

### 5.4.2 The Interpreter

The interpreter receives the data structure as described in section 4.2 and executes commands such as updating the knowledge base or mapping objects across domains. After solving the query posed by the editor, the interpreter returns with the solutions and waits for further instructions.

In section 3.4 we defined the computational task of Domain Theory and in section 4.3 we showed how a Prolog goal was appropriate for describing the computational task. It is at the interpreter level where Depicts solves the computational task. The interpreter takes the data structures produced by the editor and uses them to satisfy the Prolog goal that defines the computational task.

For instance, take the specific task of mapping objects between the `fathers` domain and the `tree` domain. The Prolog predicate `depicts_fathers_lines` defines the mapping relation between the two domains. The interpreter takes the query and data structure from the editor, and it formulates them into a Prolog goal. Appendix B shows how this goal is satisfied emphasizing the bidirectional flow of knowledge. When the Prolog goal is satisfied, the computational task is solved and the mapping across domains is done properly.
The interpreter in Depicts is independent of the types of domains used. Naturally, the various depiction relations need to be defined according to the objects found in the different domains.

5.5 Applications

In this section, we will discuss some of the possible uses of Depicts in computer graphics, expert systems and databases.

5.5.1 Computer Graphics

There are two practical uses of Depicts in computer graphics: domain independence and image transmission.

In subsection 2.3.3 we discussed the need for device independent graphics packages. With Depicts we go one step beyond device independence into image domain independence. Image domain independence implies that the image domain can be changed while leaving the scene domain intact. Depicts achieves image domain independence, as shown in section 5.2, by the two image domains tree and Venn diagram and the one corresponding scene domain classes.

The idea of image domain independence seems simple and logical. It is a natural consequence of Depicts because knowledge is represented using Domain Theory. Nonetheless, most graphics systems are not like this. Knowledge is usually embedded in an obscure, unportable program and as a result, image domain independence is
difficult to achieve. In contrast, Depicts can produce “modular, portable, versatile and intelligent graphics systems with scene domain independence as well” (Mackworth, 1983b). Scene domain independence means that the image domain is not affected when the scene domain is changed. We will illustrate scene domain independence by extending the previous example of section 5.2. We will keep the same image domain tree, but will switch from the scene domain classes to the scene domain algebraic expression.

Trees are used as internal representations of source expressions in compiler writing. A compiler takes an expression, builds a parse tree, and then uses this internal representation of a tree to generate code. Suppose that the user wants a graphical representation of the parse tree. In the image domain he would still deal with lines and points as before, but the scene domain would consist of operators and operands. Let us call this scene domain the algebraic expression domain.

In Depicts, there are Prolog predicates that define the objects in a domain and there are Prolog predicates that define the relations between the objects in the domain. These predicates produce the data structures of section 5.2. Depicts provides a predicate that performs the depiction mapping by taking domains and relations as its arguments.

To add the algebraic expression domain to Depicts, we define the operands, the relation operator, and the depiction relations. There is no need to modify the image domain. As we change the scene domain from classes to algebraic expression, we do
not alter the *lines* and *points* of the image domain *tree*.

It is necessary to formalize the relations and objects into first order logic in order to add a new domain. Consequently, the defining Prolog predicates are constructed and the depiction predicate of *Depicts* is evoked. Therefore, in adding a new domain, most of the code of *Depicts* remains the same and just a few, new, defining Prolog predicates are created. The editor also needs to be updated so that the menus will reflect the addition of the new domain.

The Prolog predicates that define the objects can be quite involved. This is especially true for image domains. There is some preprocessing from the graphical representations into the image domain objects and relations. Conversely, layout functions need to be considered when transforming image domain objects and relations into their graphical representation. These layout functions are defined explicitly in *Depicts*.

In *Depicts* we implement a form of high level graphics primitives. We define primitives like *circle* and *line* where GProlog provides `#move_abs.2(\(X,Y\))` and GKS-Prolog `g.pl< list of points>` . Some work has been done on high level graphics such as the graphical language LIG (Ross, 1984). Appendix C shows a database used by *Depicts* that uses *circle* and *line*. Using these higher level primitives allows for the true portability of *Depicts*. Obviously, some lower level functions are needed to take the primitives like *circle* and call the machine dependent graphics routines.

A system like *Depicts* could also be useful in image transmission. There are
systems that have been successful in transmitting images by encoding them in videotext schemes like Telidon. However, this process could be improved by sending high-level scene descriptions instead of transmitting the whole image. This would reduce the volume of the data transmitted. Sending the scene domain description instead of the whole image can be seen as a form of data compression. We would expect the sending and the receiving systems to have Depicts with the same definitions of the image domain, the scene domain, and the depiction relations.

5.5.2 Expert systems

Expert systems are programs that perform functions resembling those performed by human experts in their various professional fields. Some expert systems have been useful in calculating computer configurations (e.g. Digital Equipment Corporation’s XCON), in exploring for oil and minerals (e.g. SRI International’s Prospector), in trouble-shooting and repair (e.g. General Electric Corporation’s CATS-1), in identifying chemical compounds (e.g. Stanford’s DENDRAL) and in medical diagnosis (e.g. Stanford’s MYCIN) (Feigenbaum and McCorduck, 1983).

The major bottleneck in developing an expert system are acquiring knowledge from the human expert, and encoding this knowledge into the system. There is often a knowledge engineer whose task is to find out how a human expert solves problems. The knowledge engineer interviews the expert, he learns about the expert’s reasoning
procedure, and then he enters this information into the system in the form of rules and facts. The transfer of knowledge from the human expert to the expert system through the knowledge engineer is a cumbersome procedure. It would be nice if the human expert could communicate directly with the expert system. However, this is not usually the case for the simple reason that most expert systems require for the knowledge be entered in a manner that is awkward for people who are not experienced programmers.

Computer graphics is a convenient tool for displaying knowledge. Depicts can be used as an intelligent graphics system that interfaces between the human expert and the expert system. In Depicts, the image domain would consist of graphical representations, the description domain would be the knowledge base rules and the depiction relation would map to/from the image domain from/to the description domain. The human expert would express his knowledge by drawing directly on the CRT screen, and this knowledge would be mapped automatically by Depicts into the description domain of knowledge base rules. In this way the human expert would be interacting directly with the expert system without having to know much about computer programming.

The final end user would not need to be aware of how data had been entered originally by the expert (in this case it was graphically). In fact, he could converse with the expert system exclusively at the description domain level. Alternatively, he
could display the knowledge in a graphical manner.

Some expert systems do use graphics (Kinnucan, 1984) but they are usually used for displaying the knowledge base, and not for input. Furthermore, these graphical methods are usually geared to one application, making it difficult to generalize to other fields.

For instance, there is an expert system that displays the hierarchical structure of its database in a tree-like manner. However, it does not let the user define how he wants the knowledge to be represented. The user can only use the fixed graphical representations that come with the system. In contrast, Depicts offers flexibility in that the user may define an image domain that will suit his needs.

5.5.3 Databases

Representing knowledge as defined by Domain Theory can be viewed as a generalization of the relational database. A relational database is compromised of time-varying, normalized relations (Date, 1981).

As mentioned before, it is often practical to represent knowledge schematically; however, it is quite unlikely that the data has been stored in a way that lends itself to a graphical representation. Depicts can be used as a link between the data stored and a visual representation of the database.

Depicts does not lose any information as data flows from one domain to another.
As a result, a query can be made in one domain, Depicts checks the answer in another domain, and then it comes back to the user with the mapped answer.

For example, let the scene domain consist of classes and subclasses as used in the example of section 5.2 and appendix C. Let the image domain be a tree diagram and let the intermediate domain specify father relations. Now suppose that the user wants to find out the superclass of hardwoods.

$\Downarrow$ represents the depiction relation performed by Depicts.

\[
\begin{align*}
\text{subclass}(\text{class(hardwoods)},X). \\
\Downarrow \\
\text{father}(\text{node(X)},\text{node(hardwoods)}). \\
\Downarrow \\
\text{line}(\text{point(X)},\text{point(hardwoods)}).
\end{align*}
\]

after consulting the image domain database:

\[
\begin{align*}
\text{line}(\text{point(trees)},\text{point(hardwoods)}). \\
\Downarrow \\
\text{father}(\text{node(trees)},\text{node(hardwoods)}). \\
\Downarrow \\
\text{subclass}(\text{class(hardwoods)},X).
\end{align*}
\]

where $X = \text{class(trees)}$

Making a query in the scene domain, finding the answer in the image domain (which is essentially looking at the picture), and returning to the scene domain with the answer form a specially interesting application of Depicts. This progression of mappings, scene-image-scene, resembles the way a human sees and understands a diagram.
There is potential use for systems that use computer graphics to illustrate words.

There is potential for systems that use words to describe pictures. Depicts is a system that truly depicts: it illustrates words and describes pictures.
Chapter 6

Evaluation and Possible Future Extensions

At the onset of this research the problem was defined as finding a knowledge representation that would capture the knowledge in the world and that could be efficiently in computation. A good knowledge representation would have to be descriptive and procedurally adequate.

6.1 Evaluation

We presented Domain Theory as a knowledge representation scheme. We have evaluated it and found it to be descriptively adequate. All the knowledge in the world can be represented using this formalism of domains and their relations. Furthermore, Domain Theory produced a knowledge representation that was finite yet able to characterize and arbitrary number of inputs.

Domain Theory fulfilled some of the aspects of a procedurally adequate knowledge
representation but certainly not all of them. It was shown to be flexible with respect to the availability of information. It provided a computational task that could be solved indeed by an algorithm. However, we admit that obtaining the solution might be an inefficient process. Unfortunately the knowledge representation of Domain Theory did not provide any control facilities to direct the search.

We mentioned earlier that there is a tradeoff between knowledge representations that are descriptively adequate and procedurally adequate. No representation can be optimal in both respects and using Domain Theory for knowledge representation is no exception. Domain Theory favours descriptively adequacy. We can capture all the knowledge in a situation of the world, however, it is not entirely clear how efficiently it can be used in a computational procedure. We suggest that there be further work in this area of improving the algorithms used for solving the computational task of Domain Theory.

Prolog was found to be well suited for implementing Domain Theory. Firstly, we defined a close relationship between Prolog predicates and Domain Theory objects. Secondly, we showed that Prolog's backtracking control mechanism was able to solve the computational task of Domain Theory. Lastly, we demonstrated that Prolog offered the necessary flexibility that would allow for bidirectional knowledge flow.

Even though computer graphics and computational vision both deal with image-based systems, work overlapping both fields has been limited (Mackworth, 1983b).
However, we feel that this can now be changed. We need to represent knowledge in a
descriptive and procedurally adequate way.

Our system, Depicts, uses Domain Theory for knowledge representation. Depicts is a departure from the typical approaches taken so far whereby procedural adequacy was favoured above descriptive adequacy. In Depicts knowledge is used flexibly and as a result, we can apply Depicts to image analysis, image synthesis, image transmission, databases and human-machine communication.

6.2 Possible Future Extension to Depicts

Depicts does not have an elaborate natural language front end for the user to modify the scene domain clauses directly. Nonetheless, it is relatively simple to add such a natural language capability to Depicts given that knowledge has been partitioned into separate domains. We do not need to go outside the Prolog formalism when including a natural language user interface. In fact, logic programming is convenient for translation from natural language (Kowalski, 1982).

In a similar manner, one could extend Depicts to manipulate 2-dimensional continuous signals. At the time of this writing Depicts provides the user with a graphical editor, as described in subsection 5.4.1., so that he can draw pictures interactively on the CRT display. One could add a program to Depicts that analyzed hardcopies. This would be a more complex image analysis than currently available in Depicts. What
is nice about this idea of domains in Depicts is that, even if we changed the way the images are preprocessed in Depicts, the rest of the system would remain intact. Regardless of the type of image processing involved, Depicts would perform effectively if the knowledge found in the image were represented as proper objects in the form of Horn clauses.

The image domain of Depicts does not have to consist exclusively of 2-dimensional diagrams. If it is possible to define a one-to-one mapping between the image domain and the scene domain, Depicts will be able to analyze and synthesize the image.

For instance, range images are optimal candidates for the image domain. Range images record two dimensional arrays of depth measuring the distance between the camera and the surface being imaged (Archibald and Sternberg, 1986). Given the range image, it is possible to recover all the visible points of the scene, since range images are a function of only one scene property, namely depth. (Obviously, scene points hidden from the camera would not be recovered in this process.) In general there is no loss of information in the range image-scene mappings process. As a result, Depicts can be used effectively for image analysis and image synthesis.
References


Eggert, P.R. and K.P. Chow (1983) "Logic programming graphics and infinite terms", Department of Computer Science, University of California, Santa Barbara, California.


Smith, Brian (1986) "The Correspondence Continuum", invited talk at the 1986 Conference of the Canadian Society for Computational Studies of Intelligence, Montreal, Quebec.


Appendix A
An Example of a Composite Object

Section 4.2 gives the Backus Naur Form rules for an object. In section 5.2 we defined the data structure between the scene domain and an intermediate domain as

\[ \text{depicts}(\text{subclass}(\text{class}(Y), \text{class}(X)), \text{father}(\text{node}(X), \text{node}(Y))) \]

which we shall call here \(< \text{object}_1 \>\). Being a composite object itself, \(< \text{object}_1 \>\) can be subdivided into several different \(< \text{objects} \>\), all of which are relations in themselves. Note the orthogonality achieved.

\[
\begin{align*}
< \text{object}^1 > & : = < \text{composite object} > \\
< \text{composite object} > & : = < \text{predicate}(\text{object}^2, \text{object}^3) > \\
& = \text{depicts}(\text{subclass}(\text{class}(Y), \text{class}(X)), \text{father}(\text{node}(X), \text{node}(Y))) \\
< \text{object}^2 > & : = < \text{composite object} > \\
< \text{composite object} > & : = < \text{predicate}(\text{object}^4, \text{object}^5) > \\
& = \text{subclass}(\text{class}(Y), \text{class}(X)) \\
< \text{object}^4 > & : = < \text{composite object} > \\
< \text{composite object} > & : = < \text{predicate}(\text{object}^6) > \\
& = \text{class}(Y)
\end{align*}
\]
APPENDIX A. AN EXAMPLE OF A COMPOSITE OBJECT

\[
<\text{object}^5> ::= <\text{composite object}>
\]

\[
<\text{composite object}> ::= <\text{predicate(object}^7)> = \text{class(X)}
\]

\[
<\text{object}^6> ::= <\text{composite object}>
\]

\[
<\text{composite object}> ::= <\text{predicate(object}^8,\text{object}^9)> = \text{father(node(Y),node(X))}
\]

\[
<\text{object}^7> ::= <\text{composite object}>
\]

\[
<\text{composite object}> ::= <\text{predicate(object}^6)> = \text{node(Y)}
\]

\[
<\text{object}^8> ::= <\text{composite object}>
\]

\[
<\text{composite object}> ::= <\text{predicate(object}^7)> = \text{node(X)}
\]

\[
<\text{object}^9> ::= <\text{atomic object}>
\]

\[
<\text{atomic object}> ::= <\text{variable} > = \text{Y}
\]

\[
<\text{object}^7> ::= <\text{atomic object}>
\]

\[
<\text{atomic object}> ::= <\text{variable} > = \text{X}
\]
Appendix B

A Programming Example

Here is an example of how Prolog solves the task of mapping objects between the fathers domain and the tree domain (domains which are described in section 5.2). The Prolog predicate *depicts_fathers_lines* defines the mapping relation between the two domains. The interpreter takes the query and data structure from the editor, and it formulates them into a Prolog goal. In here we show how this goal is satisfied emphasizing the bidirectional flow of knowledge.

In logic, the formula for the definition of the mapping is

\[
\exists x \exists y ((x \in \text{Domain}_{\text{fathers}}) \land (y \in \text{Domain}_{\text{tree}}) \land \text{depicts_fathers_lines}(x, y))
\]

*Depicts* computes the mapping between the father domain and the tree domain by formulating Prolog goals. These goals call the predicate *depicts_fathers_lines*. In the following examples, the computed answers are in bold face.

If the *Scene_relation* is instantiated, then *depicts_fathers_lines* computes the *Image_relation*:
86

APPENDIX B. A PROGRAMMING EXAMPLE

:- depicts_fathers_lines(father(node(trees),node(softwoods)),image_relation).

Image_relation = line(point(softwoods),point(trees))
yes

If the Image_relation is instantiated, then depicts_fathers_lines computes the Scene_relation:

:- depicts_fathers_lines(Scene_relation, line(point(softwoods), point(trees))).

Scene_relation = father(node(trees), node(softwoods))
yes

If both the Scene_relation and the Image_relation are instantiated, then depicts_fathers_lines verifies that the relations map properly into each other, otherwise it fails.

:- depicts_fathers_lines(father(node(trees), node(softwoods)),
                           line(point(softwoods), point(trees))).

yes

:- depicts_fathers_lines(father(circle(trees), circle(softwoods)),
                          line(point(softwoods), point(trees))).

no

If neither relation is bound, then depicts_fathers_lines returns the Image_relation and the Scene_relation in a “general” form.

:- depicts_fathers_lines(Scene_relation, image_relation).

Scene_relation = father(node(172), node(148))
Image_relation = line(point(148), point(172))
yes
Appendix C

A Sample Session with Depicts

First of all, the user decides to enter data in query mode:

"|||" prefixes messages by Depicts

"|:" is the prompt used by Depicts and following is the user input

".:" marks end of user input

||| What do you want to classify?
|: 'trees'.

||| Into what classes can you subdivide trees?
|: 'softwoods'.
|: 'hardwoods'.
|: .:

||| Into what classes can you subdivide softwoods?
|: 'needle leaves'.
|: 'scale leaves'.
|: .:

||| Into what classes can you subdivide needle leaves?
|: 'pacific yew'.
|: 'cones woody'.
|: .: 
APPENDIX C. A SAMPLE SESSION WITH DEPICTS

Into what classes can you subdivide pacific yew?
|: none.

Into what classes can you subdivide cones woody?
|: 'clustered leaves'.
|: 'single leaves'.
|: :.

Into what classes can you subdivide clustered leaves?
|: none.

Into what classes can you subdivide single leaves?
|: none.

Into what classes can you subdivide scale leaves?
|: none.

Into what classes can you subdivide hardwoods?
|: 'opposite leaves'.
|: 'alternate leaves'.
|: :.

Into what classes can you subdivide leaves opposite leaves?
|: none.

Into what classes can you subdivide alternate leaves?
|: 'compound leaves'.
|: 'simple leaves'.
|: :.

Into what classes can you subdivide compound leaves?
|: none.

Into what classes can you subdivide simple leaves?
|: none.
Depicts transforms this information into clauses to be added to the scene domain classes:

```
subclass(class('1/softwoods'), class('0/trees')).
subclass(class('2/hardwoods'), class('0/trees')).
subclass(class('3/needle leaves'), class('1/softwoods')).
subclass(class('4/scale leaves'), class('3/needle leaves')).
subclass(class('5/pacific yew'), class('3/needle leaves')).
subclass(class('6/cones woody'), class('3/needle leaves')).
subclass(class('7/clustered leaves'), class('6/cones woody')).
subclass(class('8/single leaves'), class('6/cones woody')).
subclass(class('9/opposite leaves'), class('2/hardwoods')).
subclass(class('10/alternate leaves'), class('2/hardwoods')).
subclass(class('11/compound leaves'), class('10/alternate leaves')).
subclass(class('12/simple leaves'), class('10/alternate leaves')).
```

Depicts maps these clauses to the image domain which is illustrated in Figure C.1 and consists of the following clauses:

```
line(point('1/softwoods'), point('0/trees')).
line(point('2/hardwoods'), point('0/trees')).
line(point('3/needle leaves'), point('1/softwoods')).
line(point('4/scale leaves'), point('1/softwoods')).
line(point('5/pacific yew'), point('3/needle leaves')).
line(point('6/cones woody'), point('3/needle leaves')).
line(point('7/clustered leaves'), point('6/cones woody')).
line(point('8/single leaves'), point('6/cones woody')).
line(point('9/opposite leaves'), point('2/hardwoods')).
line(point('10/alternate leaves'), point('2/hardwoods')).
line(point('11/compound leaves'), point('10/alternate leaves')).
line(point('12/simple leaves'), point('10/alternate leaves')).
```
APPENDIX C. A SAMPLE SESSION WITH DEPICTS

The clauses in the scene domain *classes* can also be mapped to the *image domain* Venn diagram:

- contains(circle('0/trees'),circle('1/softwoods')).
- contains(circle('0/trees'),circle('2/hardwoods')).
- contains(circle('1/softwoods'),circle('3/needle leaves')).
- contains(circle('1/softwoods'),circle('4/scale leaves')).
- contains(circle('3/needle leaves'),circle('5/pacific yew')).
- contains(circle('3/needle leaves'),circle('6/cones woody')).
- contains(circle('6/cones woody'),circle('7/clustered leaves')).
- contains(circle('6/cones woody'),circle('8/single leaves')).
- contains(circle('2/hardwoods'),circle('9/opposite leaves')).
- contains(circle('2/hardwoods'),circle('10/alternate leaves')).
- contains(circle('10/alternate leaves'),circle('11/compound leaves')).
- contains(circle('10/alternate leaves'),circle('12/simple leaves')).

Figure C.1: Partial Tree Diagram of British Columbia Trees
Figure C.2: Partial Venn Diagram of British Columbia Trees
Depicts can also describe the scene domain in an English-like manner, as in:

- softwoods is a subclass of trees
- hardwoods is a subclass of trees
- needle leaves is a subclass of softwoods
- scale leaves is a subclass of softwoods
- pacific yew is a subclass of needle leaves
- cones woody is a subclass of needle leaves
- clustered leaves is a subclass of cones woody
- single leaves is a subclass of cones woody
- opposite leaves is a subclass of hardwoods
- alternate leaves is a subclass of hardwoods
- compound leaves is a subclass of alternate leaves
- simple leaves is a subclass of alternate leaves