Steps Towards A Theory of Representation Design

by

Jeffrey Van Baalen
B.S., University of Wyoming, 1978
M.S., University of Wyoming, 1980

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

at the

Massachusetts Institute of Technology

January, 1989

Copyright Massachusetts Institute of Technology, 1988

Signature of Author ______________________________

Department of Electrical Engineering and Computer Science
January 13, 1989

Certified by ______________________________

Randall Davis
Associate Professor of Management Science
Thesis Supervisor

Accepted by ______________________________

Arthur C. Smith, Chairman
Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
MAY 10 1989
ARCHIVES
LIBRARIES

Steps Towards A Theory of Representation Design

Jeffrey Van Baalen

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science at the
Massachusetts Institute of Technology, January, 1989

Abstract

A computer program was developed that designs specialized representations for problems of the sort found in the analytical reasoning section of the GRE and LSAT. These problems are intended to test a person's ability to "draw logical conclusions from information presented and to synthesize that information in order to deduce the actual structure of or interrelationships among things." [Weber83]

The system takes as input a straightforward predicate calculus translation of a problem, requests additional information if necessary, decides what to represent and how, designs representations specialized to the problem, and finally creates and executes a LISP program that uses those representations to produce a solution. Even though typically these problems are very difficult for theorem provers to solve, the LISP programs that uses the specialized representations is very efficient.

The representations the system designs are powerful because they capture the constraints of a problem, in two ways: (i) the structure of the representation resembles the structure of the thing represented and (ii) this structure enables efficient behaviors that enforce a problem's constraints by keeping those constraints invariant in the structure. More specialized representations capture more problem constraints in their structure and behavior.

As constraints get captured in a representation, fewer situations can be expressed. This reduces the space that a problem solver must consider in a specialized representation. This, in turn, results in more efficient problem solving behavior.

The goal of the system is to design a representation that captures all of a problem's constraints. The process developed to achieve this is divided into three subprocesses: *classification*, *concept introduction*, and *operationalization*. Each of these contributes in different ways to capture constraints of a problem in a representation being de-

signed.

Constraints on a problem concept are captured structurally when the concept is represented with a structure having the same properties as the concept. Classification uses a library of structures organized into a taxonomy around the constraints that they capture: Structures that capture more constraints are more specialized. Constraints on a concept are captured structurally by identifying the library structure that captures the most constraints on the concept and then representing the concept with that structure. This is done by classifying concepts in the taxonomy.

The success of classification in capturing constraints is limited by the particular vocabulary used to state a problem. Concept introduction is a way of enhancing classification. If classification fails to capture constraints on a concept, then introduction tries a different way of representing the concept. For example, if classification fails to capture all the constraints on a concept represented as a relation, introduction might try representing it as a function. This strategy works because different library structures capture different constraints and have different specializations. Representing a concept differently may be a better fit between the constraints on that concept and the constraints captured by the different representation.

Classification and concept introduction run as coroutines, trying to capture all of the constraints of a problem. As they do this, the statements of those constraints get removed from the problem. They usually fail to capture all the constraints leaving statements of the uncaptured constraints in the problem. Operationalization then tries to capture the constraints of any remaining statements by writing new procedures and using these to specialize the representations created by classification and concept introduction.

An interesting property of analytical reasoning problems is that they are usually incomplete. An important part of the representation design process developed is identifying where information might be missing from a problem and asking a user informed questions in an effort to complete the problem.

Keywords: Artificial Intelligence, Knowledge Representation, Knowledge-based Systems.

Thesis Supervisor: Randall Davis
Title: Associate Professor of Management Science

Acknowledgments

I would like to thank Randy Davis, a great and patient advisor.

I would like to thank Chuck Rich for his never ending help and support; Peter Szolovitz and Patrick Winston for their support.

I would like to thank my colleagues: Yishai Feldman, Walter Hamscher, David Kirsh, David McAllester, Mark Shirley, Howie Shrobe, Daniel Weld, Brian Williams, Peng Wu.

I would like to thank Lynne Staub, Justin Van Baalen, and Gwedolyn Van Baalen for their support and understanding.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the AI Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

Contents

1 Introduction **10**

1.1 Motivation 11

1.2 Overview of Representation Design 15

1.2.1 Representation 16

1.2.2 Specialized Representations and Problem Classes 19

1.2.3 Implementation of Representations 23

1.2.4 Knowledge about Representations 24

1.2.5 Properties of Specialized Representations 25

1.2.6 Deriving an Initial Representation 26

1.2.7 Classification 29

1.2.8 Concept Introduction 33

1.2.9 Operationalization 36

1.3 Analytical Reasoning Problems 38

1.4 Soundness of Representation Design 41

1.5 Scope and Limitations 42

1.5.1 The Class of Problems 42

1.5.2 Coverage 43

1.6 Related Work 45

1.6.1 Systems Solving Word Problems 45

1.6.2 Automatic Programming 46

1.6.3 Good Representation 46

1.6.4 Problem Reformulation 46
1.6.5 Specialized Reasoners 47
1.7 Reader's Guide 47

2 Descriptions Used in Representation Design 49
2.1 The Problem Statement Language 49
2.2 A Language for Defining Representations 50
2.2.1 Maintaining The Correspondence 53

3 Deriving an Initial Representation 57
3.1 Identifying The Primitive Concepts of a Problem 60
3.1.1 Determining that a Concept is Defined 61
3.1.2 Section Summary 61
3.2 Eliminating Irrelevant Information 62
3.2.1 Strong and Weak Irrelevance 63
3.2.2 An Approximate Strong Irrelevance Filter 64
3.2.3 Strengthening the Filter 72
3.2.4 Eliminating Irrelevance in Incomplete Problem Statements . . 75
3.2.5 Section Summary 76
3.3 Deriving Instance Definitions for the Represented Concepts 76
3.3.1 Section Summary 79
3.4 Chapter Summary 79

4 Classification 81
4.1 Implementation of Library Structures 87
4.2 The Classification Process 87
4.2.1 Answering Questions Posed by Classification 90
4.2.2 Assumptions About Library Structures 93
4.2.3 Classification Example Revisited 94
4.3 Capture Verification 97

4.3.1 How Capture Verification Works 97
4.3.2 How Capture Verification is Used 100
4.3.3 Why Capture Verification Works 101
4.4 Interaction Between Knowledge Acquisition and Capture Verification 102
4.5 Summary 104

5 Concept Introduction **106**
5.1 Introduction Rules 109
5.2 Soundness of Introduction 112
5.3 Exploratory Introduction 114
5.3.1 Exploratory Introduction in Incomplete Problems 115
5.3.2 Comparing Alternative Formulations 118
5.3.3 Formulations Including More than One Alternative Concept . 124
5.4 Other Types of Introduction 131
5.5 Extended Classification of Married 133
5.5.1 Introduction of Spouses 133
5.5.2 Classification of Spouses 134
5.5.3 Introduction of Spouse 136
5.5.4 Classification of Spouse 137
5.5.5 Introduction of Couple 138
5.5.6 Summary of Extended Classification of Married 140
5.6 Extended Classification of Child 140
5.6.1 Classification of Parents 141
5.6.2 Introduction of Children' 142
5.6.3 Classification of Children' 144
5.6.4 Classification of Children 145
5.6.5 Summary of Extended Classification of Child 146
5.7 Deriving New Mixed Constraints 146
5.8 Detecting Redundant Introductions 147

5.9 Chapter Summary 149

6 **Operationalization** **159**

6.1 Overview of the Operationalization Procedure 161

6.2 Identifying the Operational Literals in a Problem Class 166

6.3 Operationalization Sequences 167

6.3.1 Rewriting Intermediate Statements 172

6.3.2 Operationalizing Statements that Contain Existential Quantifiers 173

6.3.3 Optimizations of Operationalization 174

6.4 Generating Procedures from Statements in Operational Form 176

6.5 Soundness of Operationalization 180

7 **Representation Machinery** **181**

7.1 The Equality System and Anonymous Individuals 182

7.2 Implementation of Library ADTs 183

7.3 Solving the Example Problem 188

8 **Analysis** **192**

8.1 Semantics of Situations 192

8.1.1 Discussion 195

8.2 Attempts to Characterize The Class of Problems 195

8.3 Coverage of Analytical Reasoning Problems 196

8.4 Soundness of Representation design 198

8.4.1 The Irrelevance Filter 200

8.4.2 Concept Introduction 200

8.4.3 Classification 201

8.4.4 Operationalization 201

8.5 Complete Procedures for Answering Queries 202

9 **Related Work** **207**

9.1 Previous Systems that Solve Word Problems 207
9.2 Relationship to Automatic Programming 209
9.3 Research on Good Representation 213
9.4 Problem Reformulation 214

10 Summary **219**

10.1 The Process of Representation Design 220
10.2 Summary of An Example of Representation Design 224

Chapter 1

Introduction

It has long been acknowledged that having a good representation is key in effective problem solving. But what is a "good" representation? Most answers fall back on a collection of somewhat vague phrases, such as "make the important things explicit; expose natural constraints; be complete, concise, transparent; facilitate computation" [Winston84]. These are of some assistance, but leave unresolved at least two important issues. First, saying that a "good" representation makes the "important" things explicit really only relabels the phenomenon – How are we to know what is important? Second, while phrases like these can conceivably serve as recognizers, allowing us to determine whether a given representation is good, little progress has been made on understanding how to *design* a good representation prospectively.

I have developed a new approach to this problem with the following key properties:

- It begins with an initial problem statement, determines what to represent and assists in identifying missing information that is required to solve the problem. Thus it assists in determining what is "important" in a problem statement.

- It offers a more technical explanation of what makes for a good representation, claiming that it is one that captures constraints of a problem in its structure and behavior.

- My approach shows how to *design* a representation with these properties, then how to solve the problem using that representation.

I have implemented a demonstration of this approach, which I call the *representation design system*, and tested it on a small number of verbal reasoning problems of the sort found on graduate school level admissions tests. One of the problems, shown in Figure 1.1, is used throughout this thesis for illustration. This problem will be referred to as the FAMILIES problem.

My system takes as input a straightforward predicate calculus translation of a problem, requests additional information if necessary, decides what to represent and how, designs representations tailored to the problem, and finally creates and executes a LISP program that uses those representations to produce a solution.

> *Given:* M, N, O, P, Q, R, and S are all members of the same family. N is married to P. S is the grandchild of Q. O is the niece of M. M has red hair. The mother of S is the only sister of M. R is Q's only child. M has no brothers. N is the grandfather of O.
> *Query:* Name the siblings of S.

Figure 1.1: The FAMILIES Analytical Reasoning Problem

1.1 Motivation

My approach is motivated in large part by my own observations of the problem solving behavior people exhibit when solving problems of the sort shown in Figure 1.1, and inspired by the striking difference between that behavior and what we might call a "classroom logic approach."

The classroom logic approach begins by translating the problem into predicate calculus (Figure 1.2), then uses theorem proving to search for a solution.

One problem with this approach is that the problem specification (and hence its translation into predicate calculus) is incomplete: nothing in Figure 1.2, for instance,

M, N, O, P, Q, R, and S are all members of the same family.	sort(M,FAMILY-MEMBER), ..., sort(S,FAMILY-MEMBER)
N is married to P.	$married(N, P)$
S is the grandchild of Q.	$grandchild(S, Q)$
O is the niece of M.	$niece(O, M)$
The mother of S is the only sister of M.	$mother(S, x) \Leftrightarrow sister(M, x)$ $[sister(M, x) \wedge sister(M, y)] \Rightarrow x = y$
R is Q's only child.	$child(Q, x) \Leftrightarrow x = R$
M has no brothers.	$\neg brother(M, x)$
N is the grandfather of O.	$grandfather(O, N)$
Name the siblings of S.	find-all x : $sibling(S, x)$

Figure 1.2: Line-by-line translation of FAMILIES problem to predicate calculus. (Upper case symbols are constants. Lower case symbols in variable positions are universally quantified.)

indicates that the *married* relation is symmetric or that *grandfather* is the father of the father, etc. Once identified, that information is easily encoded as additional axioms. The harder part is knowing what is missing: on this task the classroom logic approach offers us little or no guidance.

More important from the perspective taken in this research, is that even a moderately experienced human problem solver would not proceed in this fashion, using an unstructured collection of axioms, but would instead ***design and use specialized representations***, and as a direct result produce solutions far more effectively. By a specialized representation I mean the sort of thing illustrated in Figure 1.3, which shows two statements of the sample problem in a representation people commonly use.

Such representations are powerful because they capture the constraints of a problem, in two ways: (i) the structure of the representation resembles the structure of the thing represented (i.e, they are "direct" [Sloman71]), and (ii) this structure enables efficient behaviors that enforce a problem's constraints by keeping those constraints invariant in the structure. These are both illustrated here with the example "children-of" link.

"R is the only child of Q" "S is the grandchild of Q"

Figure 1.3: Two statements in a specialized representation
A divided rectangle represents a couple; a circle represents a set of children of the same couple: solid circles are closed sets, dashed circles are sets all of whose members may not be known; the directed arc represents the "children-of" function between couples and their set of children.

The "children-of" link is a 1-1 function that syntactically captures the relation between a couple and their set of children. This link also has specialized behavior because it is a 1-1 function. One behavior uses the fact that "children-of" is a function to combine children sets when they are the "children-of" the same couple (i.e., if two couples are equal, the objects their "children-of" links point to are equal). Because it is a *1-1* function, another behavior does the same to couples that are parents of the same children sets.

Using the fact that couples are disjoint if we have what appears to be two distinct couples (the top boxes in Figure 1.3) and also know that they share an individual (Q), then they are in fact the same and hence can be combined. Using a behavior that embodies this fact and the behaviors associated with "children-of," the two structures in figure 1.3 can be combined to yield Figure 1.4.

This combination process is of fundamental importance because it computes all the relevant consequences of the conjunction of statements as they are combined. This process is tightly constrained by the syntax of the representations. For instance, figure 1.4 represents all the relevant consequences of the conjunction of the structures in figure 1.3 (e.g., "S is the child of R"). The consequences are computed by the procedures (behaviors) described above that perform local operations on the two

structures.

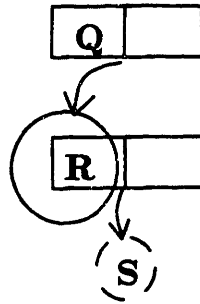

Figure 1.4: Composition of the Structures in figure 1.3.

The representation design system accepts a problem stated in predicate calculus as input, for example, the system is given the predicate calculus version of the FAMILIES problem (i.e., exactly those statements shown on the right in figure 1.2). The task of the system is to design representations like these, by picking out the important concepts in the problem (such as "couples" or "the siblings of an individual"), and finding ways to operate on them using special purpose manipulations of the sort illustrated by Figure 1.4. My system chooses what to represent and how, then solves the problem using those representations. In fact, it solves the problem by designing and using, among others, the representations illustrated in Figure 1.3 and Figure 1.4.

The representations produced by the representation design system are expressed in terms of abstract data types, data structures and associated procedures. The data structure of a type is used to implement the structure part of a representation; the procedures are used to implement the behaviors. For example, the representation design system designs a representation for couples in the FAMILIES problem. Instances of this representation serve roughly the same function as the rectangles in figure 1.3 and figure 1.4: they represent specific couples like the one containing Q. The data structure part consists of two slots for holding the two individuals in a couple. There are several procedures associated with couple. One procedure causes separate couples to be combined if they have a common member.

Note that while my system designs representations in terms of abstract data types,

language is not so much the issue: much the same effect can no doubt be accomplished by a skilled logic programmer carefully selecting axioms, lemmas, and special purpose inference rules. Whatever the language, the important point is the careful selection of representations specialized to the problem and capturing the constraints of the problem.

1.2 Overview of Representation Design

This section provides an overview of the process of representation design that has been developed in this research. The next chapter discusses the descriptions that the system uses. The four subsequent chapters describe in detail the different elements of the representation design process.

Representation design begins with a *problem statement*, a collection of statements in a sorted first order logic, along with one or more queries written in a separate query language described later. The constant symbols in the statements refer to what we will call *concepts*: individuals, relations, and functions. For example, the FAMILIES problem the statement,

$$\forall x \forall y [married(x,y) \Leftrightarrow married(y,x)],$$

refers to the concept "married."

A familiar notion in logic is that statements constrain the possible models of a problem. For example, the above statement constrains all models of the FAMILIES problem to be models in which "married" is symmetric. An important part of my method is to design representations that are constrained in the same way as the models of a problem.

1.2.1 Representation

A *representation* is a mapping between concepts and syntactic structures in a representation language. For instance, the representation of a problem expressed in first order logic is the mapping between the concepts mentioned in the problem statement and syntactic structures of first order logic. In classical first order logic there are three syntactic structures: constants (i.e., symbols that appear as terms), relations (structures that appear as atomic formulas and consist of a symbol followed by a list of arguments), and functions (structures that appear as terms and consist of a symbol followed by a list of arguments). We can determine from the use of a symbol whether the concept it denotes is represented as an individual, relation, or function. For example, in the FAMILIES problem statement "married" is represented as a relation because the symbol *married* appears in atomic formulas of the form $married(term_1, term_2)$.

Let us consider the sense in which a syntactic structure can be said to represent something. The predicate calculus structure function represents the notion of a "function" by capturing the "single valued" property (among other things). Functions are written as a symbol followed by a list of arguments and must appear as terms in predicate calculus statements. The combination of function application (i.e., $F(x_1, \ldots, x_n)$) and the restriction that they can only appear as terms captures the "single valued" property. In addition, the unification procedure for first order logic preserves the single value property by enforcing the fact that $F(x)$ can only unify with $F(y)$ if x unifies with y. Thus, it is the combination of structure (i.e., $F(x_1, \ldots, x_n)$) and behavior (unification) that captures this constraint.

My methodology takes explicit account of the fact that a combination of structure and behavior is required to represent. The structures appearing in the range of the representations that my system designs have behavior associated with them. The structures capture constraints by having features that correspond to those constraints.

For example, in the FAMILIES representation, married couples are represented as a structure with two slots. The feature, two slots, corresponds to the property "married couples have exactly two individuals in them."

Structures also have procedures associated with them that provide them with behavior enforcing their constraints [1]. For example, the procedures associated with the structure for married couples interpret each slot in the structure as containing exactly one individual. It is through the combination of structure (two slots) and behavior of that structure (each slot is interpreted as containing one individual) that structures capture properties.

The procedures of a structure enforce its constraints as the structure is used to represent things, i.e., as instances of the structure are created and modified. For example, there is a procedure that produces a single couple from the two separate couples $< A, B >$ and $< A, C >$. This procedure enforces the fact that couples are disjoint by combining two couples that are separate structures when they share a member. When the new structure is created, the procedures make $B = C$ to ensure that the new couple structure has exactly two members.

Notice that if $B \neq C$ in the example above, stating that $< A, B >$ and $< A, C >$ are couples is contradictory. In this case, the structure used to represent couple must signal a contradiction to enforce its constraints. In general, when a representation captures a constraint, it signals a contradiction if a collection of facts is stated that violate the constraint.

A representation can be *specialized* to a problem in varying degrees. In a more specialized representation, some problem concepts are represented with more specialized structures. More specialized structures capture more properties of the concepts they represent. For example, consider two different representations of a problem in first order logic: In the first, the concept "mother" is represented as the binary relation

[1]Or, put slightly differently, the procedures preserve the semantics of the representations.

mother, while in the second it is represented as the function *mother-of*. A problem statement using the relation representation might contain the statements:

$mother(A, B)$
$\forall x \forall y \forall z[mother(x, y) \land mother(x, z) \Rightarrow y = z]$

The representation in which "mother" is represented as the function *mother-of* is more specialized. In particular, the same information expressed in the two statements above is expressed in the single statement $mother\text{-}of(A) = B$ because the single valued property of "mother" is captured by the structure *mother-of*.

Because specialized representations capture properties, they aid in problem solving by reducing the search space that a problem solver must consider. Let us consider solving a simple problem in two different representations, one more specialized than the other. Suppose we are using a resolution theorem prover as our problem solver. In the first representation, the problem of interest is expressed in terms of, among other things, a relation R that is single valued. That is, the problem statements imply:

(1) $\forall x \forall y \forall z[R(x, y) \land R(x, z) \Rightarrow y = z]$

Suppose further that in order to solve the problem the theorem prover must find a contradiction in the two statements:

$\forall x \forall z \exists y[R(x, y) \land S(y, z)]$
$\forall x \forall z \exists y[R(x, y) \land \neg S(y, z)]$

Finding the contradiction requires using the fact that R is single valued — that for every x there can be at most one y such that $R(x, y)$ — to determine that the negative and positive occurrences of S resolve. The solution path for a simple resolution theorem prover will have length greater than or equal to four to reach this conclusion using (1) above. For the interested reader, figure 1.5 gives the clausal form of this problem and one possible shortest solution.

In the second, more specialized, representation, the concept "R," which was represented as a relation, is represented as the function symbol F. In this representation:

The clausal form of the problem:
(2)$R(x_1, F_1(x_1))$
(3)$S(F_1(x_1), z_1)$
(4)$R(x_2, F_2(x_2))$
(5)$\neg S(F_2(x_2), z_2)$
(6)$\neg R(x_3, y_3) \vee \neg R(x_3, z_3) \vee y_3 = z_3$
Here is one possible shortest solution path:

Conclusion	*Justification*
(7) $\neg R(x_1, z_3) \vee F_1(x_1) = z_3$	(2) and (6)
(8) $F_1(x_1) = F_2(x_1)$	(7) and (4)
(9) $S(F_2(x_1), z_1)$	(8) and (3)
(10) $\square$	(9) and (5)

Each of the steps, except (9), is an application of the binary resolution rule. Step (9) requires a special inference rule for equality.

Figure 1.5: A proof with the "single valued" property stated as an axiom.

- the constraint of (1) is enforced by the theorem prover's unifier without the need for any explicit axiom
- the problem formulation becomes simply:
  $S(F(x), z)$
  $\neg S(F(x), z)$
- and the required inference is reduced to one step.

In the second, more specialized, representation the example problem's search space has been reduced, allowing the theorem prover to find a solution more directly.

1.2.2 Specialized Representations and Problem Classes

The representation design process incrementally specializes a representation by selecting more and more specialized structures. One example of a step in this process is specializing "mother" from a relation to a function.

One question that arises about this process is: How long should it continue? How

specialized should specialized representations be? Should they take advantage of every detail of a particular problem? The advantage of doing so is that every constraint of a problem, no matter how serendipitous, gets captured in the specialized representation. The disadvantage is that the resultant representation is irrelevant to every other problem.

Given a problem, my system designs the most specialized representation that it can for that problem's class.

Intuitively, two problems are in the same class when the same general constraints are relevant to solving them. They can differ in the individuals they mention, in the particular relationships between those individuals, or in which individuals are constrained in a particular way. For example, a problem in the same class as the FAMILIES problem would refer to the same collection of concepts (e.g., *married*). However, it could mention a different collection of individuals and could have a different number of individuals being married.

In defining problem class more formally, problem statements are divided into three types: those that we term *specific* because they mention only individuals (these include existentially quantified variables that are not in the scope of any universally quantified variables); those that we term *general* because they do not mention individuals (i.e., have only universally quantified variables or existentially quantified variables in the scope of universals); and those that we term *mixed* because they contain both individuals and universally quantified variables.

To capture the intuition that specific individuals do not affect a problem's class, we map a problem into its class by generalizing its specific and mixed statements: replacing named individuals with existentially quantified variables. For example, we generalize the statement

$$\forall x \neg brother(M, x)$$

to

$\exists p \forall x \neg brother(p, x).$

Two problems are in the same class when they generalize to logically equivalent sets of statements.

There are good reasons for specializing to a problem class instead of attempting to specialize to a particular problem. One reason is reusability of the specialized representation. Another reason is that the efficiency arising from specializing begins to diminish rapidly when we try to continue specializing beyond the class, while at the same time the cost of specializing goes up sharply. These points are discussed in more detail in Chapter 8.

Although the definition of problem class does not explicitly refer to a problem's query, problem class is defined in terms of what is relevant to solving a problem. This depends on what is being asked. The representation design system accounts for this by eliminating irrelevance from a problem statement before it derives a class description. The definition of irrelevance and the system's technique for eliminating it are discussed in Chapter 3.

After eliminating irrelevance from a problem statement, the system creates two sets of statements. One set contains the specific and mixed statements of the problem. The other set contains a description of the problem class, i.e., the specific statements are generalized to replace named individuals by existentially quantified variables. The system then designs a collection of structures with respect to the problem class.

As a representation is specialized, constraints of the problem class get captured. When a representation captures a constraint, the statement of that constraint is removed from the problem. For example, when *mother* is specialized to a function, the statement expressing the "single valued" constraint is removed.

Representation design continues until all of the statements in a problem class are captured or until the system has no more specializations that it can try.

The specific and mixed statements are then translated into the specialized representation by creating instances of the structures designed from the class description. Let us call this set, a *problem situation.* As the instances are created, the constraints of the problem class are enforced by the specialized representation. For example, the specialized representation designed for the FAMILIES problem captures the symmetry of "married" and, therefore, when a specific statement like $married(N, P)$ is translated into this representation, the symmetry of "married" is enforced. As a result, the situation representing $married(N, P)$ will also represent $married(P, N)$.

When a representation captures all of the constraints of a class, all the situations created with it will satisfy the constraints of the class. [2]

In general, when a representation captures a constraint, the procedures of that representation respond when a specific or mixed statement is added to a problem situation by adding additional new facts. This is required to ensure that the constraint is maintained. For example, when a representation captures the constraint of the statement,

$$\forall x \forall y [married(x, y) \Rightarrow married(y, x)],$$

it must ensure that in any situation in which an individual, say N, is married to another individual, say P, that P is also married to N. One way to do this is to add $married(P, N)$ to the situation when the statement $married(N, P)$ is added.

There is an important special case of capturing called *structure capturing* in which a constraint is captured without having to add additional facts as in the above example. Consider the structure actually designed for "married" in the FAMILIES problem and how it captures the symmetry constraint. Married couples are represented with sets of size two. This avoids having to do any extra work to enforce the symmetry constraint because, using the fact that $\{x, y\} = \{y, x\}$, the combination of the structure and procedures associated with the set representation make it the case that the statement $married(N, P)$ is the same as $married(P, N)$. Thus, constraints on a concept are

[2] The notion of satisfaction is not usually applied to data structures. A formal definition of this satisfaction relationship is given in chapter 8.

captured structurally when the concept is represented with a structure having the same properties as the concept.

The representation design system captures constraints structurally by matching the properties of structures that it knows about with the constraints of concepts that it needs to represent, e.g., it captures the symmetry of "married" by matching that constraint with a property of the set representation.

The system prefers to capture constraints structurally. It has a library of useful structures and always tries to capture constraints with those structures first. It resorts to enforcing constraints by adding facts to problem situations only when it can not find a structure for capturing a constraint. For example, only if it fails to find sets for representing "married," will it enforce the constraint of the statement

$$\forall x \forall y [married(x, y) \Leftrightarrow married(y, x)],$$

by adding $married(y, x)$ to situations whenever $married(x, y)$ is added.

1.2.3 Implementation of Representations

As noted earlier, a representation is a mapping between concepts and structures with behavior. The representation design system implements structures in terms of abstract data types (ADTs). The data structure part of an ADT is used for implementing structures and the procedures are used for implementing behaviors. Access to the data structure "inside" an ADT is controlled by its procedures, preventing arbitrary manipulations of the data structure. The procedures of the ADTs that my system designs enforce constraints by maintaining properties in their data structure as information is added to problem situations.

For example, when "married" is represented as a relation, the system designs an ADT, denoted MARRIED, that maintains a list of the pairs of individuals that are known to be married in a problem situation. MARRIED captures the symmetry of "married" because is has a procedure that adds $< x, y >$ to the list in MARRIED

whenever $< y, x >$ is added. The MARRIED ADT controls access to its internal list so that it is impossible for $< x, y >$ to get on the list without $< y, x >$ also appearing there. Thus, using this ADT, no problem situation can be constructed which violates the symmetry constraint.

In general, there are three different kinds of procedures associated with ADTs. The first kind add information to problem situations by adding to the ADT instances. For example, one procedure adds new pairs of individuals to MARRIED. The second kind of procedure enforces properties. The procedure associated with MARRIED that enforces symmetry is an example of this. The third kind answers queries by inspecting the structures built by ADTs. For example, another procedure associated MARRIED checks to see whether two individuals are married in a problem situation by searching the list for a particular pair.

In the example given in this section, "married" is represented by the syntactic structure MARRIED. Note that we will often be somewhat loose in exposition and refer to structures like MARRIED as representations.

1.2.4 Knowledge about Representations

The representation design system synthesizes specific representations from a library of prototypical structures. For example, there is a library prototype called **relation**. Representations of individual relations are created by instantiating the prototype **relation**. For example, MARRIED is an instance defined in terms of **relation**. Here is its definition:

MARRIED: **relation**(FAMILY-MEMBER,FAMILY-MEMBER).

The relations, functions, and individuals in a problem are represented by instantiating the library prototypes **relation**, **function**, and **individual** respectively.

Let us now be specific about the typographic conventions I have been using. They

are summarized in figure 1.6.

Ontological class	*Typographic convention*	*Example*
concept	quoted	"married"
symbol	italics	*married*
representation	small-caps	MARRIED
prototype	typewriter font	`relation`

Figure 1.6: Typographic conventions

1.2.5 Properties of Specialized Representations

The next four sections outline the four processes of representation design. The process as a whole tries to design specialized representations that are *fully expressive*, *fully constrained*, and *maximally specialized*. Each of the four processes contributes to representation design in different ways; their contributions will be explained in terms of the properties noted.

A representation is fully expressive with respect to a class if every problem situation in the class is representable. By definition, the original problem statement is fully expressive. Thus, by definition, the representation design system starts out with a fully expressive representation. The system must be sure to preserve this property as a problem's representation is specialized.

A fully constrained representation enforces all of the constraints of a problem's class. Another way to say this is that a representation is fully constrained with respect to a class if the structures that can be created in it always satisfy the constraints of the class. This is, of course, what the representation design system tries to achieve.

A representation is maximally specialized with respect to a problem class when the representation design system knows no more specialized structure for implementing a complete and fully constrained representation for that class. This is the way the representation design system goes about generating a fully constrained representation.

1.2.6 Deriving an Initial Representation

To get things started, the representation design system develops a description of a problem's initial representation by identifying the concepts in the problem and determining whether they are represented as individuals, relations, or functions. This information is implicit in the use of concepts in the problem statement and must be made explicit.

A straightforward way to develop this description is to determine the syntactic category of each symbol denoting a concept. For example, from the statement $married(N, P)$, the representation design system determines that *married* is a relation because the statement uses *married* in an atomic formula of the form $married(term_1, term_2)$. Therefore, MARRIED is defined in terms of `relation`. When a concept is used in a similar expression, i.e., symbol followed by a list of arguments, and the expression appears as a term, the concept is defined as a function.

This straightforward approach will result in a description of a fully expressive representation because the initial representation of the problem is fully expressive and we have included all of the concepts mentioned in it. However, subsequent processes of representation design will design specialized representations for all the concepts that appear in the initial description and some of these concepts may be unnecessary either because it is not relevant to solving the problem or because it is logically redundant.

Instead of including all concepts, the system attempts to develop a description of the smallest set of concepts that constitute a fully expressive representation. There are two classes of concepts that the system attempts to exclude in this process: irrelevant concepts and redundant concepts.

Problems often contain concepts that are not relevant to their solution. We do not want to design specialized representations for irrelevant concepts. Therefore, the system attempts to exclude irrelevant concepts from its description using a technique

described in chapter 3.

Problem statements also, at times, contain redundant information in the form of definitions. It is usually redundant to capture constraints on a defined concept if we have captured all the constraints on the concepts in its definition. For example, *grandchild* is defined in terms of *child*. If we design a representation to capture all the constraints on *child*, it is not necessary to design a separate representation for *grandchild*.

While deriving the description of an initial representation, the system tries to identify defined concepts and exclude them from the description. This effort is heuristic and a subsequent process of representation design may decide to represent a concept that is excluded at this point. The techniques that the representation design system uses to identify defined concepts are discussed in Chapter 3.

Once the representation design system has identified a set of concepts that it believes is the smallest set sufficient to describe a fully expressive representation, it uses this as its description of the initial representation.

The rest of the processes of representation design work to capture all the constraints of a problem class and thereby design a fully constrained representation.

As noted, the constraint of a general statement has been successfully captured in a representation when the representation will create only problem situations that satisfy the constraint. The constraint of an unconditional specific statement is captured when there are representations defined for all the concepts in the statement. For example, the constraint of the statement $married(N, P)$, namely that N is married to P, is captured simply by stating that fact and, as long as there is a representation for *married*, it can be stated.

The strategy for capturing the constraint of a conditional specific statement is described in chapter 6.

The strategy for capturing mixed constraints depends on whether or not the constraint is a *restriction.* A restriction is a mixed statement that restricts the individuals that can stand in some relation to a particular individual. Here are two examples of mixed constraints that are restrictions:

$\forall x \neg brother(M, x)$
(restricts the individuals that can be brothers of M),
$\forall x[child(Q, x) \Leftrightarrow x = R]$
(restricts the individuals that can be children of Q).

By contrast, the statement

$$\forall x \; brother(M, x)$$

is not a restriction because it does not restrict the brothers of M.

My system does not design representations that capture restrictions directly. Instead it explores ways to reformulate problems so that restrictions become specific constraints. For example, the constraint of the statement, $\forall x \neg brother(M, x)$, is captured by reformulating the `relation` *brother* as a `function` *brothers* that maps from an individual to his/her set of brothers. This reformulation transforms the above statement into $brothers(M) = \emptyset$. Notice that this is a specific statement, i.e., it does not contain any universally quantified variables. Similarly, the statement

$$\forall x[child(Q, x) \Leftrightarrow x = R]$$

is captured by reformulating *child* as a `function` *children* that maps individuals to their sets of children. Then this statement becomes the specific statement $children(Q) = \{R\}$.

The processes that design a fully constrained representation are called *classification*, *concept introduction*, and *operationalization.* Classification and concept introduction run as coroutines to capture the constraints on concepts appearing in the initial description. They capture as many constraints on a concept as they can by identifying more and more specialized library structures for representing those concepts. As classification and concept introduction capture constraints, the statements of those constraints get removed from the problem. Operationalization then tries to capture

the constraints of any remaining statements by writing new procedures and using these to specialize the representations created by classification and concept introduction.

1.2.7 Classification

Recall that the constraints on a concept are captured structurally when the concept is represented with a structure having the same properties. The representation design system captures constraints structurally by classifying problem concepts in a taxonomy of structures that are organized around the properties of the structures. (see figure 1.7). Structures with more properties are more specialized than structures with fewer properties. For example, the structure **1-1 function** is more specialized than **function** because it has an additional property.

The perspective taken in this thesis is that properties are viewed as constraints that need to be enforced. For example, **function** enforces the "single valued" constraint and **1-1 function** enforces the additional constraint that the image of every domain element is unique.

Now consider what happens when the representation of a concept F is specialized from a **function** to a **1-1 function**. When F is represented as a **function**, problem situations can have multiple domain elements mapping to the same range element. When F is represented as a **1-1 function**, it is more constrained because the legal situations in the new representation are a subset of those in the previous representation, i.e., those situations in which domain elements map to unique range elements.

Classification proceeds by "pushing" concepts down to a leaf node in the appropriate taxonomy. In order to determine where to begin classifying a concept, the system inspects the definition of the initial representation for that concept. Classification begins at the node mentioned in the definition. For example, the representation for *married* is initially defined as

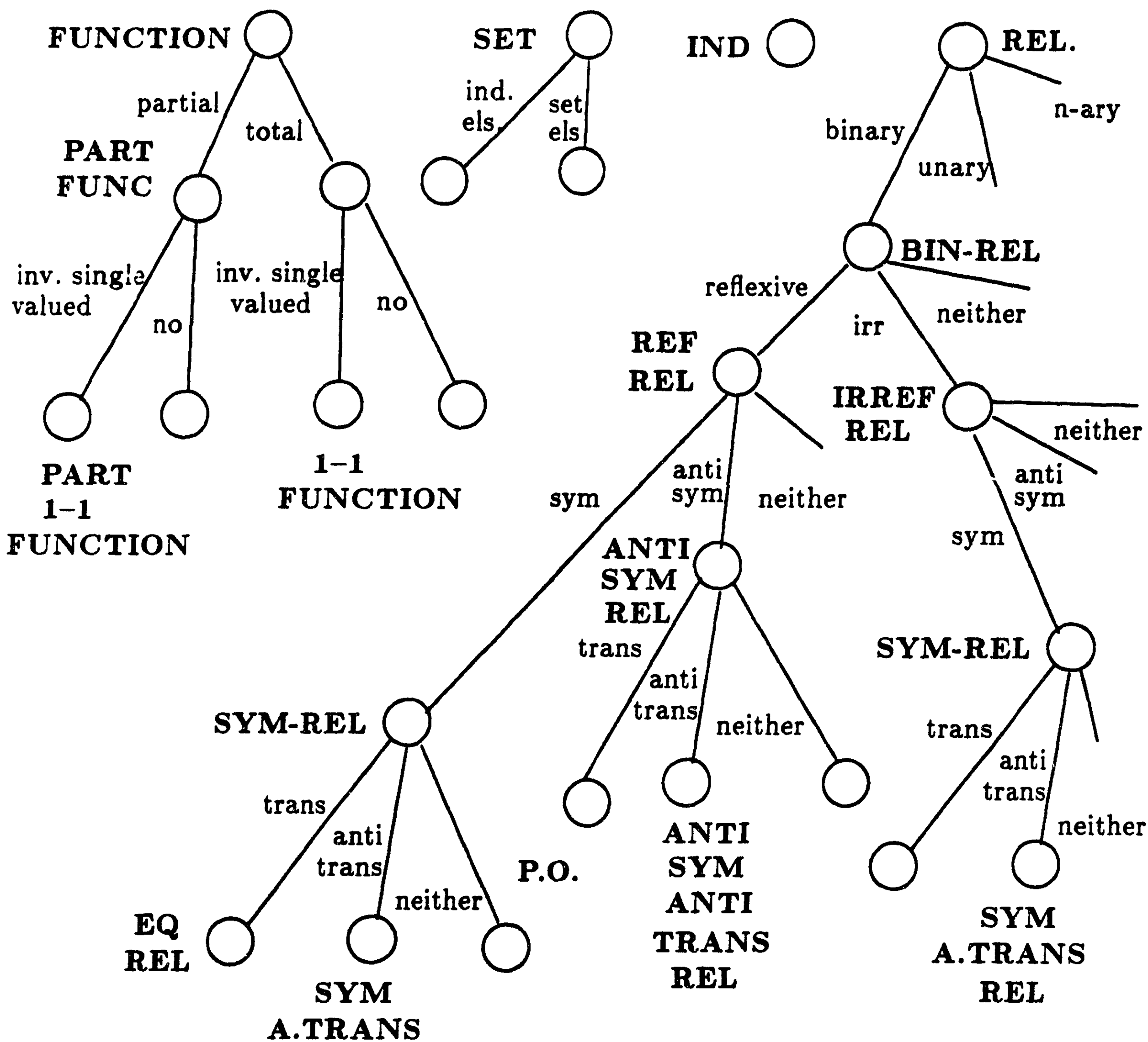

Figure 1.7: Part of the structure taxonomy

MARRIED: **relation**(FAMILY-MEMBER,FAMILY-MEMBER)

so the classification of *married* begins at the **relation** node.

As it reaches each node in the taxonomy, the system tries to show that the concept has one of the properties labeling the links leaving that node. For example, the classification of *married* begins with the system looking at the links leaving the **relation** node and tries to determine whether *married* is unary, binary, or n-ary. It is found to be binary simply by looking at a problem statement that mentions it.

In general, to determine whether a concept has a property, the representation design system looks for a statement in the problem that indicates that the property holds. For example, to determine that *married* is symmetric, the representation design system looks for a statement of the form:

$$\forall x \forall y [married(x,y) \Leftrightarrow married(y,x)].$$

This technique by itself often fails because properties can be stated in many ways. The system also has a heuristic technique (discussed in Chapter 4) that tries to transform statements into a form that classification is expecting. Since the technique is heuristic, the system may fail to determine that a concept has a property even when the property is stated.

When the representation design system is unable to show that a concept has any of the properties labeling the links leaving a taxonomy node, it asks the user about each property in turn. For example, if it is not able to determine whether or not *married* is symmetric, it asks the user (presumably the answer is "yes"). If the answer were "no," the system would ask whether it is antisymmetric, and so on.

When the system obtains positive information from the user about a property, it adds the statement of the property to the problem. For example, when the user replies that *married* is symmetric, the system adds the statement (shown above) expressing that fact.

Not all nodes in the library taxonomies are labeled by a library structure because not all combinations of constraints have useful representations. If classification of a concept succeeds in pushing down to a new node in the taxonomy and that node is labeled, the representation of the concept has been specialized. For example, when *married* is classified as a symmetric relation, its representation is specialized to **symmetric-relation**.

As constraints get captured during classification, the statements of those constraints are identified by a process called *capture verification*, then removed. Each time the classification of a concept reaches a leaf node in the taxonomy, capture verification checks the general statements referring to the concept to see whether their constraints have been captured. It checks a statement by attempting a constructive proof using the representations of the concepts in the statement. If the situations created with those representations satisfy the statement, then its constraint is captured.

Capture verification constructs proofs for general statements, so it must have a way to construct general situations, i.e., situations in which the particular individuals mentioned do not matter. It does this using anonymous individuals, concluding that anything that is true of an anonymous individual will be true of any individual put in its place and is, therefore, true of all individuals. Chapter 4 demonstrates the validity of this use of anonymous individuals to prove general statements.

For conditional statements, capture verification constructs a problem situation representing the antecedent of the statement and then checks that situation to see if the consequent is true. If so, then those representations capture the constraint of that statement. For example, consider how capture verification checks the following statement:

$$\forall x \forall y [married(x,y) \Leftrightarrow married(y,x)].$$

This is checked by treating one side of the by-conditional as the antecedent and constructing a problem situation representing it, i.e., a situation in which one anonymous

individual, say A, is married to another, say B It then inspects the situation to see if it is also the case that B is married to A. When married couples are represented as sets of size two, this proof will succeed. The other direction of the by-conditional must also be checked and since that proof will also succeed, the system concludes that the constraint of the statement is captured.

1.2.8 Concept Introduction

Classification has an important limitation: Its success depends on the particular vocabulary used to state a problem. The FAMILIES problem, for example, is stated in terms of *married*, which is classified beginning with **relation**. None of the specializations of a **relation** capture the fact that married couples are all of size two. However, if the problem had been stated in terms of *couples*, classification would have been more successful because *couples* are a specialization of **set** that take advantage of size constraints, i.e., **fixed-size-set**. Introduction enhances classification in this example by introducing *couples* when it detects that the size constraint on *married* is not captured by MARRIED. This illustrates that, given a different vocabulary, classification is more successful because it allows different and sometimes more specialized knowledge in the structure library to be brought to bear in representation design.

More generally, introduction extends the classification process by adding a new concept to a problem when classification of an existing concept does not capture all of its constraints. The introduced concepts allow the system to view an existing concept differently. For example, introducing *couples* for *married* allows the system to view "married' differently.

When a new concept is introduced, a corresponding representation is also introduced. For example, the concept married couple discussed above is actually introduced in several steps. I will use the first step to illustrate the parallel introduction of new concepts and representations. In the first step, the concept *spouses* is introduced.

This is a function mapping individuals to the sets of individuals who are their spouses. A representation, SPOUSES is also introduced and defined as a `function` mapping an `individual` to a `set` of `individuals`.

A new concept is always defined in terms of an existing concept in such a way that the semantics of a problem are not changed. For example, *spouses* is defined as

$$\forall x \forall y [y \in spouses(x) \Rightarrow married(x, y)].$$

There are two reasons for introducing new concepts. The primary reason is to give the representation design system access to different representation design knowledge: the library structure in the new definition enforces different constraints, provides different operations, and has different specializations. In the example above, for instance, since MARRIED does not capture all the constraints on "married," *spouses* is introduced to gain access to representations that capture different constraints.

The other reason to introduce a new concept is that it enables reformulation of the statements in the problem. This is useful because it often allows new properties to be discovered in the problem statement. Reformulation is accomplished by treating the logical definition of a new concept as a rewrite to perform on the problem statements to derive new statements. For example, when *spouses* is introduced, it is defined as

$$\forall x \forall y [y \in spouses(x) \Leftrightarrow married(x, y)].$$

This is treated as a rewrite rule that derives a statement referring to *spouses* for every statement referring to *married*. For instance, this rule derives the statement $P \in spouses(N)$ from $married(N, P)$.

As new concepts are introduced, the representation design system explores a space of alternative problem formulations. For example, when *spouses* is introduced, an alternative formulation of the problem is created. There are then two formulations: one in terms of *married* and one in terms of *spouses*. The system has a technique for estimating the cost of problem formulations so that alternatives can be compared

(discussed in Chapter 5). Alternative formulations are maintained because the cost estimation technique works only when the relevant alternative concepts have been fully classified. For example, the formulation of the problem in terms of *married* can not be compared to the formulation in terms of *spouses* until *spouses* is fully classified.

Recall that earlier I discussed the strategy that the representation design system uses for capturing restrictions. A restriction is a mixed statement that restricts the individuals that can stand in some relation to a particular individual, for example,

$\forall x \neg brother(M, x)$.

The system captures a restriction by introducing a concept that reformulates it to a specific statement. The system captures the statement above, for instance, by introducing the concept of *brothers*, a mapping from an individual to his set of brothers. When this is introduced, the statement is reformulated as $brothers(M) = \emptyset$. Notice that this is a specific statement, i.e., it does not contain any universally quantified variables. Furthermore, the instance BROTHERS is defined as a **function** mapping an **individual** to a **set** of **individuals**. The library structure **set** has an assignment procedure for assigning a set the value of a constant set such as $\emptyset$.

Extended Classification

Classification extended by introduction is called *extended classification.* Extended classification is interesting because, while the two processes that are involved in it are fairly simple, the behavior of the combination can result in sequences that significantly reformulate a problem. Several examples of extended classification are given in Chapter 5; one example shows how extended classification of *married* results in the following sequence of introductions:

1. The concept *spouses* is introduced. This is a function from individuals to the sets of individuals to whom they are married.

2. The concept *non-empty-spouses* is introduced. This is a partial function from individuals to the non-empty sets of individuals to whom they are married.

3. The concept *spouse* is introduced. This is a partial function that captures the fact that individuals have at most one spouse.

4. The concept *couple* is introduced. This is a partial function from individuals to the married couple that they are members of. COUPLE captures the following facts: not all individuals are married, married couples are disjoint from all other married couples, married couples contain exactly two members.

1.2.9 Operationalization

In the ideal case, extended classification fully constrains and maximally specializes the initial representation. Given a collection of concepts, classification comes up with the most specialized, most constrained collection of representations while introduction modifies the collection of concepts when they fail to capture all of a problem's constraints. These processes can, however, fail to produce a fully constrained representation because there are some combinations of constraints that the library structure can not capture. In that case, the representation design system makes a final effort through a process called *operationalization.*

Operationalization tries to capture constraints by writing procedures that respond to any new information added to situations, in order to enforce the constraints of statements. Each procedure written for a problem statement responds to the addition of a specific kind of information by adding still additional information to capture that statement's constraint.

Consider an example: Suppose that in designing a representation for the FAMILIES

problem, extended classification produces the concept *siblings*, a function from individuals to their sets of siblings. Sets of siblings are defined in terms of **set**. Suppose further that the following statement has been left uncaptured,

$$\forall x \forall y\, x \in siblings(y) \Rightarrow y \in siblings(x).$$

Operationalization will write a procedure that adds y to the siblings of x whenever x is added to the siblings of y. This new procedure captures the constraint of this statement whenever an element is added to a set of siblings in a problem situation.

Notice that if this operation were the only one that could change sibling sets, it would capture the constraint of the statement above because it would ensure that no problem situation could be created in which y is a member of the siblings of x unless x is also a member of the siblings of y.

Of course, in general, different kinds of information can be added to a problem statement. Therefore, operationalization must generate a procedure like the one above for every kind of information that can be added.

It turns out that the kinds of information that can be added to a problem situation are restricted by the kinds of operations that can add information. Operations that can add information are restricted by the procedures that already exist in the abstract data types implementing the problem representation.

Even after operationalizing, a representation may not be fully constrained. Fortunately a representation that captures most of a problem's constraints is still useful because, we are still better off than we were with the initial representation. The reason is that as constraints get captured in a representation, the space that must be searched by a problem solver using that representation is reduced. Furthermore, the representation design process removes statements from a problem as they are captured. So, when representation design terminates, any uncaptured statements will still be present. Therefore, the result of representation design is a specialized representation and a *smaller* collection of statements (the uncaptured ones) that the

problem solver must reason about explicitly in that representation.

In effect, the problem solver uses the specialized representation to accelerate the problem solving process in the same way that specialized reasoners have been used to accelerate theorem proving. There are issues involved in the interface between a theorem prover and the specialized representation which I have not yet attempted to work out. However, the approach used in [Miller & Schubert 88] should work here.

1.3 Analytical Reasoning Problems

The current system is designed to specialize representations for analytical reasoning problems. These problems appear on intelligence tests like GREs and LSATs. They are intended to test a person's ability to "draw logical conclusions from information presented and to synthesize that information in order to deduce the actual structure of or interrelationships among things." [Weber83]

Analytical reasoning problems present a collection of facts about a specific situation and then ask questions that require deduction from those facts. The problems do not ask questions that require induction from a set of facts. There are four types of questions:

1. Is some specific fact or restriction true of a problem situation? For example, "Is it true that M has no brothers?"

2. Is it possible for some specific fact or restriction to be true in a problem situation? For example, "Is it possible for N to be the brother of M in the current situation?"

3. Find all the individuals that stand in some relation to a specific individual in a problem situation. The FAMILIES problem question is an example of this type: "Find all the individuals that are siblings of S." This type of question may also

ask if the individuals found are all the individuals that can have this property or are just all that can be found in the problem situation. For example, a question may ask us to find all the siblings of an individual in a problem and then ask whether it is possible for there to be others.

4. Find the individual that stands in some relation to a specific individual. This question assumes there can be only one such individual. For example, "Find the mother of S."

The representation design system accepts problems with the types of queries described above and designs a representation specifically to answer those queries. For example, the system usually reformulates problems so that find-all queries are translated into find-the queries because find-the queries can be answered more efficiently. For instance, the FAMILIES problem is initially stated in terms of the relation *sibling*. Given the query, "find all the siblings of S," for the FAMILIES problem, the system reformulates the problem in terms of *siblings*, a function mapping an individual to his/her set of siblings so that the query becomes, "find the sibling set of S."

The system also reformulates problems that have queries asking about mixed facts because these, too, are often very expensive to answer. For example, if instead of the statement $\forall x \neg brother(M, x)$, the FAMILIES problem contained the query, "Is it necessarily the case that M has no brothers?" the system would reformulate the problem in terms of brother sets.

Here is how each of the different types of queries is answered: [3]

1. To determine if some specific fact is true in a problem situation, the system inspects the situation to see if the fact is present. To determine if a mixed statement is true, the statement is expanded into a conjunction of specific statements. Then the situation is checked to see if each statement is necessarily

[3] These methods are not complete, in the sense that a fact may follow from a problem situation and not be found by these methods. Chapter 8 describes how to extend them to be complete.

true. Finally a possibility query is done. For example, to answer the question, "Are A, B, and C the only children of S?" the system first checks that A, B, and C are children of S and then it checks to see if it is possible for any other individual to be a child of S.

2. To determine if it is *possible* for some specific fact to be true in a problem situation, the system adds that fact to the situation. If no contradiction results, the system concludes that the fact is possible.

   Possibility queries about mixed statements are expanded in a way similar to necessity queries about mixed statements. The difference is that each specific statement in the conjunct obtained is added to the problem situation. If a contradiction occurs from any of these additions, the query is answered negatively. For example, to answer the question, "Is it possible that A, B, and C are the only children of S?" the system adds $child(S, A)$, etc. to the problem situation. Then it does a find-all query to see if there are any other individuals in the collection referred to by the mixed statement. Continuing the example, the system finds all the children of S. If there are any children other than A, B, and C, the query is answered "no." Otherwise, it is answered "yes."

3. To find all the individuals that stand in some relation to a specific individual, the system searches the problem situation looking for all facts relating individuals to the specific individual.

4. A find-the query can only ask for the value of a functional expression like "find-the $siblings(S)$." These questions are answered by retrieving from the problem situation the image of an individual under the `function`. For example, the query, "find-the $siblings(S)$," is answered by retrieving from the problem situation the image of S under SIBLINGS (the representation of $siblings$).

   As noted above, the representation design system can transform find-all queries into find-the queries by reformulating a problem. The reformulation has the

effect of adding a new kind of individual to the problem. For example. the query "find-all x : $sibling(S, x)$" is transformed into "find-the $siblings(S)$," where the range elements of *siblings* are sets of individuals. One advantage of reformulating in terms of a find-the query is that sets can be marked to indicate that all members are known. This makes answering the second part of a find-all query much easier.

1.4 Soundness of Representation Design

One question that arises about the representation design process concerns how much faith we should place in the answers that we get with representations designed by the system. This is the question of whether or not the representation design process is sound.

I have shown that if the representation design system produces a fully constrained representation and that representation halts while building a problem situation, then the answers produced with that representation are always answers to the original problem.

There are two kinds of lemmas that must be proved to obtain the soundness result. First, since representation design changes problem statements, we must prove that all transformations done on the predicate calculus problem statement are sound. Of particular interest is the process of introduction because it adds new concepts to a problem. Chapter 5 gives a soundness condition for introductions. I have shown that all introductions in the system meet this condition and, therefore, the introduction process as a whole is sound.

The purpose of the second kind of lemma is to show that the process of capturing constraints in a representation is sound. In particular, we must show that when the constraints of a set of general statements Φ are captured by a representation, the following is the case. If a set of specific facts Ψ is added to that representation, then

the problem situation created will contain every specific fact that follows from the union of Φ and Ψ. Since the allowable queries can be answered given the specific facts that follow from a problem. this result is sufficient to ensure correct answers.

1.5 Scope and Limitations

The representation design system is knowledge based and. as with other such systems. it is very difficult to formally characterize its scope of applicability. This section, therefore, attempts to converge on this issue from two perspectives. First, I try to provide a characterization of the class of problems for which it is theoretically possible to design representations of the sort used by current system. I also give examples of two kinds of problems that are outside of this class. Second, I give evidence indicating that the current implementation provides the framework and much of the knowledge required to handle most analytical reasoning problems. More complete discussions of these issues can be found in chapter 8.

1.5.1 The Class of Problems

This section outlines attempts to formally characterize the class of problems that my methodology can handle and provides an example of a problem outside of this class.

We can characterize the way in which specialized representations are used to solve problems by saying that they build a model of the specific situation described in the problem and then inspect that model to answer questions. [4] In this view, we should only expect to be able to solve a problem if it has at least one finite model. If there are no finite models, then when the problem situation is expressed in the specialized representation, there will be an attempt to create an infinite structure.

[4]Chapter 8 provides a definition of the semantics of the representations that the system designs. There, I show how the structures built by representations of the sort that my system designs can be viewed as models. For example, situations built with the representation designed for the FAMILIES problem can be viewed as models of that problem.

This notion of class appears to mesh quite nicely with ordinary intuitions about the way people solve analytical reasoning problems: the process is often described as constructing a "model" of a problem situation, then inspecting it to solve the problem. Some difficulties with this characterization are discussed in chapter 8. Despite these difficulties, I believe the intuition here is sound and helpful in understanding the theoretical limits of the current representation design method.

The finite model characterization is helpful in understanding when problems are beyond the scope of this methodology. An example of a problem outside the scope will help make this clear. Any problem that requires reasoning by induction is outside the scope. Induction problems violate the syntactic restriction that queries may not ask about general facts (facts that do not contain individuals). More importantly, induction theorems are questions about the properties of infinite well orderings. All models of such orderings are infinite, so such problems are outside the scope of my methods.

1.5.2 Coverage

Section 1.3 provided an informal characterization of the class of analytical reasoning problems. I believe that the current system can easily be extended to provide reasonably high coverage of analytical reasoning problems. I characterize the coverage of the existing knowledge by answering the following two questions: First, how well does the existing library cover a large body of analytical reasoning problems? Second, how much overlap is there in the knowledge used on different problems?

There is evidence to suggest that the current structure library population is very close to a sufficient collection for designing representations for most analytical reasoning problems. The current system was designed after a survey of approximately two hundred analytical reasoning problems. From these, I compiled a set of twenty representative problems. From the representative set, I chose three problems that I

found among the most difficult to solve. I call these the *paradigm* problems. I then studied the representations that I and two other people used in solving the paradigm problems. The current structure library population is the result of that study.

Examination of the other seventeen representative problems showed that the existing structure library was sufficient: the problems did not use any additional structures.

This research would not be interesting if each problem solved used a disjoint subset of the knowledge. This has proven to be far from the case.

The current representation design system generates representations for the three paradigm problems and some variations (eight problems total). One of the paradigm problems is the FAMILIES problem. The others are shown in figure 1.8 and figure 1.9.

Eight law professors are housed in a single wing of a building. The wing contains ten offices, numbered 1 to 10, in that order; each professor is assigned to a different office, and two offices are left empty for use as meeting rooms. The professors are named Boswell, Dyer, Garrett, Harrelson, Kranepool, Ryan, Taylor, and Weis.
Dyer is four offices away from Kranepool.
There is one empty office and one occupied office between Taylor and Harrelson.
Ryan is in an office next to Boswell.
Dyer is in an office next to Garrett.
Kranepool is between an occupied office and an empty office.
Weis is in office 2.
Garrett is in office 7.
Who is in office 4?
Which offices are unoccupied?
Is Ryan, Dyer, Garrett, Taylor a possible sequence of offices?

Figure 1.8: The PROFESSORS problem

Even though the paradigm problems appear quite different, the representations generated for them only use structures from the existing library. Also there is a high degree of overlap in the introduction knowledge used in generating representations for these problems. All the rules used in designing a representation WAITERS were also used in the other two problems and there was a 90% overlap between FAMILIES and PROFESSORS. Also each introduction was performed more than once in each

A restaurant employs eight waiters, D, E, F, G, H, I, J, and K, each of whom works four days a week. The restaurant is open every day except Monday. On Friday and Saturday, a staff of six waiters is needed. On all other days when the restaurant is open, a staff of five is needed.
D cannot work on Tuesday or Thursday.
E cannot work on Wednesday.
G cannot work on Thursday or Saturday.
H cannot work on Friday.
J cannot work on Tuesday or Sunday.
K cannot work on Wednesday or Friday.

Is D,E,F,I,K a possible staff of waiters for a Tuesday?
Is Tuesday, Wednesday, Thursday, Sunday a possible work week for G?

Figure 1.9: The WAITERS problem

problem, often three or four times.

1.6 Related Work

This section highlights the differences between the research reported in this thesis and previous work in several areas. A more thorough treatment can be found in Chapter 9.

1.6.1 Systems Solving Word Problems

One might expect there to be an interesting relationship between my system and previous systems that solve word problems. However, most work in this area has been concerned with translating an English problem statement to a preestablished target representation. For example, the object of STUDENT [Bobrow68] was to translate a high school level algebra word problem into a system of algebraic equations. Also, Bobrow was primarily concerned with natural language problems and I have not addressed this.

1.6.2 Automatic Programming

In most previous work in this area, automatic programming systems have performed tasks such as algorithm design, data structure selection, and optimization (of algorithms and data structures) within a fixed representation which is chosen by a person prior to the point where the system gets involved. By contrast, my system is concerned with those earlier steps in the problem solving process during which a representation is designed.

Many efforts in automatic programming have generated a program from a formal specification including axioms describing the desired program and the programming language operations available to implement that program. As many researchers working in this area have pointed out, the formulation of these axioms can have a dramatic effect on whether this approach succeeds. Searching for better formulations of a set of axioms is a large part of what my research is about.

1.6.3 Good Representation

As I have already stated, the principle difference between my research and previous efforts to understand what makes a representation good that they were concerned with recognizing the properties of good representations and my research is about generating such representations prospectively.

1.6.4 Problem Reformulation

In some ways my work can be seen as an extension of [Korf80]. Korf was concerned with characterizing a space of possible representations and types of transformations on representations. My work is concerned with *how to choose the right transformations to do to arrive at a good problem representation.* I have identified some of the essential properties of representations and given a method to design representations with those

properties.

Korf (and Amarel) viewed problem solving as state space search and observed that changes in representation (i.e., the description of a problem state) affect the size of the space. The focus of my work has been explaining *how* representations do this and how to design representations that yield smaller search spaces. The claim is that when a representation captures more constraints in its structure and behavior the problem solving space is reduced.

1.6.5 Specialized Reasoners

There is a body of work whose concern is to develop a framework in which multiple specialized reasoners can be used together in problem solving. Each reasoner is specialized to perform certain inferences efficiently and together a collection of such mechanisms can be used to accelerate a general problem solver (usually a theorem prover). My research complements this work because my system can be viewed as a specialized reasoning system designer.

1.7 Reader's Guide

The next chapter explains the three evolving descriptions that the representation design system maintains throughout the design process. One can think of these as descriptions of the problem statement, the representation being designed for that problem, and the relationship between the first two descriptions. The chapter also describes a language for defining and instantiating abstract data types. This is used by a person defining library structures and by the system in introducing new representations.

Chapter 3 describes how the system derives a description of a problem's initial representation. This is done in three steps:

1. The primitive concepts are identified from a problem statement.

2. Irrelevance is removed from the problem.

3. The representation of each relevant primitive is added to the representation description.

Chapter 4 discusses the classification process and the knowledge used in the process. Then, Chapter 5 discusses the concept introduction process, explains how classification and introduction are integrated as extended classification, and gives detailed examples from extended classification in the FAMILIES problem. Chapter 6 discusses the operationalization process and gives several examples, again from the FAMILIES problem.

Chapter 7 details the way representations are implemented. First, it explains the equality system relied on by representations and a mechanism for creating anonymous individuals. It explains the behavior of the library structures `individual`, `relation`, `function`, and `set` and how they integrate with the equality system. An example is given of the system creating a specialized representation.

Chapter 8 discussed several loosely related issues involved in developing a more formal theory of representation design. It gives a semantics for the representations the system design, outlines a proof of the soundness of representation design, shows how to extend the existing query mechanisms so that they are complete, and discusses the system's coverage of analytical reasoning problems in more detail.

Finally, Chapter 9 discusses related work and Chapter 10 provides a summary and a discussion of future work.

Chapter 2

Descriptions Used in Representation Design

The next four chapters discuss the representation design process in more detail. In preparation for this, this chapter explains the descriptions that the representation design system maintains throughout the design process. One can think of these as three evolving descriptions: one of the problem statement, one of the representation being designed for that problem, and the third a description of the relationship between the first two.

The *problem statement language* (PSL) is a sorted first order logic. The *representation description language* (RDL) is a collection of constructs for defining abstract data types implementing representations and prototypes.

2.1 The Problem Statement Language

Problems consist of a collection of statements and a collection of queries. The statements include sort declarations (e.g., sort$(N, FAMILY\text{-}MEMBER)$) and a collection of statements in the logic. The logic also contains the distinguished relations $\in$ and $=$. For convenience, appropriate syntax is included so that terms can denote

extensional and intentional sets. [1] Extensional sets are denoted by terms of the form, $\{t_1, \ldots, t_n\}$, where each of the t_i are terms. Intentional sets are denoted by terms of the form, $\{x \mid \phi\}$, where ϕ is a formula in which x occurs free.

The query language allows necessity, possibility, find-all, and find-the queries. Necessity queries, written $\Box\phi?$ (where ϕ is a specific statement in the description language), ask if a given fact is true in a problem situation. Possibility queries, written $\Diamond\phi?$, ask if a fact is consistent with a problem situation. Again ϕ must be a specific statement. Find-all queries, written find-all $x : \phi$ (where x is free in ϕ), ask for all individuals in a problem that stand in some relation to a specific individual. The FAMILIES problem query is an example of this type,

find-all $x : sibling(S, x)$.

In addition, an answer to a find-all query states whether the individuals found are *all* the individuals for which ϕ is true, or just all the individuals that could be found given the problem statement. Find-the queries are written find-the τ, where τ is a function application term in the problem description language. These queries ask for the individual that the function application denotes in a problem situation.

2.2 A Language for Defining Representations

As mentioned, representations are implemented as abstract data types (ADTs). The representation design system has a language for defining representations as ADTs. It is used in two ways: by a person to define structure prototypes and by the representation design system to define new representations. Library prototypes are implemented as parameterized ADTs, for example, the library structure `relation` is parameterized so that representations can be defined with different arities and over different collections of individuals.

The representation design system uses this language to define representations in

[1]These are not formally necessary.

terms of library prototypes. I defines two kinds of representations: representations of concepts and representations of collections. Concept representations are defined by instantiating library structures, for example, MARRIED is defined by instantiating `relation`. Collection representations are defined as subtypes of library structures. Individuals of a collection are represented as instances of the subtype for that collection. For example, the collection *FAMILY-MEMBER* is defined as a subtype of `individual` and specific family members are represented with instances of FAMILY-MEMBER.

Data types are defined with the **deftype** construct which has two forms. The first is used to define library prototypes. Its form is:[2]

```
deftype type-name (parameter-list) is
        structure [declaration-list]
        procs [proc-list]
```

Type-name is a symbol that names the type being defined. **Parameter-list** is a list of formal parameters that are bound to particular types and constants when instances or subtypes of **type-name** are defined. The second line in the definition describes the data structures of the type. The **declaration-list** is a list of instance definitions. Thus new types are defined with instances of existing types as components. The **proc-list** is a list of definitions for the procedures that operate on the components to provide the abstract operations of the new type. For example, here is part of the definition of the library type `relation` (some of the procedures have been left out):

```
deftype relation (dom1:type . other-doms: list(types))
        structure [relation-list: list(tuple(dom1 . other-doms))]
        procs [defproc add-relationship(rel: tuple(dom1 . other-doms))
                        relation-list = rel . relation-list
                  defproc related?(rel: tuple(dom1 . other-doms))
                        member(rel,relation-list)]
```

[2]The syntax used here is a stylized version of the actual syntax used in the implementation. The actual syntax is similar to LISP flavor syntax.

The type **relation** takes as parameters a list of n domains and produces a representation of an n-ary relation. For example, this type is instantiated to define MARRIED. The domains in the parameter list are the names of types representing collections. The data structure part of **relation** is a list of n-tuples. List is another type defined in this language, the operator "." is the consing operation defined by the list data type and member is a predicate defined by that data type. Representations defined in terms of **relation** store n-tuples of individuals that stand in a relation in a particular problem situation. For example, when creating a problem situation for the FAMILIES problem, new pairs like $< N, P >$ get added to the relation list inside MARRIED. This is done by executing the procedure add-relationship with $< N, P >$ as its argument.

The second form of deftype allows new types to be defined by fixing formal parameters of existing types:

deftype **new-name** is **type-name(parameters)**.

Type-name must be the name of an existing type; **Parameters** is a (possibly partial) list of actual parameters to bind to the formal parameters of **type-name**. **New-type** is an abbreviation for the type **type-name(parameters)**. The internal components of the new type have the restricted types given by the actual parameters. For example, consider:

BIN-REL-ON-FAMS: **relation**(FAMILY-MEMBER,FAMILY-MEMBER).

BIN-REL-ON-FAMS can be thought of as a restricted prototype for defining representations of binary relations over FAMILY-MEMBER, i.e., $FAMILY\text{-}MEMBER \times FAMILY\text{-}MEMBER$.

The RDL has a construct for defining representations of collections. It has the form:

COLLECTION **new-type** OF **type-name(parameter)**.

This has the effect of to defining a subtype for the collection. For example, a representation for the collection of all brother sets is defined as

COLLECTION BROTHER-SET OF **set**(FAMILY-MEMBER).

This actually defines a subtype of **set**. Note that the meaning of this statement is quite different from the second form of **deftype**. For example, it would be incorrect to define BROTHER-SET as

deftype BROTHER-SET is **set**(FAMILY-MEMBER)

because not all sets of family members are brother sets.

The RDL also has a construct for defining representations of concepts. It has the form:

instance-name: type-name(parameters).

As an example, MARRIED is defined as

MARRIED: **relation**(FAMILY-MEMBER,FAMILY-MEMBER).

2.2.1 Maintaining The Correspondence

One part of representation design is designing representations for existing problem concepts and collections. The other part involves introducing new concepts and collections (reformulation). A correspondence is maintained between concepts and representations as new concepts are introduced. A new concept is given a definition in the logic in terms of an existing concept. Section 1.2.8 explained how this definition is used as a rewrite rule. Logical definitions of new concepts have another use. The system uses them to define new collections. For instance, an earlier example described the introduction of brother sets. This is actually done by introducing a function, *brothers*, from family members to their sets of brothers. It is introduced with the statement,

$$\forall x \forall y [x \in brothers(y) \Leftrightarrow brother(y, x)].$$

This statement defines a new collection, call it *brother-set*, whose individuals

are denoted by *brothers*(y), i.e., as the variable y ranges over individuals in the *FAMILY-MEMBER* collection, *brothers*(y) ranges over the individuals in the *brother-set* collection. A name is declared for collections defined in this way with the **sort** statement which indicates the collection that an individual belongs to by giving a term in the PSL denoting that sort of individual. For example, a name is declared for the collection defined by above logical definition as follows:

sort(*brothers*(x), *brother-set*).

Then to maintain a correspondence, a representation is defined for the collection *brother-set* as

COLLECTION BROTHER-SET OF **set**(FAMILY-MEMBER).

This defines the representation of a typical member of the *brother-set* collection. Each instance of BROTHER-SET is a **set** of FAMILY-MEMBERS.

Finally, BROTHERS is defined as

BROTHERS: **function**(FAMILY-MEMBER,BROTHER-SET)

The introduction of new collections and their representations, as in the above example, allows the representation design system to explore constraints between individuals, i.e., between brother sets. For example, one question that gets asked about *brother-set* is whether all of its members are sets of the same size.

Logical definitions introducing new relations also define new collections. For example, the following statement is the definition of a relation *is-couple*:

$\forall x \forall y[\textit{is-couple}(\{x,y\}) \Leftrightarrow \textit{married}(x,y)]$.

The collection defined by a statement introducing a new relation is the sub-collection of the relation's domain for which the relation is true. For instance, the statement above defines a collection which I call *married-couple*. Its individuals are those doubleton sets (of family members) for which *is-couple* is true. The collection *married-couple* is declared as

$\text{sort}(x \mid \textit{is-couple}(x), \textit{married-couple})$.

This statement is read. "all individuals x such that *is-couple*(x) is true are *married-couples*."

The logical definitions that we have seen so far define new concepts by giving an equivalence between atomic formulas. The system can determine which concept is being defined because only one concept in the definition may be new, hence, it does not care which side of the by-conditional the new concept appears on. However, for clarity of presentation, we will always write logical definitions with the new concept mentioned on the left hand side.

Logical definitions of new concepts can be more complicated than an equivalence between two atomic formulas. Either side of the by-conditional can be a conjunction of atomic formulas. When the left hand side is a conjunction, it is treated as a rewrite rule that replaces the atomic formula on the right by a conjunction of atomic formulas. For example, the definition

$$\forall x \forall y [\textit{couple}(x) = \textit{couple}(y) \wedge x \neq y \Leftrightarrow \textit{married}(x,y)]$$

is treated as a rewrite rule that replaces atomic formulas of the form *married*(x, y) by the conjunction on the left hand side. When this rule is used to rewrite the statement *married*(N, P), the result is

$$\textit{couple}(N) = \textit{couple}(P) \wedge N \neq P.$$

When the right hand side of a definition is a conjunction, it is treated as a conditional rewrite rule. For example, consider the following definition of the function *non-empty-brothers*:

$$\forall x \forall y [x = \textit{non-empty-brothers}(y) \Leftrightarrow x = \textit{brothers}(y) \wedge \textit{brothers}(y) \neq \emptyset].$$

This is interpreted as the following rewrite rule:

> "If a statement S contains a term of the form *brothers*(y) and it can be shown from the structure of S that the term denotes a non-empty brother set, then replace the term by *non-empty-brothers*(y)."

The intention behind the phrase, "can be shown from the structure of S" is that

the representation design system will only attempt to show a condition by a syntactic analysis of S. For example, the above rewrite rule would be applied to the statement, $A \in brothers(B)$, (where A and B are constants) because the statement unconditionally asserts that B's brother set is non-empty.

Clearly, checking conditions on syntactic grounds, as in the above example, is not complete in the sense that it will not always be possible to design syntactic tests that enable application of a conditional rewrite in all appropriate situations. Therefore, definitions that are interpreted as conditional rewrites are considered to partially define collections. Representations are designed based on the best information that can be deduced about a collection by syntactic methods. Thus the representation design system trades off the inherent undecidability of general questions about collection for some inefficiency in the representations designed.

Chapter 3

Deriving an Initial Representation

This chapter describes how a description of the initial representation is derived from a problem statement. The processes described here result in a fully expressive representation that is the starting point from which a fully constrained and maximally specialized representation is designed.

The goal in deriving a description of the initial representation is to identify a set of concepts that the specialized representation can be designed in terms of. This set, which we will call the set of *represented* concepts, should be sufficient to express all the constraints of a problem class. In addition, it is desirable that it only include concepts that are necessary for expressing the constraints of a problem class.

We first discuss the sufficient condition. A *primitive* concept is one that is not defined in terms of other concepts. To be sufficient, the set of represented concepts should include all the primitive concepts that are relevant to solving the problem. Primitive concepts are required because a representation can not be designed to capture all the relevant constraints without them. Suppose a relevant primitive concept is left out. Since this concept is relevant, it must have constraints defined on it that are used in solving the problem. But since the concept is not represented and not definable in terms of represented concepts, a representation can not be designed to capture those constraints.

Relevant concepts with restrictions [1] on them are included in the representation description even if they are not primitive. For example, the statement,

$$\forall x \neg brother(M, x),$$

causes *brother* to appear in the set of represented concepts for the FAMILIES problem even though it is not a primitive.

These concepts are included in recognition of a fact about the representation design process: The strategy that the representation design system uses to capture restrictions is to introduce alternative concepts that transform the restrictions into specific constraints. For example, to capture the constraint of the statement above, the concept *brothers* is introduced and the statement is transformed into the specific statement,

$$brothers(M) = \emptyset,$$

where *brothers* is a function from an individual to his set of brothers.

Such introductions are done by extended classification and introduction. Since the only concepts that get classified are those that are represented, concepts with restrictions on them are included so that they will get classified.

It is desirable that the set of represented concepts only include primitives and concepts with relevant mixed constraints on them. The basic reason for this is economy of mechanism. Since instances will be designed for all the represented concepts, representing more concepts than are strictly necessary can cause unnecessary machinery to be designed. This will not only cause the representation design system to do more work than necessary but it will also cause extra work to be done in maintaining unnecessary constraints during problem solving.

Two types of concepts are considered unnecessary: those that are irrelevant to answering a problem's queries and those that are not primitive and do not have relevant

[1] Recall that a restriction is a mixed statement that restricts the collections of individuals that stand in some relation to a specific individual.

restrictions on them.

When a defined concept that does not have restrictions on it is represented, redundant general constraints will be maintained by the resultant representation. For example, suppose the concepts *sibling* and *male* are represented. Since *brother* can be defined in terms of these, designing a separate representation for *brother* causes its constraints to be captured redundantly. As an illustration, note that if BROTHER is included in a fully constrained representation the constraint of the statement

$$\forall x \forall y [brother(x,y) \Leftrightarrow sibling(x,y) \land male(y)]$$

will also be captured. In addition, BROTHER will be designed to capture the constraints on *brother*. For example, BROTHER will capture the irreflexivity of *brother*. Furthermore, SIBLING will capture the irreflexivity of *sibling*. But since $brother(x,y) \Rightarrow sibling(x,y)$, the irreflexivity of *brother* is captured twice.

We now proceed to discuss the following three steps which identify the collection of represented concepts:

1. An initial set of primitive problem concepts is identified. As we will see, when there is missing definitional information, it is identified in this step.

2. Concepts that are irrelevant are removed from the problem statement and from the set of represented concepts. The procedure that does this is sound but not complete: Only irrelevant concepts will be eliminated but it is possible for it to miss irrelevant concepts. Irrelevance is identified after the primitive concepts because the additional definitional knowledge acquired in step one can change what is relevant.

3. Concepts that have explicit restrictions on them are identified and added to the set. Concepts that have implicit restrictions on them (i.e., those for which a restrictions follows from the problem but is not stated initially) are identified later in representation design.

A description of the initial representation is constructed by defining representations for all of the concepts identified in the three steps above.

The next two sections of this chapter describe how steps one and two above are performed. The last section discusses the derivation of representations for the concepts in the initial description.

3.1 Identifying The Primitive Concepts of a Problem

A concept is defined in terms of others when it appears alone on one side of a by-conditional. For example, the following is a definition of *brother*:

$$\forall x \forall y[brother(x,y) \Leftrightarrow sibling(x,y) \land male(y)].$$

Identifying primitive concepts can be complicated by the fact that real problems are incomplete. An incomplete problem is one that does not supply sufficient information to answer its queries. When a problem is incomplete, we can not determine whether a concept is primitive since its definition may simply be missing. In recognition of this possibility, the representation design system asks the user whether he/she would like to define any of the concepts that it thinks are primitive. Acquiring definitions for such concepts often uncovers additional concepts that did not appear in the problem statement. The process of prompting for new definitions continues until the user declines to further define any of the concepts that the representation design system believes to be primitive.

Acquiring definitional information is one of the techniques that the representation design system uses to try to complete problem statements. It is clear that this technique is at the mercy of the user. There is nothing the system can do if he/she declines to provide a definition that is required to complete the problem statement.

3.1.1 Determining that a Concept is Defined

A very simple syntactic strategy is used to determine if a problem statement contains a definition for a concept. A concept is considered defined when an expression denoting it is found at the top level of one side of an if and only if statement. For example, the statement,

$\forall x \forall y[son(x,y) \Leftrightarrow child(x,y) \wedge male(y)]$,

is treated as the definition of "son." If a statement defines two concepts solely in terms of each other, then one is arbitrarily selected as being defined by the statement.

Concepts are never considered to be defined in terms of themselves. Hence, the statement,

$\forall x \forall y[married(x,y) \Leftrightarrow married(y,x)]$,

is not considered a definition of *married.*

When the user is asked for a concept definition, he/she is required to supply a by-conditional. However, a concept definition can follow from a problem statement and not be explicit. For example, a problem statement can contain necessary and sufficient conditions for a concept which are semantically equivalent and syntactically different. The representation design system will miss this situation and include such concepts as primitive. This is one way the set of represented concepts can fail to be minimal. This does not cause serious problems in the representation design process; it just causes the system to do extra work.

3.1.2 Section Summary

Throughout the rest of this chapter and the next three, we will summarize each section by showing how the processes described in it change the statement of the simplified FAMILIES problem shown in figure 3.1.

sort(P, *FAMILY-MEMBER*), sort(Q, *FAMILY-MEMBER*),
sort(R, *FAMILY-MEMBER*), sort(S, *FAMILY-MEMBER*)
$grandchild(Q, S)$
$\forall x\ child(P, x) \Leftrightarrow x = R$
$married(Q, P)$
Query: find-all x: $parent(S, x)$

Figure 3.1: A small problem about families.

The process of acquiring missing definitional knowledge expands this problem statement to the one shown in figure 3.2.

sort(P, *FAMILY-MEMBER*), sort(Q, *FAMILY-MEMBER*),
sort(R, *FAMILY-MEMBER*), sort(S, *FAMILY-MEMBER*)
$grandchild(Q, S)$
$\forall x\ child(P, x) \Leftrightarrow x = R$
$married(Q, P)$
$\forall x \forall y[grandchild(x, y) \Leftrightarrow \exists z(child(x, z) \wedge child(z, y))]$
$\forall x \forall y[child(x, y) \Leftrightarrow parent(y, x)]$
Query: find-all x: $parent(S, x)$

Figure 3.2: The example problem with definitions added.

In addition, *child* and *married* have been identified as primitives.

3.2 Eliminating Irrelevant Information

It is desirable to eliminate irrelevant information before a representation is designed because we do not want representations to capture irrelevant constraints. However, part of my approach is design representations for problems before solving them. In light of this, I have developed an irrelevance filter that can be used before a problem is solved. However, it does not always eliminate all irrelevance.

The filter is guaranteed not to eliminate relevant information as long as it is run on a complete problem statement. However, the system runs the filter right after primitive concepts are identified. Subsequent steps in representation design may further complete a problem statement and, therefore, running the filter at this point

may cause relevant information to be eliminated. In light of this, the system allows for the possibility that information filtered out of a problem statement may later be resurrected.

The discussion that follows begins by defining the class of irrelevance that is usefully eliminated in representation design. Then the filter is described and its soundness proved. Next, I discuss the kinds of irrelevant information that the filter misses.

3.2.1 Strong and Weak Irrelevance

We define two classes of irrelevance: strong and weak. A fact is *strongly irrelevant* to a problem P if it can not be used to derive the answer for any problem in P's class. A fact is *weakly irrelevant* to a problem P if there is at least one problem in P's class whose answer can be derived without the fact. We will also say that a fact that is not strongly irrelevant is *weakly relevant* because it is relevant to at least one problem in the class.

Since representations are designed for problem classes, weak irrelevance should not be eliminated and, therefore, our interest is in identifying strong irrelevance.

A direct method for identifying strong irrelevance would be to generate all the problems in the input problem's class, solve them, and then identify statements that were not used in any of these proofs. Clearly this method is impractical.

The method that the representation design system actually uses abstracts away from the original problem, retaining only its propositional structure. It then identifies a subset of the statements that can not appear in any proof in the problem class based on the connectivity of the problem's propositional approximation.

I will show that the facts that can not appear in any propositional proof are strongly irrelevant. I will also show a counter example to the converse, illustrating that the method does not eliminate all strong irrelevance.

3.2.2 An Approximate Strong Irrelevance Filter

The filter begins by constructing a propositional version of the problem by replacing the atomic formulas in each problem statement by propositional symbols, retaining the propositional structure of the statement. Then this version of the problem is converted to clause form. Next, the propositional form of the problem query is negated, converted to clause form, and added to the clause set. Finally, a *connection graph* [Kowalski75] is used to determine which clauses are irrelevant.

The nodes of the connection graph correspond to the propositional clauses and are linked together when they contain literals that resolve. [2] Any clause that contains a literal that is not connected to any other clause in the graph is strongly irrelevant. Note that disconnected clauses are called *impure* in the connection graph literature. Strong irrelevance is then eliminated from the original (non-clausal) problem statement by identifying the statements from which the strongly irrelevant clauses were derived and removing those clauses. This problem statement is what is used in the representation design process.

Deriving the Propositional Problem

The first step is to extract the propositional structure of the problem. This is done by substituting a predicate's name — as a propositional symbol — for any atomic formula referencing it. For example, $P(x)$ becomes P; $\neg P(x)$ becomes $\neg P$. In doing this, equalities are replaced by the predicate symbol $EQUAL$ and membership expressions are replaced by the symbol $MEMBER$. As this is done, a record is kept of the original statement each propositional statement is derived from. Also, sort statements are removed in this step.

Figure 3.3 gives an example that we will use to illustrate the methods explained in

[2] In a connection graph built from a set of first order clauses, the links between nodes are labeled with the most general unifier of the two literals that resolve.

this section. This is our running example problem with statements added giving the connection between *married* and *child*. The result of transforming this problem as described so far is shown in figure 3.4. Note that the query has temporarily been removed.

sort($P, FAMILY\text{-}MEMBER$), sort($Q, FAMILY\text{-}MEMBER$),
sort($R, FAMILY\text{-}MEMBER$), sort($S, FAMILY\text{-}MEMBER$)
$grandchild(Q, S)$
$\forall x\ child(P, x) \Leftrightarrow x = R$
$married(Q, P)$
$\forall x \forall y \forall c[child(x, c) \wedge child(y, c) \wedge x \neq y \Rightarrow married(x, y)]$
$\forall x \forall y \forall c[married(x, y) \wedge child(x, c) \Rightarrow child(y, c)]$
$\forall x \forall y[grandchild(x, y) \Leftrightarrow \exists z(child(x, z) \wedge child(z, y))]$
$\forall x \forall y[child(x, y) \Leftrightarrow parent(y, x)]$
Query: find-all x: $parent(S, x)$

Figure 3.3: The example problem with the relationship between *child* and *married* added.

grandchild
child $\Leftrightarrow$ *EQUAL*
married
child $\wedge$ *child* $\wedge \neg$*EQUAL* $\Rightarrow$ *married*
married $\wedge$ *child* $\Rightarrow$ *child*
grandchild $\Leftrightarrow$ *child* $\wedge$ *child*
child $\Leftrightarrow$ *parent*

Figure 3.4: The propositional form of the FAMILIES problem with acquired definitions added.

The next step in this process is to extract the fact from the problem query, negate it, convert it to propositional form, and add it to the propositional problem statement. The procedures for extracting a fact from each of the query types are as follows:

1. For queries of the form $\Box\ \phi$, the fact extracted is ϕ. This fact will be negated and added to the problem statement. This is exactly what is standardly done in building a connection graph: the statement to be proved is negated.

2. To understand how a fact is extracted from a query of the form $\Diamond\ \phi$, recall that these queries are answered by adding ϕ to a problem situation and looking for

a contradiction. This is equivalent to trying to prove $\Box\neg\phi$ and reporting that ϕ is possible unless the proof succeeds. Therefore, ϕ should be added to the connection graph and, since query facts get negated, $\neg\phi$ is the fact extracted from this type of query.

3. For queries of the form find-all $x : \phi$, the fact extracted is also ϕ because in order to answer queries of this type the system must construct proofs of ϕ for all individuals that it can.

4. The system can only answer find-the queries when the functions mentioned in them are defined in terms of relations. For example, the query, "find the mother of A" can be answered because *mother-of* is defined in terms of the relations *parent* and *female*. The system derives a fact for a query of the form find-the τ by using definitions to expand the formula $x = \tau$ into one that mentions only relations. For example, the system uses the definition of *mother-of* to expand the formula $x = mother\text{-}of(A)$ into

   $\forall y \; parent(y, A) \land female(y).$

   This is the fact extracted from the query, "find-the *mother-of*(A).

Extracting the query fact from the problem in figure 3.4 adds $\neg parent$ to the propositional problem statement.

The next step is to convert the propositional problem statement to clause form, still keeping track of which original problem statement each clause came from.

Before the graph is built, the literals $EQUAL$ and $\neg EQUAL$ are deleted from the clauses. This has the effect of assuming that no clause should be considered impure on the basis of an equality or inequality; this in turn is equivalent to assuming that (in)equalities are always relevant. It is correct to assume that equalities are relevant because a complete problem statement must contain inequalities between the distinct individuals in the problem, for example, $N \neq M, N \neq P$. These inequalities connect

to the equalities in other problem statements.

Also since there can be unnamed (existentially quantified) individuals in a problem, two unnamed individuals could be equal. Such equalities will connect to inequalities in problem statements and, therefore, it is correct to assume that inequalities are relevant.

Building the Connection Graph

This process begins by dividing the clauses into two sets. The *query clauses* are those derived from the query facts. The *descriptive clauses* are those derived from the descriptive problem statements.

When the graph is complete, all descriptive clauses that do not appear in the graph are marked strongly irrelevant. Then the graph is checked for impure clauses. These are marked strongly irrelevant and deleted from the graph. These deletions may cause other nodes to become impure, in which case these are marked and deleted. This process continues until no more impure clauses can be found. [3]

As an example, consider figure 3.5. This is the problem of figure 3.3 with the following strongly irrelevant statements added:

$\forall x[child(P, x) \Rightarrow haircolor(x, RED)]$
$\forall x[haircolor(x, RED) \Rightarrow pidgeon\text{-}toed(x)]$

The propositional form of the problem is shown in figure 3.6; The clausal propositional form is shown in figure 3.7, while figure 3.8 shows the connection graph built from the clausal propositional form.

The node enclosed in a rectangle in that figure is impure. This node corresponds to the propositional clause, $\neg haircolor \vee pidgeontoed$, which, in turn, corresponds to the statement,

$\forall x[haircolor(x, RED) \Rightarrow pidgeontoed(x)]$,

[3] This is a standard process, see [Kowalski75].

$sort(P, FAMILY\text{-}MEMBER)$, $sort(Q, FAMILY\text{-}MEMBER)$,
$sort(R, FAMILY\text{-}MEMBER)$, $sort(S, FAMILY\text{-}MEMBER)$
$grandchild(Q, S)$
$\forall x\ child(P, x) \Leftrightarrow x = R$
$married(Q, P)$
$\forall x \forall y \forall c[child(x, c) \wedge child(y, c) \wedge x \neq y \Rightarrow married(x, y)]$
$\forall x \forall y \forall c[married(x, y) \wedge child(x, c) \Rightarrow child(y, c)]$
$\forall x \forall y[grandchild(x, y) \Rightarrow \exists z(child(x, z) \wedge child(z, y))]$
$\forall x \forall y[child(x, y) \Leftrightarrow parent(y, x)]$ $\forall x[child(P, x) \Rightarrow haircolor(x, RED)]$
$\forall x[haircolor(x, RED) \Rightarrow pidgeon\text{-}toed(x)]$
Query: find-any x: $parent(S, x)$

Figure 3.5: A simple problem statement used to illustrate the irrelevance filter.

$grandchild$
$child \Leftrightarrow EQUAL$
$married$
$child \wedge child \wedge \neg EQUAL \Rightarrow married$
$married \wedge child \Rightarrow child$
$grandchild \Leftrightarrow child \wedge child$
$child \Leftrightarrow parent$
$child \Rightarrow haircolor$
$haircolor \Rightarrow pidgeon\text{-}toed$
$\neg parent$

Figure 3.6: The propositional form of the example problem

in the original problem.

This node is deleted from the graph and the above statement is deleted from the problem statement. When this node is deleted, another node becomes impure, corresponding to the problem statement

$$\forall x[child(P, x) \Rightarrow haircolor(x, RED)].$$

Again the node and corresponding statement are deleted.

Formal Properties of the Irrelevance Filter

It is widely known that impure clauses can not be used in resolution proofs because they can not be used to derive the empty clause. To see that impure clauses in the

grandchild
$\neg$*child*, *child*
married
$\neg$*child* $\vee$ *married*

$\neg$*married* $\vee$ $\neg$*child* $\vee$ *child*
$\neg$*grandchild* $\vee$ *child*, $\neg$*child* $\vee$ *grandchild*
child $\Leftrightarrow$ *parent* $\neg$*child* $\vee$ *parent*, $\neg$*parent* $\vee$ *child*
$\neg$*child* $\vee$ *haircolor*
$\neg$*haircolor* $\vee$ *pidgeontoed*
$\neg$*parent*

Figure 3.7: The clausal propositional form of the example problem

propositional connection graph are strongly irrelevant, consider one of the disconnected literals in an arbitrary impure clause and notice that there can be no clause in the clausal form of the first order problem containing the negation of that literal. Otherwise, by construction, the negation of the propositional form would appear in the propositional clausal version of the problem, and the clause we assumed to be impure would be pure. But this means that there can be no proof in the clausal form of the original problem involving the impure clause. By the definition of problem class, there can be no clause in any problem in the class containing the negation of the literal and therefore it is strongly irrelevant.

This procedure does not identify all strong irrelevance in a problem. In fact, doing so can be very difficult. Consider the following example which capitalizes on the fact that function symbols have been abstracted away,

$\forall x[R(g(x)) \Rightarrow Q(x)]$
$\forall x[P(x) \Rightarrow R(f(x))]$.

It would appear that the second statement is strongly irrelevant to proving that some individual has the property Q. However, the irrelevance filter will fail to eliminate this statement. Notice, that in fact, the statement could be relevant in a problem instance where the ranges of f and g intersect. On the other hand, suppose the problem statement implies,

$\forall x \forall y[f(x) \neq g(y)]$.

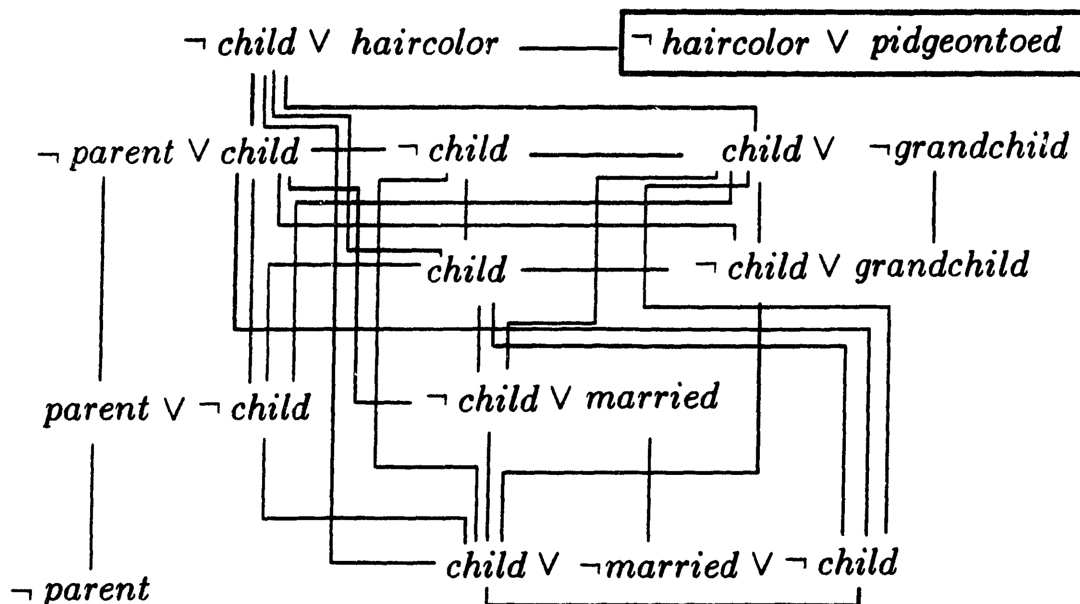

Figure 3.8: The connection graph for the example problem with irrelevance added.

It seems reasonable for the filter to leave the statement,

$$\forall x[P(x) \Rightarrow R(f(x))],$$

in the above example. However, consider a more blatant problem with the filter which results from the appearance of definitions in a problem. Suppose a problem contains the definition,

$$\forall x \forall y[sibling(x,y) \Leftrightarrow \exists p\ child(p,x) \wedge child(p,y) \wedge x \neq y],$$

mentions *child* elsewhere in the problem, but never mentions *sibling*. The propositional version of the problem will contain,

$$sibling \Leftrightarrow child \wedge child \wedge \neg EQUAL,$$

and this will be included in the graph in spite of the fact that it is strongly irrelevant. This is because *sibling* connects to itself through the definition. Concepts like *sibling* above that are irrelevant but are connected through a definition in a graph are called *definitionally irrelevant.* Definitional irrelevance can be detected using the connection graph to identify concepts that are connected to themselves through a definition, but

are otherwise strongly irrelevant. This is discussed below in section 3.2.3.

Eliminating Strong Irrelevance

Once a collection of strongly irrelevant clauses has been identified, the original problem statement is simplified to remove all mention of them. The desired effect of this process should be the same as converting the problem to clause form and then removing the clauses that mention disconnected literals. However, getting the correct effect on a collection of non-clausal statements is more complicated. For example, suppose the literal P is disconnected and consider the difference between what should be done to the two statements,

$$P \vee Q \Rightarrow R$$
$$P \wedge Q \Rightarrow R$$

The first statement should be simplified to $Q \Rightarrow R$ because either P or Q implies R. However, the second statement can be entirely thrown away because R's truth value must depend on P.

The insight used in the elimination procedure is that when some literal is disconnected, the solution to the problem will not depend on the truth value of the literal. Therefore, whether it is true or false, the problem solution will be the same. This insight translates rather directly to the following manipulation of a problem statement to eliminate irrelevance. Replace each statement $\mathcal{S}$ in the problem that contains a disconnected literal $\mathcal{L}$ with,

$$\mathcal{S}[\mathcal{L}/\mathbf{true}] \vee \mathcal{S}[\mathcal{L}/false].$$ [4]

The new statement is then simplified according to the rules in figure 3.9.

Continuing the example above, if P is irrelevant, then,

$$P \vee Q \Rightarrow R$$

becomes,

$$[false \vee Q \Rightarrow R] \vee [\mathbf{true} \vee Q \Rightarrow R],$$

$$
\begin{array}{l}
\neg true \gg false \\
\neg false \gg true \\
true \wedge P \gg P \\
false \wedge P \gg false \\
true \vee P \gg true \\
false \vee P \gg P \\
true \Rightarrow P \gg P \\
P \Rightarrow true \gg true \\
false \Rightarrow P \gg true \\
P \Rightarrow false \gg \neg P \\
P \Leftrightarrow true \gg P \\
P \Leftrightarrow false \gg \neg P
\end{array}
$$

Figure 3.9: Simplification Rules ($\gg$ means "simplifies to").

and then simplifies:

$[Q \Rightarrow R] \vee false,$

$Q \Rightarrow R.$

Similarly,

$$P \wedge Q \Rightarrow R$$

is transformed as follows,

$[false \wedge Q \Rightarrow R] \vee [true \wedge Q \Rightarrow R],$

$true \vee [Q \Rightarrow R],$

$true.$

The second statement simplifying to *true* means that when P is irrelevant, the statement no longer adds a constraint to the problem.

3.2.3 Strengthening the Filter

The irrelevance filter discussed above is a procedure that removes irrelevance based on strongly irrelevant clauses in the propositional connection graph. This section discusses using the propositional connection graph to identify strong irrelevance missed by the filter. As discussed in section 3.2.2, a definitionally irrelevant concept can get connected to itself in a graph through its definition. Concepts that potentially fit into this category can be detected from the structure of the connection graph. Once a concept has been identified from an analysis of the graph, the original problem statements mentioning the concept are checked to determine whether or not the concept

is irrelevant.

One might think that to detect definitional irrelevance it is sufficient to check the problem statement for concepts that only appear in their own definitions. For example, suppose the following definition were added to the problem in section 3.2.2:

$$\forall x \forall y[sibling(x, y) \Leftrightarrow \exists p(child(p, x) \land child(p, y) \land x \neq y)].$$

Then *sibling* would be such a concept. This technique is too dependent on the syntax of a set of statements. For example, if we break the definition of *sibling* up into two implications the technique will not detect that sibling is irrelevant.

The representation design system uses a more general technique for identifying this sort of irrelevance. The technique detects many cases where a concept is connected only to itself through equivalent necessary and sufficient conditions. The technique relies on the following ideas. A connection graph is *pure* when it contains no impure nodes. The set of clauses in which a literal appears is called the literal's *reference set*.

The technique for detecting definitional irrelevance makes use of a further subdivision of clauses into *general* and *individual* clauses. General clauses do not mention individuals, individual clauses do.

The technique works as follows. After the graph is built, it is searched for literals whose reference sets contain only general clauses. A literal whose reference set is a pure subgraph containing only general clauses may be definitionally irrelevant. For instance, consider adding the definition of *sibling* from above to the example problem in section 3.2.2. The clausal propositional form of this statement is:

$\neg child \lor EQUAL \lor sibling$
$\neg sibling \lor child$
$\neg sibling \lor \neg EQUAL.$

Adding this set of clauses to the graph in figure 3.8, yields the additional graph structure shown in figure 3.10. This is a pure subgraph.

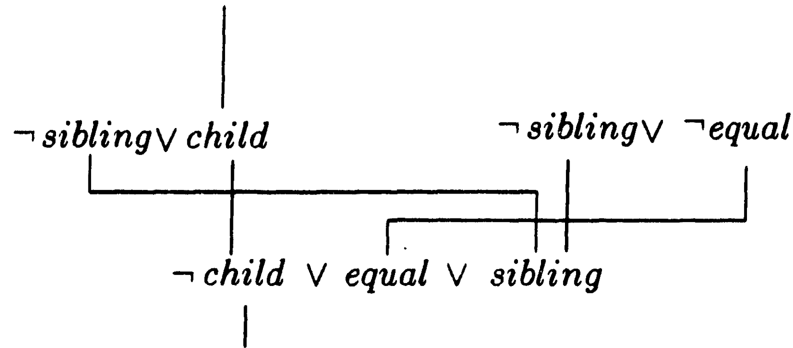

Figure 3.10: Connection graph fragment for definition of *sibling*.
(dangling lines indicate connections to other parts of the problem's graph)

When a literal's reference set is of this form, the system uses the clauses in the set to identify the original problem statements that define the suspect concept. These statements are then checked to see if the concept is indeed irrelevant. Checking the appropriate first order statements is necessary because having a reference set that mentions only general clauses does not ensure that the concept is definitionally irrelevant. Here is a simple counter example showing why the condition is not sufficient. Suppose a concept A has the following reference set:

$A(x,y) \vee \neg B(x,y)$
$\neg A(x,y) \vee B(y,x)$.

These are both general clauses and they constitute a pure graph. However, they do not constitute a definition of A since $B(x,y) \Rightarrow B(y,x)$ follows from these clauses.

A concept is definitionally irrelevant when the set of statements associated with its reference set constrains only that concept. For example, from the definition of *sibling* we can derive only constraints on *sibling*, however, the concept A is not definitionally irrelevant in the statements

$A(x,y) \vee \neg B(x,y)$
$\neg A(x,y) \vee B(y,x)$

because we can derive $\forall x \forall y [B(x,y) \Rightarrow B(y,x)]$, a statement constraining B.

3.2.4 Eliminating Irrelevance in Incomplete Problem Statements

As mentioned, there is a trade off between running the filter early and eliminating relevant concepts and running it later only to find that representations have been designed for irrelevant concepts. Since the cost associated with resurrecting relevant concepts eliminated prematurely is much less than designing unnecessary representations, it is best to run the filter early and keep track of what gets eliminated so that statements can be resurrected as information is added.

After running the filter, the user is asked about another type of potentially missing information: connections between primitive concepts. The user has already been asked for a definition for every primitive concept. If none was given or if the definition given did not connect the concept, it may be that there is a missing necessary or sufficient condition for the concept that will. To address this possibility, the following heuristic is used:

> "When a literal is determined to be disconnected, ask the user if he can supply necessary or sufficient conditions for it."

This is a heuristic in the sense that it can never be guaranteed to ask all potentially useful questions about the connections between concepts. Any concept could always connect to a non-primitive concept. The only way to augment this acquisition heuristic so that it will ask for all possible connections is to ask about possible necessary or sufficient conditions connecting every pair of concepts in the problem. This is impractical.

When a new statement is provided at this point, it is converted to propositional clause form and added to the connection graph. This may cause statements that were earlier judged to be irrelevant to be resurrected. Also if any totally new concepts are mentioned in a new statement, the representation design system returns to the definition acquisition phase for those concepts and adds any new definitional information into the graph.

3.2.5 Section Summary

Figure 3.11 shows the state of our example problem after definitional knowledge is added. When the irrelevance filter is run on this problem. *married* is identified as irrelevant. As a result, the system asks the user to supply necessary or sufficient conditions connecting *married* and *child* (the other primitive concept in the problem). The user presumably responds with the two statements:

$\forall x \forall y \forall c[child(x,c) \wedge child(y,c) \wedge x \neq y \Rightarrow married(x,y)]$
$\forall x \forall y \forall c[married(x,y) \wedge child(x,c) \Rightarrow child(y,c)]$.

sort($P, FAMILY\text{-}MEMBER$), sort($Q, FAMILY\text{-}MEMBER$),
sort($R, FAMILY\text{-}MEMBER$), sort($S, FAMILY\text{-}MEMBER$)
$grandchild(Q,S)$
$\forall x\ child(P,x) \Leftrightarrow x = R$
$married(Q,P)$
$\forall x \forall y[grandchild(x,y) \Leftrightarrow \exists z(child(x,z) \wedge child(z,y))]$
$\forall x \forall y[child(x,y) \Leftrightarrow parent(y,x)]$
Query: find-all x: $parent(S,x)$

Figure 3.11: The example problem with definitions added.

3.3 Deriving Instance Definitions for the Represented Concepts

The representation design system constructs a description of a problem's initial representation from the problem statement. To understand this process, recall that the problem statement language is a sorted logic. The FAMILIES problem contains one sort called *FAMILY-MEMBER*. The system represents a problem's sorts as collections.

The description of a problem's initial representation contains a definition for the initial representation of each relevant primitive concept and each concept with a relevant mixed constraint on it. A concept's initial representation is defined from its syntactic category and the sorts that it is defined over. For example, *married* is

a binary relation defined over family members. In cases where the system can not determine the sorts that a concept is defined over, it asks the user to supply them. For example, if it were not able to tell that family members are married it would ask what sorts of individuals are.

A few restrictions are placed on problem statements to make it possible to extract information about the sorts. An initial problem statement must make reference to one or more domain sorts that are assumed to be collections of domain individuals. For example, there is one domain sort in FAMILIES which is referred to as *FAMILY-MEMBER*. All domain sorts are assumed to be disjoint from all others.

All constants in the problem statement must have their sort declared. For example, note that in the FAMILIES problem there is a sort declaration (e.g., sort(N, *FAMILY-MEMBER*)) for every individual mentioned.

To describe the fact that *FAMILY-MEMBER* is a domain sort, the system declares it as follows:

COLLECTION* FAMILY-MEMBER OF individual

This statement means that family members are a subtype of the individuals in the "world." The asterisk means that family members are disjoint from all other domain sorts.

Given sorts for all the individuals, the system attempts to determine the domain of relations and the domain and range of functions by inspecting the statements in which they appear.

It attempts to determine the domain of a relation in two ways. When there is a problem statement that relates individuals, the domain of the relation involved is defined from the sorts of the individuals. For example, from the statements

sort(A, *FAMILY-MEMBER*)
sort(B, *FAMILY-MEMBER*)
married(A, B)

it is determined that *married* is a binary relation on family members.

The initial representation of a relation is defined from the information about its domain in terms of the library structure **relation**. For example, *married* is defined as

MARRIED: **relation**(FAMILY-MEMBER,FAMILY-MEMBER).

When the domain of a relation can not be determined directly, the system looks for statements that use variables in the argument position in question. It then tries to determine the sort of that variable by its other uses in the statement. For example, given,

SIBLING: **relation**(FAMILY-MEMBER,FAMILY-MEMBER),

and the statement,

$\forall x \forall y [brother(x,y) \Leftrightarrow sibling(x,y) \wedge male(y)]$,

the representation design system can determine that *brother* is a binary relation defined over family members and that *male* is a unary relation over family members.

The system attempts to determine the sorts in the domain of a function in the same way as relations. For the range of a function, it looks for uses of the function as an argument in a relation whose domain is known. If this fails to produce a sort for the range, then the system looks for an equality between an application of the function and another term whose sort is known. For example, from the statements,

$\mathrm{SORT}(A, FAMILY\text{-}MEMBER)$
$\mathrm{SORT}(B, FAMILY\text{-}MEMBER)$
$father(B) = A$,
father is defined as

FATHER: **function**(FAMILY-MEMBER,FAMILY-MEMBER).

Problem statements are also checked for consistent use of functions and relations. When the sort of a function or relation is ambiguous in the problem statement, the system asks the user to disambiguate.

3.3.1 Section Summary

The primitive concepts of our example problem are *married* and *child*. Their representations are defined as follows:

MARRIED: **relation**(FAMILY-MEMBER,FAMILY-MEMBER)
CHILD: **relation**(FAMILY-MEMBER,FAMILY-MEMBER).

3.4 Chapter Summary

This chapter has described the derivation of the description of the initial representation of a problem. The basic strategy is to identify a collection of concepts that is sufficient for representing the problem and then to define representations for them. A specialized representation will be designed by classifying the concepts in this collection. It is desirable to identify the smallest collection of concepts that is sufficient for this purpose to avoid designing redundant representation machinery. The smallest collection of concepts that is sufficient is the collection of relevant primitive concepts plus concepts that have relevant restrictions on them.

The relevant primitive concepts are identified in three steps:

1. The primitive concepts are identified and the user is given the opportunity to give definitions for any of these that are not, in fact, primitive.

2. Irrelevant concepts are eliminated from the problem. A key idea in understanding irrelevance is the notion of strong irrelevance: a fact is strongly irrelevant to a problem class when it can not be used in solving any problem in the class. Since representations are designed for problem classes this is the kind of irrelevance we attempt to eliminate.

3. Concepts that have explicit restrictions on them are identified. Concepts that have implicit restrictions on them (i.e., those for which a restriction follows from the problem but is not stated) are identified later in representation design.

The final step in deriving the description is to define representations for the concepts identified by the previous steps. Initial representations for concepts are defined from their syntactic category and the sorts they are defined over. This information is extracted from the problem statement when possible. Otherwise the system asks the user to supply this information.

Chapter 4

Classification

The representations that my system designs are mappings from concepts to structures that have behavior. Constraints on a concept are captured structurally when the concept is represented with a structure having the same properties. The system has a library of structures, part of which is shown in figure 4.1. It captures constraints on a concept structurally by identifying the library structure that captures the most constraints on the concept and then representing the concept with that structure. Such a structure is identified by classifying concepts in a taxonomy organized around the constraints that library structures capture (again, see figure 4.1). Structures capturing more constraints are more specialized.

The taxonomy is organized by increasing specialization and the links are labeled with the additional constraints that more specialized structures capture. For example, `1-1 function` appears below `function` and the link between them is labeled "inverse single valued." Figure 4.2 gives axiom schemas defining the meaning of each of the constraint names used in figure 4.1.

The set of links leaving a node covers all possibilities for a single constraint. For example, there are three links leaving the `ref-rel` (reflexive relation) node in figure 4.1 to cover the possibility that a reflexive relation is symmetric, antisymmetric, or neither. Not all taxonomy nodes have structures associated with them because some

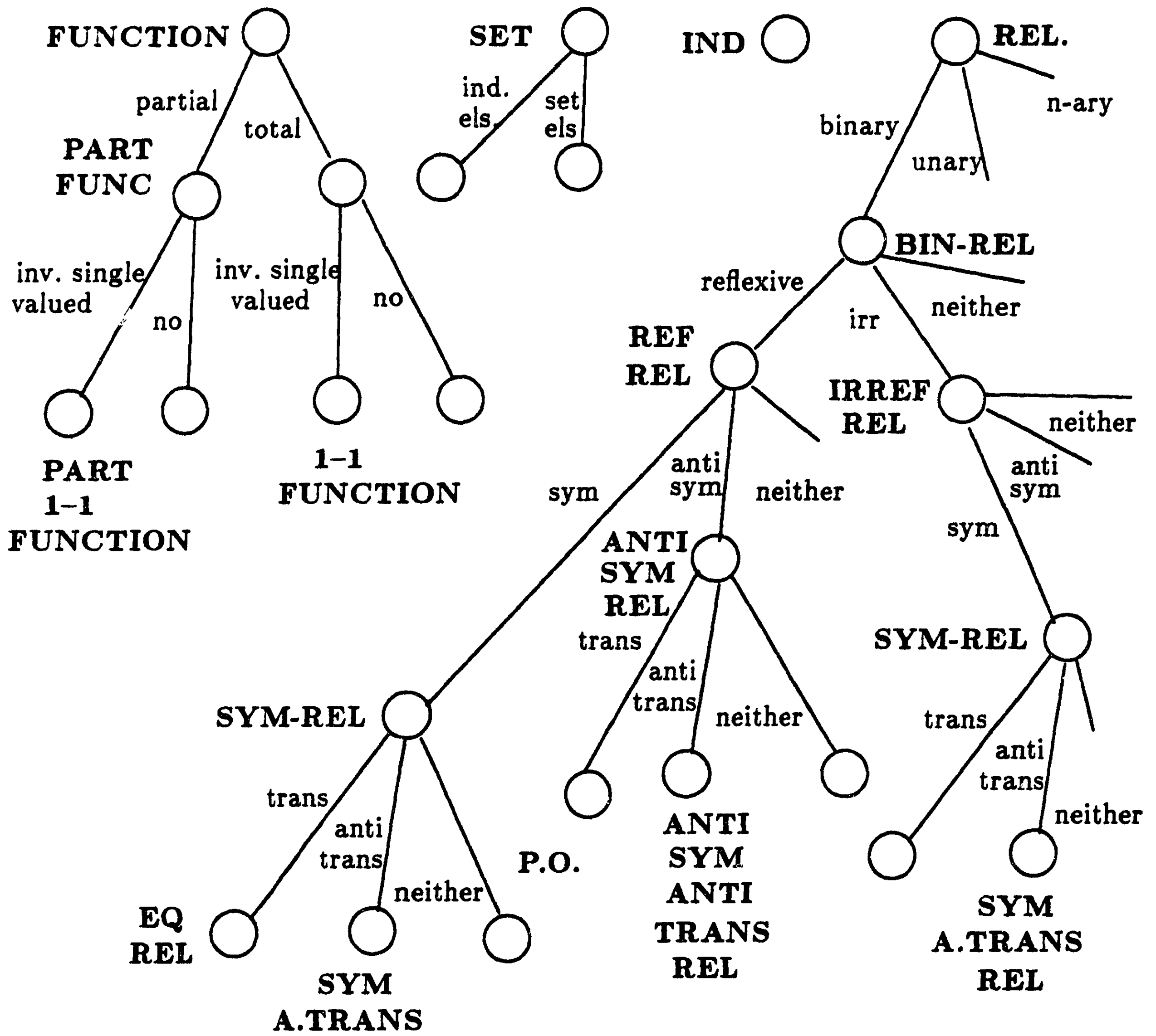

Figure 4.1: Part of the structure library.

Property	*Definition*
reflexive	$\forall x\ R(x,x)$
irreflexive	$\forall x\ \neg R(x,x)$
symmetric	$\forall x \forall y [R(x,y) \Rightarrow R(y,x)]$
antisymmetric	$\forall x \forall y [R(x,y) \Rightarrow \neg R(y,x)]$
transitive	$\forall x \forall y \forall z [R(x,y) \wedge R(y,z) \Rightarrow R(x,z)]$
antitrans	$\forall x \forall y \forall z [R(x,y) \wedge R(y,z) \Rightarrow \neg R(x,z)]$
total	$\forall x \exists y\ y = f(x)$
inv. 1-valued	$\forall x \forall y [f(x) = f(y) \Rightarrow x = y]$

Figure 4.2: The definitions of the properties labeling links in figure 4.1. (Properties not defined here are determined by inspection.)

links indicate the absence of a constraint and, therefore, there is no more specialized structure for that node. For example, the fact that a relation is neither symmetric nor antisymmetric can not be captured by a structure.

The system also designs representations for collections of "objects." Classifying collections increases the kinds of constraints that can be captured by classification. There is a separate taxonomy for classifying collections, shown in figure 4.3. The structures in this taxonomy capture the common constraints on individuals in a collection and constraints between individuals in a collection. Axiom schemas giving the meaning of each of the constraint named used in figure 4.3 are given in figure 4.4

Notice that the top nodes in the collection taxonomy are the same as the concept taxonomy. Even so, there is an important difference: The structure associated with a node in the concept taxonomy represents a single concept, while a structure in the collection taxonomy represents a collection whose individuals are that kind of concept. For example, the **set** structure in the concept taxonomy is used to represent a concept that is a set, while the **set** structure in the collection taxonomy is used to represent a collection of sets.

As a concept (or a collection) gets classified, its representation gets specialized by redefining it in terms of the library structures that are reached. For example, *married*,

REL.

IND

unique names

UNIQUE IND

all known

SET

same size members

no

FIXED SIZE SET

limited size members

no

disjoint members

no

LIM SIZE SET

empty members

no

disjoint members

no

FIXED SIZE DISJOINT SET

disjoint members

no

LIM SIZE DISJOINT SET

DISJOINT SET

FUNCTION

Figure 4.3: The collection taxonomy.

same size	$\forall x \exists y_1, \ldots, y_n [\{y_1, \ldots, y_n\} \subseteq f(x) \subseteq \{y_1, \ldots, y_n\}]$
limited size	$\forall x \exists y_1, \ldots, y_n [f(x) \subseteq \{y_1, \ldots, y_n\}]$
empty	$\exists x [f(x) = \emptyset]$
disjoint	$\forall x \forall y \forall z [z \in f(x) \wedge z \in f(y) \Rightarrow f(x) = f(y)]$

Figure 4.4: Axiom schemas defining the meaning of the constraints names used in the collection taxonomy.
(Note that when these schemas are instantiated during a classification effort, the term $f(x)$ is replaced by a function mapping onto the collection being classified.)

whose initial representation is

MARRIED: **relation**(FAMILY-MEMBER.FAMILY-MEMBER),

gets classified as an irreflexive binary relation, so MARRIED gets redefined as

MARRIED: **irref-rel**(FAMILY-MEMBER,FAMILY-MEMBER).

When a representation is more specialized, it is better for two reasons. The first reason is that the search space explored by a problem solver using a specialized representation is smaller because the set of expressible situations in that representation is smaller. For example, when the concept F is represented as a `function`, problem situations can be expressed that have multiple domain elements mapping to the same range element. When F is represented as a `1-1 function`, it is more constrained because the legal situations in the new representation are a subset of those in the previous representation, i.e., those situations in which domain elements map to unique range elements. To make the example concrete, when F is defined in terms of `function`, we can create a situation representing the following facts,

$F(A) = B$
$F(C) = B$

By contrast, if F is defined in terms of `1-1 function`, we can not create a situation representing these facts. If we try, then the instance F will signal a contradiction. As a result, when a problem solver uses the specialized representation of F, it does not have to explore problem situations like this.

The second reason that more specialized representations are better is that more specialized library structures exploit constraints to gain efficiency. For example, one specialization of **set** in the collection taxonomy is **disjoint-set**. **Set** provides an equality procedure which reports that two sets are equal if all elements of both sets are known and they are the same. **Disjoint-set** provides a more efficient equality procedure which exploits the additional constraints that the sets being compared are disjoint: it reports true if it can show that two sets share any members. Thus, it is more efficient in two ways: it does not necessarily have to check all members of the

sets its comparing and it can work even if some members of those sets are unknown.

One way to view classification is as a goal directed process of identifying useful constraints in a problem. This gives the system a way of directing the search for interesting constraints, saying that they are those that library structures can capture. Without such a technique the representation design process is faced with an unstructured collection of problem statements, not knowing which are important for design or which it should try to capture first.

Classification can be used to assist in the acquisition of missing information. The constraints that it asks about are assumed to be important enough in representation design and applicable to a sufficiently wide variety of problems that they are worth asking about when they are not found in a problem statement. This assumption is discussed below in section 4.2.2.

Classification can not capture all possible kinds of constraints. Intuitively, we can divide possible constraints into those that constrain a concept in terms of itself and those that constrain a concept in terms of other concepts. An example of the first type is,

$$\forall x \forall y [married(x,y) \Leftrightarrow married(y,x)]$$

and an example of the second type is,"two different parents of an individual are married."

Classification can only capture constraints of the first kind because it classifies concepts individually. A similar statement can be made about classifying collections: it can only capture constraints between individuals in that collection. For example, one collection introduced during design of a representation for FAMILIES is *parent-set*. It can capture the constraint, "parent sets are disjoint from each other which is a constraint between individuals in a collection.

Operationalization can capture both types of constraints and is the only way to capture constraints between concepts. This is one reason it is done after classification.

4.1 Implementation of Library Structures

Recall that library structures are implemented as abstract data types. Each ADT is a prototype for creating representations. Representations of concepts are created from ADTs in the concept taxonomy by instantiation. For example, a structure that results from instantiating **relation** is used to represent a particular relation like *married.* This structure is created by the definition

MARRIED: **relation**(FAMILY-MEMBER,FAMILY-MEMBER).

As has been explained, this instance stores a list of the pairs of family members that are married in a problem situation.

Representations of collections are created from ADTs in the collection taxonomy by defining subtypes. For example, a representation of the collection *FAMILY-MEMBER* is created by the definition

COLLECTION FAMILY-MEMBER OF **individual**.

Instances of FAMILY-MEMBER are used to represent family members like N.

The top nodes in the two taxonomies are, in fact, labeled with the same ADTs. Concept representations are created from these by instantiation, while collection representations are created by defining subtypes. For example, a concept that is a set is represented with an instance of **set**, while a collection of sets is represented with a subtype of **set**.

4.2 The Classification Process

The main loop in classification systematically selects concepts that are included in the initial representation description and "pushes" them down into the concept taxonomy. The collections that a concept is defined over are classified (by pushing them into the collection taxonomy) before that concept is classified.

For example, when the main loop of classification selects *married* to be classified, it inspects the definition of MARRIED to determine if the collections it is defined over have been classified. The definition of MARRIED is in terms of FAMILY-MEMBER, so *FAMILY-MEMBER* will be classified (in the collection taxonomy) before beginning classification of *married*.

Classification is a goal directed process that normally begins at the top node of one of the taxonomies and tries to work its way down. At each step, the goal is to determine whether the concept being classified has any of the constraints labeling the specialization links of that node.

When classification reaches a node that has a library structure associated with it, the concept is redefined in terms of that library structure.

As an example, consider classification of *married*. For the moment we will assume that the collection *FAMILY-MEMBER* has already been classified. Classification begins at the **relation** node in the concept taxonomy. The first question is the arity of *married*. It can be answered by looking at any use of the concept that it is binary (actually, in this case, it can be determined directly from the definition of the relation's representation). Following the links down, classification asks about the reflexivity, symmetry, and transitivity of *married*. Techniques for answering these questions are discussed below; for now, assume the representation design system determines that *married* is irreflexive, symmetric, and antitransitive. No more classification can be performed since we are now at the leaf node for symmetric antitransitive relation (shown shaded in figure 4.5). This classification effort results in the following definition for MARRIED,

MARRIED: **antitrans-sym-irref-rel**(FAMILY-MEMBER,FAMILY-MEMBER).

We now address two issues. The first is how the representation design system goes about answering questions posed by classification. The second is the assumptions about the knowledge in the structure library and how these contribute to the useful-

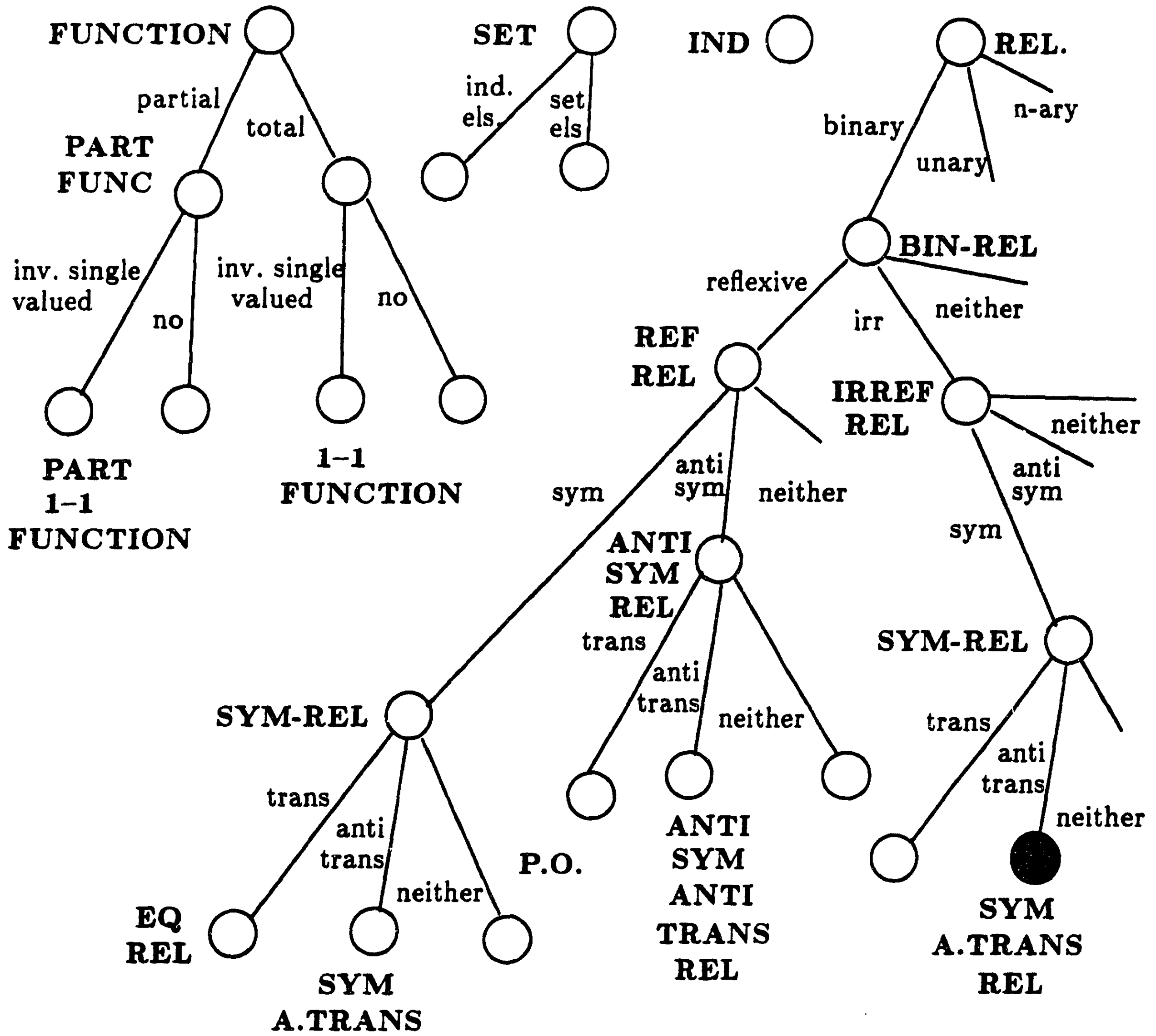

Figure 4.5: Node reached in classifying *married*

ness of classification as a representation design and knowledge acquisition technique. We then return to consider a classification example in more detail.

4.2.1 Answering Questions Posed by Classification

A "brute force" approach to answering questions posed by classification would be to construct a statement of the property in the question and then to use a theorem prover to try to show that the statement follows from the problem statement. For example, to determine whether *married* is symmetric, we could construct the statement,

$$\forall x \forall y [married(x,y) \Leftrightarrow married(y,x)],$$

and then instruct a theorem prover to determine whether this statement follows from the problem statement.

However, recall that analytical reasoning problems are usually missing information. This means that a problem statement may not contain any information about a property, either because the property is not relevant to solving the problem or because the information is simply missing. This is problematic for the brute force approach. If we turn a theorem prover loose on some proof attempt, it may never halt and we can never know whether this is because we have not given it enough time or the problem statement is incomplete.

The representation design system constructs a statement of the property but does not try to prove that it follows from the problem. Instead, it checks the problem for a statement "nearly" matching the one it constructs. "Nearly matching" is defined by a combination of two mechanisms in the system. First, it employs a rewrite system that simplifies problem statements. Second, the matcher recognizes a few syntactic variations.

If the system's attempt to find a statement fails, it assumes that the information is missing from the problem and asks the user about it.

The advantage of this approach is it avoids asking the user most questions that would be considered obvious, and at the same time avoids getting bogged down trying to prove that some constraint holds when a problem statement may not contain the information.

The approach is implemented as follows. First, as problem statements are added to the system's description, they are simplified by the rewrite system. The intention of this step is to "partially" canonicalize statements, attempting to rewrite them so they will be recognized by classification. The canonicalization is partial in that it is not guaranteed to produce a unique form for statements with the same meaning.

The rewrite system is implemented in a standard way, as a collection of rules that match and replace patterns in statements. The current body of rules exploits properties of sets to simplify statements. One of the rules is paraphrased as follows:

> "If a statement contains a conjunction of expressions two of which are of the form, $x \in S$ and $y \in S$, and a third expression in the conjunction has the form $x \neq y$, then replace the first two expressions by $\{x, y\} \subseteq S$."

This rule will, for example, rewrite the statement

$$\forall x \exists y \exists z [y \in parents(x) \wedge z \in parents(x) \wedge y \neq z]$$

as

$$\forall x \exists y \exists z [\{y, z\} \subseteq parents(x) \wedge y \neq z].$$

Both statements express the fact that every individual has at least two parents, but only the second statement will be recognized by classification.

To check the problem for a property, the system generates a statement expressing that property from a schema by substituting the appropriate concept for the concept symbol in the schema. Every property in the taxonomy has one or more such schemas associated with it (these were given earlier in this chapter in figure 4.2 and figure 4.4). For example, the fixed size property has two schemas expressing that a set is bounded from below and above respectively. The schema for "bounded from below" is

$$\forall x \exists \vec{y}\,[\{\vec{y}\} \subseteq C(x)].$$

When classifying the collection of parents of an individual, C is replaced by *parents* in this schema, where *parents* is a function mapping a family member to his/her set of parents. The result of the replacement is

$$\forall x \exists \vec{y}\,[\{\vec{y}\} \subseteq parents(x)].$$

When this statement is matched against problem statements, $\vec{y}$ will match any sequence of terms, allowing the system to check for a constant set of any size.

The matcher allows statements that are syntactic variations of each other to match. Two syntactic variations are recognized. First, $\wedge$ and $\vee$ are treated as commutative, associative operators so that two statements will match if changing the order of a group of conjuncts in one of the statements makes it unify with the other statement. For example, the following two statements match:

$$\forall x \forall y [P(x,y) \wedge (Q(x,y) \wedge R(x,y))]$$
$$\forall u \forall v [(R(u,v) \wedge P(u,v)) \wedge Q(u,v)].$$

Second, two statements that are right associated sequences of implications will match if changing the order of any of the sequence elements but the last will allow the two statements to unify, i.e., a statement of the form

$$\phi_1 \Rightarrow \cdots (\phi_{n-1} \Rightarrow \phi_n)$$

will match any statement that unifies with a right associated sequence of implications in which $\phi_1, \ldots, \phi_{n-1}$ are permuted. For example,

$$\forall x \forall y [P(x,y) \Rightarrow (Q(x,y) \Rightarrow R(x,y))]$$

matches

$$\forall u \forall v [Q(x,y) \Rightarrow (P(x,y) \Rightarrow R(x,y))].$$

4.2.2 Assumptions About Library Structures

For the library to be useful, it should be the case that the properties found in it appear in a wide variety of problems and that enforcing them as constraints provides significant leverage in specializing representations. When this is the case, classification becomes a technique for recognizing when a problem contains properties that the representation design system knows special ways of capturing. It assumes that certain properties are worth looking for in a problem because of the leverage obtained in specializing representations that enforce those constraints. For example, it assumes that it is worth trying to determine when relations are symmetric because it has special ways of capturing symmetry.

There is at least one rule of thumb in searching for structures that are widely applicable and provide significant leverage: the more general the properties that a structure captures the larger the variety of problems it will be usable in. However, it is usually the case that the more general a structure is the less leverage it provides in representation design. One of the challenges in finding the "right" collection of structures to look for is trading off generality against usefulness.

In this research, useful structures were found by studying the representations that people use to solve analytical reasoning problems. I looked for structures that people commonly use in their representations and I identified the properties that those structures enforce. The current library population is the result of this investigation conducted with twenty analytical reasoning problems. This investigation is discussed in Chapter 8.

It may be that the particular collection of types in the current library prove to be less applicable as different problems are investigated. However, notice that the structures in the library are similar to concepts that have been important in mathematics for a long time. I did not build the library by attempting to replicate what I knew mathematicians consider important. Instead, I tried to capture what I found people

using in representations. Because I ended up with a collection of structures similar to those that mathematicians consider important, I believe that it is likely that these have fairly broad applicability.

The issue of whether the current collection of structures will provide significant leverage in a wide variety of problems is empirical. So far, the only substantive claim I can make is that they do provide significant leverage in designing representations for twenty analytical reasoning problems.

Assuming that library structures have these properties, classification can be viewed as the following useful knowledge acquisition heuristic:

> "Properties of library structure are useful enough that if they are not present in a problem, they are worth asking about."

This is a heuristic because it can cause the system to acquire information that is irrelevant to solving a problem and this, in turn, causes it to over-design a representation.

Finally, the idea of organizing one's knowledge in a specialization taxonomy has already been shown to be useful both for classification problems and for knowledge acquisition. The contribution of this work is showing that designing good representations can be viewed, in part, as specialization.

4.2.3 Classification Example Revisited

We demonstrate the classification of *married* as it is performed by the system when given the problem used in our running summary. The purpose of this example is to illustrate the knowledge acquisition behavior that the system exhibits while classifying. The problem statement modified by processes of the last chapter is shown in figure 4.6.

Before MARRIED can be classified, FAMILY-MEMBER must be classified in the collection taxonomy. Since FAMILY-MEMBER is defined as

$sort(P, FAMILY\text{-}MEMBER)$, $sort(Q, FAMILY\text{-}MEMBER)$,
$sort(R, FAMILY\text{-}MEMBER)$, $sort(S, FAMILY\text{-}MEMBER)$
$grandchild(Q, S)$
$\forall x\ child(P, x) \Rightarrow x = R$
$married(Q, P)$
$\forall x \forall y [grandchild(x, y) \Rightarrow \exists z(child(x, z) \wedge child(z, y))]$
$\forall x \forall y [child(x, y) \Rightarrow parent(y, x)]$ $\forall x \forall y \forall c [child(x, c) \wedge child(y, c) \wedge x \neq y \Rightarrow married(x, y)]$
$\forall x \forall y \forall c [married(x, y) \wedge child(x, c) \Rightarrow child(y, c)]$
Query: find-all x: $parent(S, x)$

Figure 4.6: State of the example problem at the end of the last chapter.

FAMILY-MEMBER: **individual**,

classification begins with the **individual** node in figure 4.3. The first question is whether the names mentioned in the problem statement refer to unique individuals. To determine this, the system looks for inequalities between all the individuals. The problem statement does not contain any, so the system asks the following question:

Are M, N, O, P, Q, R, S all different individuals?
Yes.

All the analytical reasoning problems I have studied make the unstated assumption that individuals with different names are different. However, this could easily not be the case, so the system explicitly asks about this assumption. Since the answer is "yes," FAMILY-MEMBER is specialized as

COLLECTION* FAMILY-MEMBER OF **unique-individual**.

The structure **unique-individual** represents a collection of individuals with the property that individuals with different names implies are different. This is implemented by an equality procedure associated with **unique-individual** that simply compares individual's names.

Classification of FAMILY-MEMBER now terminates because we have reached a leaf node. The system proceeds with the classification of MARRIED. As before, it is classified by inspection as binary.

Next comes the question of whether it is reflexive, irreflexive, or neither. The representation design system looks for statements in the problem that answer the question. None are found so the user is queried again:

```
Is MARRIED reflexive? No.
Is MARRIED irreflexive? Yes.
```

In response to this answer, MARRIED is specialized as

MARRIED: `irref-rel`(FAMILY-MEMBER,FAMILY-MEMBER)

and the statement $\forall x \neg married(x, x)$ is added to the problem.

Then comes the question of whether it is symmetric, antisymmetric, or neither.

```
Is MARRIED symmetric? Yes.
```

In response to this answer, MARRIED is further specialized as

MARRIED: `sym-irref-rel`(FAMILY-MEMBER,FAMILY-MEMBER)

and the following statement is added to the problem:

$$\forall x \forall y [married(x, y) \Leftrightarrow married(y, x)].$$

Classification now proceeds to the question of transitivity and, again, the problem statement provides no assistance, so the user is asked:

```
Is MARRIED transitive? No.
Is it antitransitive? Yes.
```

MARRIED is now specialized as

MARRIED: `antitrans-sym-irref-rel`(FAMILY-MEMBER,FAMILY-MEMBER)

and the following statement is added to the problem,

$$\forall x \forall y \forall z [married(x, y) \wedge married(y, z) \Rightarrow \neg married(x, z)].$$

This classification effort is now complete. The resulting problem statement is shown in figure 4.7 with the added statements enclosed in a box.

Also the description of the specialized representation designed so far is:

sort($P, FAMILY\text{-}MEMBER$), sort($Q, FAMILY\text{-}MEMBER$),
sort($R, FAMILY\text{-}MEMBER$), sort($S, FAMILY\text{-}MEMBER$)
$grandchild(Q, S)$
$\forall x\ child(P, x) \Leftrightarrow x = R$
$married(Q, P)$
$\forall x \forall y [grandchild(x, y) \Leftrightarrow \exists z(child(x, z) \wedge child(z, y))]$
$\forall x \forall y [child(x, y) \Rightarrow parent(y, x)]$ $\forall x \forall y \forall c [child(x, c) \wedge child(y, c) \wedge x \neq y \Rightarrow married(x, y)]$
$\forall x \forall y \forall c [married(x, y) \wedge child(x, c) \Rightarrow child(y, c)]$

$\forall x \neg married(x, x)$ $\forall x \forall y [married(x, y) \Leftrightarrow married(y, x)]$ $\forall x \forall y \forall z [married(x, y) \wedge married(y, z) \Rightarrow \neg married(x, z)]$

Query: find-all x: $parent(S, x)$

Figure 4.7: Our example problem after *married* and $FAMILY\text{-}MEMBER$ have been classified.

MARRIED: **antitrans-sym-irref-rel**(FAMILY-MEMBER,FAMILY-MEMBER)
CHILD: **relation**(FAMILY-MEMBER,FAMILY-MEMBER)
COLLECTION* FAMILY-MEMBER OF **unique-individual**.

4.3 Capture Verification

When all the situations that can be created with a collection of representations satisfy a set of statements, those statements are said to be *captured.* [1] This is a central idea in my approach to representation design. This section shows how the idea has been turned into a test to determine when statements have been captured and why such statements can be removed from the system's consideration. The test, called *capture verification*, is done when a classification effort reaches a leaf node in the library taxonomy.

4.3.1 How Capture Verification Works

The basic idea of capture verification is to do a constructive proof using the representations of concepts in a statement to test whether the statements are captured. The

[1] This notion of satisfaction is made precise in chapter 8.

representations are used to build a situation, i.e., a collection of data structures, and then that situation is inspected to determine if the consequences of the statement are true in it.

Suppose, for example, the relation *couple* defines a collection whose members are married couples (i.e., $couple(x)$ is true if x is a set of size two whose elements are family members married to each other). Suppose further that this collection, call it *married-couple*, has been classified as a collection of fixed size disjoint sets and, consequently, is represented as

COLLECTION MARRIED-COUPLE OF
`fixed-size-disjoint-set(2,FAMILY-MEMBER)`.

Finally suppose a problem containing *couple* and *married-couple* also contains the statement

$$\forall x \forall y \forall z[couple(\{x, y\}) \wedge x \neq y \wedge couple(\{x, z\}) \wedge x \neq z \Rightarrow y = z].$$

Capture verification proves that this statement is captured by constructing a situation representing the antecedent of the statement and then checking that situation to see if the consequent is true. First it creates two anonymous family members by instantiating the FAMILY-MEMBER ADT without giving names for the instances. In this exposition we will call these instances X and Y. After creating these, the system assumes that they are unequal by adding the statement $x \neq y$ to the situation. This is done using an equality system (discussed in Chapter 7) that is part of all specialized representations that the system designs. This equality system supports an operation that makes two instances of the same ADT unequal.

Next the system creates an instance of MARRIED-COUPLE whose elements are X and Y. It then performs similar steps to create Z, assume that $z \neq x$, and create a married couple containing X and Z. When the second instance of MARRIED-COUPLE is created, a procedure associated with MARRIED-COUPLE enforces the disjointness property by combining the two separate instances (since they share X and, therefore, must be the same). Then another procedure enforces the fixed size of members of *married-couple*

by combining the instances Y and Z (this is true because neither Y nor Z is equal to X and because the MARRIED-COUPLE may contain only two individuals).

When capture verification checks the situation created, it finds the consequent of the above statement to be true and concludes that the statement is captured.

Capture verification is also used to test general statements that are conjunctions of atomic formulas. This is done by creating anonymous individuals for each universally quantified variable mentioned in the statement and then testing whether the statement is true in the situation containing those individuals. For example, suppose that *child-set* is a collection of sets of family members that are children of the same couple and that *child-set-of* is a function mapping a family member to the set of children he/she is a member of.

Then the statement,

$$\forall x[x \in \textit{child-set-of}(x)],$$

is tested as follows. First we create an anonymous family member X and then we create an instance of CHILD-SET, call it Y. Next, we make Y be the image of X under CHILD-SET-OF. Finally, we check to see if X is a member of Y.

In fact, capture verification can be used on statements of the form

$$\phi_1 \Rightarrow (\phi_2 \Rightarrow \cdots (\phi_n \Rightarrow (\psi_1 \wedge \cdots \wedge \psi_m))),$$

where ϕ_i and ψ_i are atomic formulas, or equivalently on statements of the form

$$(\phi_1 \wedge \cdots \wedge \phi_n) \Rightarrow (\psi_1 \wedge \cdots \wedge \psi_m).$$

I call this *implication normal form.* Note that a conjunction of atomic formulas is in implication normal form, i.e., n can be 0.

Statements of this form are tested by creating a situation representing the conjunction of $\phi_1, ..., \phi_n$ and then checking that all the ψ_i are true in that situation.

Any first order statement can be converted to an equivalent statement in implication normal form. Therefore, this test can be used on any first order statement. However,

as a practical matter, the representation design system does not do this during classification. Instead it has special purpose methods for handling some disjunctions and negations. The methods are not complete in the sense that converting to implication normal form is. However, they handle the cases that come up in practice.

As an example of one such method, consider testing statements of the form

$$(P \Rightarrow Q) \Rightarrow R.$$

Since this statement is equivalent to

$$(\neg P \vee Q) \Rightarrow R,$$

which, in turn, is equivalent to

$$\neg P \Rightarrow R$$
$$Q \Rightarrow R.$$

The method that tests statements in this form encodes this fact in a procedure by creating a situation in which $\neg P$ is true, checking that R is true, then creating a situation in which Q is true, and finally checking that R is true in that situation.

4.3.2 How Capture Verification is Used

As representations are designed, constraints that are expressed in problem statements get captured by those representations. Capture verification is used to identify statements in a problem that express captured constraints. The system removes such statements so that the problem statement is, at any point, an accurate record of which statements are not captured by the specialized representations designed. The problem statement plays two important roles as a record of statements left to be captured. First, when a problem statement is empty, the system knows that all of the constraints of a problem are captured and, consequently, the design effort is complete. Second, recall that concept introduction creates alternative problem formulations which are compared by seeing what is left uncaptured in each. This comparison process relies on the problem statement being an accurate record of which statements are not captured.

4.3.3 Why Capture Verification Works

One reason capture verification works is because the library structures are *monotonic.* This means that when a problem situation is created with a structure, all the specific facts that are true in that situation continue to be true no matter what information is subsequently added to that situation.

As an example of a structure that violates monotonicity, suppose `relation` was implemented as one list of n-tuples with the absence of some n-tuple being interpreted as the negation, e.g., the absence of $< A, B >$ from the MARRIED list meaning $\neg married(A, B)$. In this implementation `relation` is not monotonic because adding pair $< A, B >$ changes the truth value of $married(A, B)$.

Monotonicity has the following important consequences:

1. If a representation captures a set of statements, then any specialization will capture at least those statements.
2. If a representation captures a set of statements, then it captures every subset as well.

The combination of these properties means that any subset of the statements in a problem can be tested by capture verification at any time and if they are captured, subsequent representation design activity will not change this.[2] Therefore, once capture verification indicates that a statement is captured it can be removed from the consideration of subsequent representation design.

If monotonicity is relaxed, this ceases to be the case and statements must be continually rechecked as representation design proceeds. Furthermore, there would no longer be any guarantee that the representation design process monotonically moves towards more fully constrained representations since redefining the representation of a concept could cause captured constraints to become uncaptured.

[2]Actually we must also show that operationalization is monotonic. This is done in chapter 6.

The other reason that capture verification works is that creating situations with anonymous individuals demonstrates the truth of general statements. This is guaranteed by the law of generalization of constants for first order predicate calculus [Bell & Machover 77, p.65] which states that if it is possible to prove a theorem of the form $\alpha(x/c)$ [3] from a set of formulas Φ and the constant c does not occur in Φ or α, then $\Phi \models \forall x[\alpha]$. Introducing new anonymous instances is equivalent to introducing previously unmentioned constants, thus the test is valid.

4.4 Interaction Between Knowledge Acquisition and Capture Verification

In our example problem, classifying *married* results in statements expressing its properties being added to the problem by knowledge acquisition. Then, when classification of *married* reaches a leaf node, capture verification removes these statements. One might think that these two processes are self defeating. This is not the case because originally the constraints on *married* are missing from the problem. When they are removed by capture verification it is because the constraints are captured by MARRIED.

It is the function of knowledge acquisition to attempt to ensure that constraints are not left out of the problem, while the function of capture verification is to ensure that constraints are expressed in the problem statement or captured in the problem representation but not both.

The combination of these processes designs representations correctly for each of the following:

1. Cases where constraints that classification looks for are left out of a problem statement.

[3]The notion $\alpha(x/c)$ means c is substituted for every occurrence of x in α.

2. Cases where classification finds the constraints it is looking for in the problem.

3. Cases where constraints that classification looks for are in the problem but it does not recognize them.

The first case is illustrated by our example problem. In this case, the system acquires missing constraints, designs a representation to capture them, and then verifies that those constraints are indeed captured.

The second case would occur, for example, if our example problem contained the statement

$\forall x \forall y[married(x,y) \Leftrightarrow married(y,x)]$.

In this case, the system would not ask the user about the symmetry of *married* and this statement would be removed by capture verification after the classification of *married*.

The third case occurs because the techniques that the system uses to find a statement (or statements) expressing a constraint are, by design, incomplete. In this case, knowledge acquisition will add redundant information to the problem. However, capture verification is complete and, therefore, when a constraint is captured, *all* statements expressing it are identified by capture verification.

Consider, for example, the point at which classification tries to determine if *married* is symmetric and suppose the problem contained the following statement:

$\forall x \forall y[married(x,y) \Rightarrow married(y,x)]$.

The system will fail to recognize that this expresses the symmetry of *married* and, therefore, it will ask the user (who will presumably indicate that *married* is symmetric). As a result, **MARRIED** will be specialized to a symmetric relation and the statement

$\forall x \forall y[married(x,y) \Leftrightarrow married(y,x)]$

will be added to the problem. Then, capture verification will identify both these statements as captured by the specialized representation MARRIED.

4.5 Summary

A representation captures constraints structurally when the concept is represented with a structure having the same properties. The representation design system captures constraints on concepts structurally by classifying them in a taxonomy of structures.

Classification does knowledge acquisition, assuming that if the properties it is looking for are not found in the problem statement, then they are worth asking the user about. When the user says that a concept has some property, the system adds a statement to the problem that expresses that property. Thus, classification often results in an expansion of the problem statement. The example given in this section is the classification of *married* in our sample problem. The result is reviewed in figure 4.8. The statements added during the classification of *married* are enclosed in the box.

sort($P, FAMILY\text{-}MEMBER$), sort($Q, FAMILY\text{-}MEMBER$),
sort($R, FAMILY\text{-}MEMBER$), sort($S, FAMILY\text{-}MEMBER$)
$grandchild(Q, S)$
$\forall x\ child(P, x) \Leftrightarrow x = R$
$married(Q, P)$
$\forall x \forall y [grandchild(x, y) \Leftrightarrow \exists z(child(x, z) \wedge child(z, y))]$
$\forall x \forall y [child(x, y) \Leftrightarrow parent(y, x)]$
$\forall x \forall y \forall c [child(x, c) \wedge child(y, c) \wedge x \neq y \Rightarrow married(x, y)]$
$\forall x \forall y \forall c [married(x, y) \wedge child(x, c) \Rightarrow child(y, c)]$

$\forall x \neg married(x, x)$
$\forall x \forall y [married(x, y) \Leftrightarrow married(y, x)]$
$\forall x \forall y \forall z [married(x, y) \wedge married(y, z) \Rightarrow \neg married(x, z)]$

Query: find-all x: $parent(S, x)$

Figure 4.8: Review of the state of the example problem.

Classifying *married* also results in a the following specialized representation for it:

MARRIED: **antitrans-sym-irref-rel**(FAMILY-MEMBER,FAMILY-MEMBER).

Capture verification is a general technique for testing whether the constraint of a statement is captured. The system uses it to identify the statements that a classification effort has captured. The system removes statements from its description of the problem when they are captured so that subsequent representation design need not consider them. In our example, the system uses capture verification to remove the statements added by classification. Thus, the statements in the box in figure 4.8 removed.

Chapter 5

Concept Introduction

Introduction extends the classification process by adding new concepts to a problem. A new concept is introduced by defining it in terms of an existing concept in such a way that the semantics of a problem are not changed. When a new concept is introduced, a new representation is also introduced.

New concepts are introduced to explore the classification of alternative problem formulations. New concepts are represented differently than existing concepts. This gives the representation design system access to different parts of the library taxonomy, i.e., the the new representation captures different constraints, provides different procedures, and has different specializations.

Gaining access to a different part of the taxonomy provides the potential to capture more problem constraints or to capture the same constraints more efficiently. Consider, for example, during the design of the representation of the FAMILIES problem, the concept *parents* is introduced and defined in terms of *child* as

$$\forall x \forall y [x \in parents(y) \Leftrightarrow child(x, y)].$$

This has the effect of allowing the system to view the relation *child* as a function into a collection of sets because *parents* is a function from family members to sets of their parents. When *parents* is introduced, representations are introduced for the

function and the collection of sets: PARENTS is defined in terms of **function** and PARENT-SET, is defined as a collection of **sets**. Since all sets of parents have exactly two members, classification specializes PARENT-SET in terms of **fixed-size-set**.

The PARENTS representation, along with the specialized version of PARENT-SET, captures the constraint on the size of parent sets, a constraint that CHILD was not able to capture. The PARENTS representation leaves some of the statements uncaptured that CHILD does capture. The representation design system evaluates these two alternative formulations in an attempt to establish a preference. This requires it to determine how expensive it is to enforce the constraints that each representation leaves uncaptured. When a constraint is left uncaptured by classification, the representation design system tries to capture it by operationalization. In this case, it turns out that the machinery operationalization generates to capture the constraints left uncaptured by PARENTS is less expensive than the machinery generated to capture the constraints left uncaptured by CHILD. Consequently, *parents* is preferred over *child.*

Alternative formulations are constructed by using the logical definition of a new concept to rewrite statements in an existing formulation. For example, the definition of *parents* is treated as a rewrite rule to construct a formulation of the problem in terms of *parents* by rewriting every occurrence of *child* to *parents*. For instance, one statement that is rewritten in the small FAMILIES problem being used as our running example is

$$\forall x \; child(P,x) \Leftrightarrow x = R.$$

The formulation in terms of *parents* contains the equivalent statement

$$\forall x \; P \in parents(x) \Leftrightarrow x = R.$$

The representation design system maintains alternative problem formulations because it can not always tell, when it introduces an alternative concept, whether its representation will be better. The maintenance of alternative formulations is implemented

by a context mechanism in which each statement is tagged by the concepts it mentions. Each alternative problem formulation is identified by the set of concepts in that formulation. The system accesses different problem formulations by specifying different sets of concepts. Given a set of concepts, the context mechanism shows the system only statements whose concepts are all members of the set.

There are times when the system introduces alternative concepts and then decides that the problem should be formulated in terms of more than one alternative. It does this when the representation can be implemented more efficiently by representing more than one equivalent concept. For example, during the design of a representation for FAMILIES, an alternative for *parents* is introduced which is a function mapping a family member to his/her set of children. Let us call this function *children*. After a comparison analysis, it is decided that the problem representation should contain PARENTS and CHILDREN (but not CHILD). From a sufficiency point of view only one is necessary, however, it turns out that the representation containing both captures the problem's constraints more efficiently than a representation of either one alone.

There are also times when the system introduces alternative concepts and is forced to keep more than one equivalent concept. This happens when it is not able to rewrite a problem entirely in terms of an introduced concept.

Classification extended by introduction is called *extended classification*. Extended classification is interesting because while the two processes involved are fairly simple, the behavior of the combination can result in multiple reformulations of a problem. Several examples of this will be given later, including the sequence of introductions that results during the classification of *married*:

1. The concept *spouses* is introduced. This is a function from individuals to the sets of individuals to whom they are married.

2. The concept *non-empty-spouses* is introduced. This is a partial function from individuals to the non-empty sets of individuals to whom they are married.

3. The concept *spouse* is introduced. This is a partial function that captures the fact that individuals have at most one spouse.

4. The concept *couple* is introduced. This is a partial function from individuals to the married couple that they are members of. This function captures the following facts: not all individuals are married, married couples are disjoint from all other married couples, married couples contain exactly two members.

5.1 Introduction Rules

Introduction is implemented as a collection of condition-action rules that are associated with nodes in the structure library taxonomy. A rule's condition part checks properties of the concept being classified. When the conditions are met, the action part introduces a new concept and its representation.

Figure 5.1 gives an example of an introduction rule which is associated with the node in figure 4.5 for symmetric, antitransitive relation. When this node is reached while classifying a relation the system reformulates that relation as a function into a collection of sets. Given a relation R, the rule introduces a function F_R into a collection of sets of the form $\{x \mid R(x,y)\}$ and reformulates the problem by replacing R with the newly introduced concepts. This is done to allow the system to explore the use of the **function** and **set** representations (and their specializations) to design a more specialized representation than was possible for R.

The rule is read as:

> "If the concept R is being classified and it is represented as a binary relation on a collection S, then introduce the new concept F_R defined as $\forall x \forall y [y \in F_R(x) \Leftrightarrow R(x,y)]$.
> Also introduce the collection *R-left-proj* whose members are range elements of F_R and introduce representations F_R and R-LEFT-PROJ."

The representation F_R is defined as a **function** and the representation R-LEFT-PROJ is defined as a collection of **sets**.

IF		R is being classified
	AND	R: **relation**(S.S),
THEN		introduce F_R as,
		$\forall x \forall y [y \in F_R(x) \Rightarrow R(x, y)]$
	AND	introduce the collection *R-left-proj* as,
		SORT($F_R(x)$, *R-left-proj*)
	AND	introduce the representation R-LEFT-PROJ as,
		COLLECTION R-LEFT-PROJ OF **set**(S)
	AND	introduce F_R as,
		F_R: **function**(S,R-LEFT-PROJ)

Figure 5.1: An example Introduction rule

Let us take the clauses of the figure 5.1 one-by-one. The first precondition gives a name R to refer to the concept being classified when the rule is invoked. The second precondition is matched against the system's description of the representation to check that R is represented as a binary relation on a collection named S. The matching operation binds S to the representation of the actual collection that R is defined over.

The first action introduces the new concept F_R with the by-conditional. The second action introduces a collection whose members are range elements of F_R. The third and fourth clauses introduce representations.

In general, introduction rules specify new logic statements (usually one, occasionally two) that define a new concept. By convention, we will always write such a definition with the new concept on the left. As in figure 5.1, rules can also introduce new collections in terms of the new concept. Finally, rules introduce representations for the concept (and collection) they introduce.

The system uses the logical definition of the new concept to create a new rewrite rule each time the introduction rule is applied. The example rule rewrites occurrences of the term $R(x, y)$ as $y \in F_R(x)$, where R is whatever relation is being classified when the introduction rule is applied. These rewrite rules create alternative problem formulations.

The rule in figure 5.1 does not specify any non-trivial pre-conditions; the rule shown

in figure 5.2 illustrates that more interesting preconditions are possible.

```
IF           C is being classified
        AND  COLLECTION C OF fixed-size-set(1,C1)
        AND  F: function(C2,C)
THEN         introduce the function f' as,
             ∀x∀y[x = f'(y) ⇒ x ∈ f(y)]
        AND  introduce the representation F' as,
             F': function(C2,C1)
```

Figure 5.2: Rule that introduces a function into individuals when an existing function is into sets of size 1.

This rule is associated with the node in figure 5.3 for fixed size sets. This node is reached when a collection being classified has individuals that are all sets of the same size. The rule is paraphrased as:

> "If the collection C is being classified, it is represented as a collection of sets of size 1, and there is a representation for a function F mapping onto C, then introduce a new function f' mapping directly into the individuals in these sets."

The function f' is defined so that where the value of f is the singleton set $\{A\}$, the value of f' is A.

Note that, in both the examples, the logical definition of the new concept contains all the information necessary to generate the representations automatically, i.e., it would be a simple matter to generate definitions of F_R, *R-left-proj*, and R-LEFT-PROJ automatically from the logical definition of F_R. However, I allow rules to define the representation associated with a new concept in terms of a specialized library structure. For example, a rule could introduce a new function and define its representation as one-to-one. This allows a person writing rules to encode constraints that he knows are true of a new concept into the representation that a rule introduces so that classification does not have to figure them out for itself.

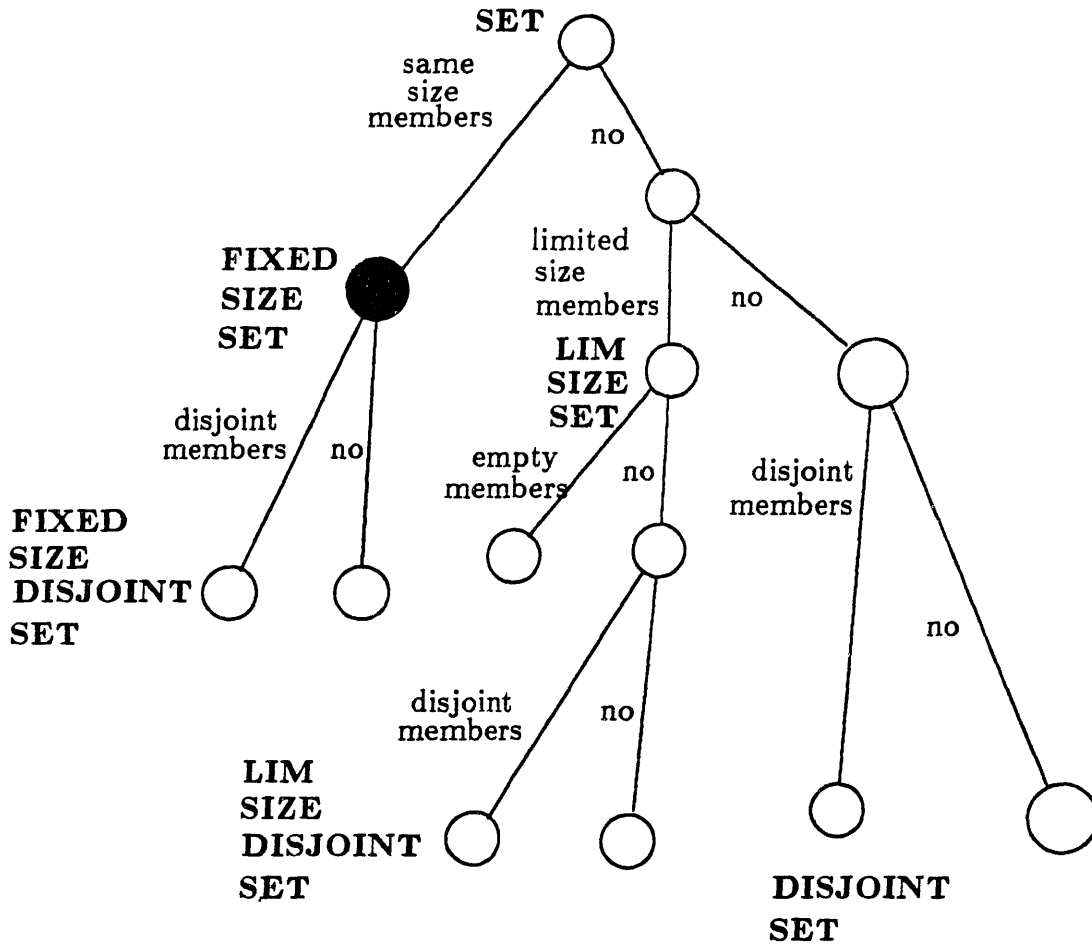

Figure 5.3: Collection taxonomy with node of fixed size set shaded.

5.2 Soundness of Introduction

As we will see in the remainder of this chapter, many new concepts can be introduced during representation design. We would like to be sure that when a solution is found in the new conceptualization of a problem, it is a solution to the original problem. In other words, we would like to show that the introduction process is sound.

From a model theoretic point of view, a new concept is introduced by two actions: adding a new constant to the language of the problem and adding a new statement to the problem. It is well known that there can be no more models of a set of

statements than of any of its subsets. Therefore, since concept introduction adds a new statement, the new problem can not have more models than the original. We must ensure that no models of the original problem are eliminated by concept introduction.

This is done by ensuring that every introduction has the property that every model of the original problem can be extended to a model of the new problem which satisfies the new statement. Since the new models are extensions of models of the original problem, they are themselves models of the original problem. Since every model can be shown to have such an extension, no model of the original problem has been lost. When every introduction meets this restriction, the process as a whole is sound because every introduction step is sound.

This restriction is checked (by hand) for each introduction rule by interpreting the statement that the rule introduces as an abstract procedure to perform on models of the original problem to get the extended models. For example, the rule in figure 5.1 can be shown to meet the restriction by showing how to treat the statement,

$$\forall x \forall y [y \in F_R(x) \Leftrightarrow R(x,y)],$$

as a procedure that extends any model of the original problem.

Take any model of the original problem and add F_R to the set of function symbols of the model. Next, for each element, x in the domain of F_R, create a pair of the form $< x, \{y \mid R(x,y)\} >$. Add the union of these pairs to the domain of the model and designate this new set by the symbol F_R. The new model is an extension of the original model.

Note that when a rule is conditional, we can assume its conditions while checking the restriction. Then we show that any model of the original problem that satisfies the conditions of the rule can be extended to a model of the new statement.

Each introduction rule in the system has been checked and shown sound in this fashion; it follows that the introduction process, as a whole, is sound.

5.3 Exploratory Introduction

The main type of introduction is called exploratory introduction. Its purpose is to introduce alternative concepts when an existing concept has been classified as much as possible and still has not captured all of its constraints. The hope is that the alternative concept will capture more constraints or the same constraints more efficiently.

In order to present the intuition behind the use of exploratory introduction, we introduce the ideas under the assumption that representations are being designed for problems that contain all the information that is necessary to solve them. Since this is not the case with analytical reasoning problems, the system's use of exploratory is more complicated and is explained later.

Exploratory introduction is implemented by associating introduction rules with leaf nodes in the classification taxonomies. When classification reaches a leaf node, capture verification is performed. Statements that can be captured by classification are either general or mixed statements that mention only the concept being classified. If any statements of this form remain uncaptured after a concept is classified, the node's introduction rules are checked.

All the rules at a node whose conditions are satisfied are applied; since there can be more than one rule associated with a leaf node more than one alternative concept can be introduced at a node.

Note that a new concept may not capture statements that the original concept captured. Therefore, the formulations associated with new concepts contain reformulated versions of all the statements mentioning the original concept whether or not they were captured by the representation of the original concept.

An example of an exploratory introduction rule was given above in figure 5.1. This rule is associated with the node in figure 5.4 for symmetric antitransitive relation.

This rule is applied in the FAMILIES problem when *married* is classified as an irreflexive, symmetric, antitransitive relation. The rule introduces the concept *spouses*, a function from family members to their sets of spouses. It is defined with the following statement:

$$\forall x \forall y [y \in spouses(x) \Leftrightarrow married(x, y)].$$

A formulation of the problem in terms of *spouses* is generated when this concept is introduced. This formulation contains a new statement for every statement in the original formulation that mentions *married*, including those statements already captured by MARRIED (e.g., symmetry). Each new statement is logically equivalent to its counterpart in the original formulation but is expressed in terms of *spouses*. For example, the statement,

$$\forall x \forall y [married(x, y) \Leftrightarrow married(y, x)],$$

has an equivalent counterpart in the new formulation:

$$\forall x \forall y [y \in spouses(x) \Leftrightarrow x \in spouses(y)].$$

The representation design system proceeds to classify *spouses* (discussed later in this chapter). It turns out that as *spouses* is being classified, an alternative is introduced for it: *spouse*, a function from family members to their spouses. The system determines that *spouse* is preferable to *spouses*. Once classification of *spouse* is complete, the representation design system runs capture verification on the statements mentioning *spouse*, then compares the formulation in terms of *spouse* with the formulation in terms of *married* and determines that *spouse* is preferable.

5.3.1 Exploratory Introduction in Incomplete Problems

Analytical reasoning problems often omit information necessary to solve the problem. Therefore, the system can not assume when a leaf node is reached classifying a concept, that exploratory introductions need only be tried if there are uncaptured

FUNCTION
partial
total
PART FUNC
inv. single valued
no
inv. single valued
no
PART 1–1 FUNCTION
1–1 FUNCTION
SET
ind. els.
set els
IND
REL.
binary
unary
n-ary
BIN-REL
reflexive
irr
neither
REF REL
sym
anti sym
neither
IRREF REL
neither
anti sym
sym
ANTI SYM REL
trans
anti trans
neither
P.O.
ANTI SYM ANTI TRANS REL
SYM-REL
trans
anti trans
neither
EQ REL
SYM A.TRANS
SYM-REL
trans
anti trans
neither
SYM A.TRANS REL

Figure 5.4: Node reached in classifying *married*.

statements. Instead, the system uses exploratory introduction both as explained in the last section and to extend the knowledge acquisition done by classification. For example, in a problem containing *married* which does not state that it is symmetric, classifying the concept acquires this property because there is a specialization of **relation** for symmetry. However, classifying *married* will not acquire the constraint that at most two people are married to each other because no specialization of **relation** captures a constraint like that.

The system always assumes the a problem is incomplete and does exploratory introduction whether or not a specialized representation captures all the stated constraints on a concept because classifying a concept so introduced may uncover additional missing information. Therefore, whenever a leaf node is reached, if there are associated introduction rules, they are tried without regard to the problem statement.

Classification of a concept introduced in this manner may yield statements that constrain the concept it was introduced for. The statements of these new constraints are reformulated in terms of the original concepts because the system is going to compare alternative concepts based on the statements that each leaves uncaptured. For example, the FAMILIES problem is stated in terms of *child* and is missing the constraint on the number of parents of a family member. Classifying *child* does not uncover this missing constraint. However, an exploratory rule introduces the concept *parents* for *child* and classifying it does uncover this constraint because the range elements of *parents* are sets and **set** has a specialization that captures size constraints. When *parents* is classified, the following statement is added to the formulation of the problem in terms of *parents*:

$$\forall x \exists y \exists z [y \in parents(x) \land z \in parents(x) \land y \neq z \Rightarrow parents(x) = \{y, z\}].$$

Versions of a statement added in this way must be translated into the alternate formulations being investigated. This is done by returning to the logical definition of the concept that is being classified and using this definition to derive a rewrite rule

"that goes in the other direction." For example, when the statement above is added to the problem, the logical definition of *parents* in terms of *child*,

$$\forall x \forall y [x \in parents(y) \Leftrightarrow child(x, y)],$$

is used to derive a rule that rewrites the newly acquired statement to an equivalent statement in terms of *child*. The result is

$$\forall x \exists y \exists z [child(y, x) \wedge child(z, x) \wedge y \neq z \wedge child(w, x) \Rightarrow w = y \vee w = z].$$

5.3.2 Comparing Alternative Formulations

Decisions to choose from amongst alternative concepts require a comparison of the relative costs of a problem formulated in terms of those concepts. The cost of a concept is the cost of any extra machinery needed to enforce the constraints that classification of the concept did not capture. When comparing two alternative concepts in a problem, the statements left uncaptured by each representation are collected and the cost of enforcing the constraints of those statements is calculated. The concept whose left over constraints are the least expensive to enforce is preferred.

Consider, for example, the comparison of the two concepts *married* and *spouse*; the two alternative formulations of the statements are shown in figure 5.5 and figure 5.6.

Irreflexive:	$\forall x \neg married(x, x)$
Symmetric:	$\forall x \forall y [married(x, y) \Leftrightarrow married(y, x)]$
Antitransitive:	$\forall x \forall y \forall z [married(x, y) \wedge married(y, z) \Rightarrow \neg married(x, z)]$
Size constraint:	$\forall x \forall y \forall z [married(x, y) \wedge married(x, z) \Rightarrow y = z]$

Figure 5.5: A set of statements expressing constraints on *married*.

Irreflexive:	$\forall x \; x \neq spouse(x)$
Symmetric:	$\forall x \forall y [x = spouse(y) \Leftrightarrow y = spouse(x)]$
Antitransitive:	$\forall x \forall y \forall z [x = spouse(y) \wedge z = spouse(y) \Rightarrow x \neq spouse(z)]$
Size constraint:	$\forall x \forall y \forall z [x = spouse(y) \wedge x = spouse(z) \Rightarrow y = z]$

Figure 5.6: Statements expressing constraints on *married* formulated in terms of *spouse*

Married is classified as an irreflexive, symmetric, antitransitive relation. The resultant representation, MARRIED, captures all but the size constraint. *Spouse* is classified as a partial one-to-one function (that is its own inverse). SPOUSE captures the size and symmetry constraints (symmetry is captured because *spouse* is known to be its own inverse). [1]

Estimating the Costs of Statements

In order to compare these two formulations, the representation design system must compare the cost of the procedures that capture the size constraint on *married* with the cost of the procedures that capture the irreflexivity and antitransitivity constraint on *spouse*.

Operationalization captures a statement's constraint by writing procedures that enforce that constraint. A reasonable measure of the cost of a statement is the sum of the complexities of the procedures that operationalization writes. The cost of a concept can be measured by summing the costs of the statements mentioning that concept that are left uncaptured by classification.

An obvious way to estimate the costs of statements is to call on operationalization to generate the procedures necessary to capture those statements, then estimate the complexity of each procedure. This is inappropriate for two reasons. First, operationalization is expensive and using it as part of the comparison procedure can be very wasteful: many of the procedures it generates will be discarded. Second, the representation design system does not need to compute costs precisely: It only needs crude estimates. For example, it does not matter what the exact cost of the size constraint is in the formulation above in terms of *married*. What does matter

[1] Recall that the representation design system determines which statements are captured by capture verification. For example, to determine that the size constraint is captured, capture verification creates a problem situation in which an anonymous individual X is the SPOUSE of another individual Y and X is the SPOUSE of Z. A procedure associated with `1-1 function` forces Y to be equal to Z. Thus, the consequent is true and the statement is captured.

is that the size constraint is more expensive to enforce in terms of *married* than the antitransitivity constraint in terms of *spouse*.

The representation design system uses a method that estimates the cost of a statement directly from the statement and the representations associated with the concepts in that statement. The method exploits a number of properties of operationalization to make reasonable relative estimates. Specifically, the method tries to ensure that if two statements S_1 and S_2 have actual costs $C(S_1)$ and $C(S_2)$ and $C(S_1) \leq C(S_2)$ then $E(S_1) \leq E(S_2)$, where E is the estimated cost.

We now discuss the cost estimation process beginning with some examples. Operationalization generates procedure that have a consistent form. I have developed a procedure that uses observations about this form (about to be discussed) in estimating complexity. The procedure is described after these observations.

The first example of cost estimation uses the procedure that operationalization generates for the following statement:

$$\forall x \forall y [R(x,y) \Rightarrow R(y,x)].$$

The procedure enforces the statement's constraint by making sure that whenever a fact of the form $R(x,y)$ is added to a problem situation, $R(y,x)$ is also added. Let us suppose that R (the representation of R) maintains a list of the ordered pairs of individuals that are related by R in a problem situation. The procedure that operationalizes the above statement watches for the addition of a new pair $< x, y >$ to a situation and responds by adding the pair $< y, x >$.

We can compute the cost of this procedure on an average problem by supposing that n distinct pairs will be added to R. This procedure will be called for each distinct pair $< x, y >$ and will add $< y, x >$ if it is a new pair. In order to determine that $< y, x >$ is a new pair, the list of existing pairs must be searched. In the average case there are $O(n)$ pairs in the list, so the cost of adding the pair is $O(n)$. The procedure is executed $O(n)$ times, hence, the total cost is $O(n^2)$.

As another example, consider the procedure that operationalization will generate for the symmetry constraint on *spouse*:

$$\forall x \forall y [x = spouse(y) \Leftrightarrow y = spouse(x)].$$

This is operationalized in much the same way as the first example: whenever an individual x is made the spouse of another individual y, the procedure makes y the spouse of x. However, making one individual the spouse of another is a constant time operation because an individual has only one spouse. Therefore, unlike the procedure for the symmetry constraint on *married*, this procedure does not have to search through a list of pairs to make sure that it is adding a new fact. Thus the total cost of this procedure is $O(n)$.

These two examples illustrate that the cost of a procedure generated by operationalization depends on the costs of the operations associated with the literals in a statement. For example, the operations associated with adding and checking literals of the form $R(x, y)$ are more expensive than the same operations associated with $x = F(y)$ (when F is an individual valued function).

A final example will illustrate that the complexity of a procedure generated by operationalization depends on the number of literals in the statement it operationalizes. Consider the procedure that operationalization will generate for the antitransitivity constraint on *married*:

$$\forall x \forall y \forall z [married(x, y) \wedge married(y, z) \Rightarrow \neg married(x, z)].$$

Each time a new pair $< x, y >$ is added to a problem situation, the procedure must check the consequent for every pair of the form $< x, z >$ related by R in the problem situation (i.e., it must check that x and z are not related by R). $O(n)$ work is required to search the list of existing pairs and checking the consequent requires $O(n)$ work. Hence, the average call to this procedure requires $O(n^2)$ work. Since it is called n times on average, the average amount of work done by the procedure in building a problem situation is $O(n^3)$.

In general, for a statement of the form

$$\phi_1 \Rightarrow \cdots [\phi_n \Rightarrow (\psi_1 \wedge \cdots \wedge \psi_m)],$$

we can think of the procedure that operationalization generates as the following form

WHEN ϕ_1 DO
  FOR EACH ϕ_2 DO
    ⋮
    FOR EACH ϕ_n DO
      ψ_1
      ⋮
      ψ_m

For example, for the statement

$$\forall x \forall y[R(x,y) \Rightarrow (R(y,z) \Rightarrow R(x,z))],$$

we can think of the procedure that operationalization generates as

WHEN $R(x,y)$ DO
  FOR EACH $R(y,z)$ DO
    ADD $R(x,z)$

This procedure is read, "when a new pair $< x, y >$ is added to the MARRIED list in a problem situation, do for each existing pair of the form $< y, z >$, add the new pair $< x, z >$."

Because of this nested loop structure, the cost of a statement of the form

$$\phi_1 \Rightarrow \cdots [\phi_n \Rightarrow (\psi_1 \wedge \cdots \wedge \psi_m)]$$

is estimated as

$$\prod_{i=1}^{n} E(\phi_i) \times max_{j=1}^{m} E(\psi_j),$$

where E is a function that estimates the cost of a literal based on the instance associated with the concept in that literal.

The system's procedure for estimating costs is given values for E and plugs these

into the above equation. The values of E are computed by table lookup based on the representation of the concept mentioned in a literal. For example, $E(married(x, y))$ is computed by looking up `relation` in the estimation table. The value stored there is n.

The cost calculation procedure can also estimate costs for several variations of implication normal form. For example, a statement that is an implication whose antecedent and consequent are both a conjunction of literals is equivalent to a structurally similar statement in implication normal form. For example, the statement

$$\forall x \forall y \forall z [married(x, y) \wedge married(x, z) \Rightarrow y = z]$$

is equivalent to

$$\forall x \forall y \forall z [married(x, y) \Rightarrow (married(x, z) \Rightarrow y = z)].$$

Accepting variations allows cost estimates to be computed without translating most statements to implication normal form.

Estimating the Cost of A Concept

Now that we have a way of estimating the cost of individual statements, we turn to estimating the cost of a concept. This requires estimating the cost of a collection of statements. Note that this is not necessarily the sum of the costs of each statement because sometimes capturing statements of lower complexity has the indirect effect of capturing statements of higher complexity. Therefore, the representation design system uses its complexity calculations to operationalize statements with low complexity and then checks to see if the extended representations capture any of the statements with higher complexity.

Once a cost estimate has been established for a collection of alternative concepts, the concept with the lowest estimate is perfered. If two concepts have the same cost

estimate. the one that is closer to an initial concept is preferred. [2] For example, if *spouse* and *married* turned out to have the same cost, *married* would be preferred because it is an initial concept. *spouse* is one introduction away from an initial concept.

Let us now return to the example earlier of two formulations of a set of statements: one in terms of *married* and one in terms of *spouse*. The two formulations were shown in figure 5.5 and figure 5.6. MARRIED left the size constraint in figure 5.5 uncaptured and SPOUSE left the irreflexivity and antitransitivity constraints in figure 5.6 uncaptured.

The cost estimate for the size constraint in the formulation using *married* is $O(n^2)$. The cost estimate for a literal of the form $x = spouse(y)$, where y is a constant, is $O(C)$ because spouse sets have at most one individual in them. The cost estimate for irreflexivity in terms of *spouse* is $O(n)$. The cost estimate for antitransitivity in terms of *spouse* is $O(n^2)$[3]

Next the representation design system operationalizes the irreflexivity constraint on *spouse* and runs capture verification on the antitransitivity constraint which it now finds to be captured. Therefore, the final estimate for *spouse* is $O(n)$ and, since the estimate for *married* is $O(n^2)$, *spouse* is preferred to *married*.

5.3.3 Formulations Including More than One Alternative Concept

Decisions about which concepts to keep after exploratory introductions are further complicated by the fact that better problem formulations can often be found by including more than one alternative concept. To allow the representation design

[2] This choice is arbitrary: a choice has to be made and computationally it does not matter which concept is chosen. The system chooses this way because doing so tends to keep problem statement looking more like it did initially. This made it easier for me to understand what was going on.

[3] This assumes that a problem situation contains a list of all the family members because without it, the second conjunction of the antitransitivity statement can not be checked.

system to find such formulations, the comparison procedure is generalized to consider all subsets of a set of alternative concepts: it then chooses the subset of alternatives that has the lowest cost estimate.

When two subsets with a different number of concepts have the same cost estimate, the set with fewer concepts is preferred on the grounds of economy of mechanism. When two subsets with the same cost estimate have the same number of concepts in them, the set whose concepts are closer to the initial concepts is preferred. When these two rules do not establish a preference between two subsets, one is chosen arbitrarily.

Normally the logical relationship between two alternative concepts is kept implicitly in the rewrite rules that are used to generate one formulation from another. For example, the equivalence between *spouse* and *married* is kept in the rewrite rule that generates the formulation in terms of *spouse* from the formulation in terms of *married*. However, when a formulation contains two alternative concepts and either of them was introduced by the representation design system, a statement expressing the relationship between the alternative concepts must be added to the formulation. For example, the representation design system must add the statement,

$$\forall x \forall y [married(x,y) \Leftrightarrow y = spouse(x)],$$

to a formulation that includes both *married* and *spouse*.

The general procedure for computing the cost of a set of more than one alternative concept first computes the cost of each concept in isolation. Then in an effort to compute the cost of the set of concepts, it identifies those statements left uncaptured by all the concepts in each set. Next it determines the minimum cost of each of those statements, which requires comparing the costs of the alternative formulations of each statement. Finally it sums the minimum costs of the uncaptured statements and adds to that value the cost estimates for the procedures required to enforce the constraints between the alternative concepts.

To illustrate this general comparison procedure, let us return to the example above and compare the cost of $\{married, spouse\}$ to $\{married\}$ and $\{spouse\}$. All the statements in the example are captured by the combination of *married* and *spouse*. Thus the cost so far is zero. Then the statement,

$$\forall x \forall y [x = spouse(y) \Leftrightarrow married(x, y)],$$

is added to the formulation (because the formulation includes both concepts). The cost of this statement is $O(n)$. Therefore, the total cost of $\{married, spouse\}$ is $O(n)$. Since the cost of *spouse* alone is also $O(n)$, the formulation in terms of *spouse* alone is preferred.

A case where it is more cost effective to keep multiple alternative concepts is illustrated by the extended classification of the *child* relation in a problem whose relevant statements are shown in figure 5.7.

Irreflexivity: $\forall x \neg child(x, x)$

Antisymmetry: $\forall x \forall y [child(x, y) \Rightarrow \neg child(y, x)$

Antitransitive: $\forall x \forall y \forall z [child(x, y) \wedge child(y, z) \Rightarrow \neg child(x, z)$

Size constraint: $\forall x \forall y \forall z \forall w [child(y, x) \wedge child(z, x) \wedge y \neq z \wedge child(w, x) \Rightarrow w = y \vee w = z]$

A mixed constraint: $\forall x \neg child(A, x)$

Figure 5.7: Statements relevant to *child*.

Child is first classified; when this completes two exploratory introductions occur. One introduces the concept *parents*, a function from people to their sets of parents. The other introduces *children*, a function from people to their sets of children. After these are classified, it turns out that the size constraint in figure 5.7 is captured by PARENTS and the mixed constraint is captured by CHILDREN. It also turns out that even though keeping both of these in the representation requires maintaining the relationship between them, this solution is still more cost effective than keeping any single concept.

The classification effort begins with the *child* relation, which is classified as irreflexive, antisymmetric and antitransitive. Then capture verification determines that the size

constraint and the mixed constraint are uncaptured, so the relevant leaf node is checked for introduction rules. There are two: one introduces *children*, the other introduces *parents*. *Children* is a function from family members to sets of the form $\{y : child(x, y)\}$. The collection of sets of this form is called *child-set*. Note that the rule that introduces this is the same rule that introduced *spouses* for *married*. The reason that two exploratory introductions were not tried for *married* is that the node for irreflexive, symmetric, antitransitive relation has only one introduction rule associated with it. This reflects the fact that there is no point in trying both rule because the collections they create are equivalent.

The two introductions cause two new formulations of the problem to be generated. The formulation in terms of *children* is shown in figure 5.8 and the formulation in terms of *parents* is shown in figure 5.9.

Figure 5.8 provides an example of the rewriting system at work. The form of the mixed constraint shown there is actually the result of first reformulating

$$\forall x \neg child(A, x)$$

as

$$\forall x\ x \notin children(A)$$

and then rewriting this statement as

$$children(A) = \emptyset.$$

Irreflexivity:	$\forall x\ x \notin children(x)$
Antisymmetry:	$\forall x \forall y [y \in children(x) \Rightarrow x \notin children(y)]$
Antitransitive:	$\forall x \forall y \forall z [y \in children(x) \wedge z \in children(y) \Rightarrow z \notin children(x)]$
Size constraint:	$\forall x \forall y \forall z \forall w [(x \in children(y) \wedge x \in children(z) \wedge y \neq z \wedge x \in children(w)) \Rightarrow w = y \vee w = z]$
A mixed constraint:	$children(A) = \emptyset$

Figure 5.8: Example problem rewritten in terms of *children*

In comparing the concepts *child*, *children*, and *parents*, the system cre-

Irreflexivity:	$\forall x\ x \notin parents(x)$
Antisymmetry:	$\forall x \forall y[x \in parents(y) \Rightarrow y \notin parents(x)]$
Antitransitive:	$\forall x \forall y \forall z[x \in parents(y) \wedge y \in parents(z) \Rightarrow \neg x \notin parents(z)]$
Size constraint:	$\forall x \forall y \forall z \forall w[(y \in parents(x) \wedge z \in parents(x) \wedge y \neq z \wedge w \in parents(x)) \Rightarrow w = y \vee w = z]$
A mixed constraint:	$\forall x\ A \notin parents(x)$

Figure 5.9: Example problem rewritten in terms of *parents*

ates formulations for {*child, parents*}, {*child, children*}, {*parents, children*}, and {*child, parents, children*}. Initially, the formulations for the subsets with more than one concept contain the constraints between the concepts in the subset. The context for {*child, children*} contains the single statement:

$$\forall x \forall y[y \in children(x) \Leftrightarrow child(x, y)],$$

which is just the logical definition of *children*. The context for {*child, parents*} contains the single statement:

$$\forall x \forall y[x \in parents(y) \Leftrightarrow child(x, y)].$$

The initial context for {*children, parents*} contains the single statement:

$$\forall x \forall y[y \in children(x) \Leftrightarrow x \in parents(y)].$$

Initially, the formulation for {*children, child, parents*} contains the three constraints,

$$\forall x \forall y[x \in parents(y) \Leftrightarrow married(x,y)]$$
$$\forall x \forall y[y \in children(x) \Leftrightarrow married(x,y)]$$
$$\forall x \forall y[y \in children(x) \Leftrightarrow x \in parents(y)]$$

The extended classification effort proceeds by classifying *parents* and *children*. The details of this are given later. Of interest for the current example are the four definitions that result from the classification efforts:

COLLECTION CHILD-SET OF **disjoint-set**(FAMILY)
CHILDREN: **partial-function**(FAMILY,CHILD-SET)
COLLECTION PARENT-SET OF **fixed-size-disjoint-set**(2,FAMILY)
PARENTS: **function**(FAMILY,PARENT-SET)

To decide which concepts to keep, the system first estimates the cost associated with

the statements left uncaptured by each concept.

The CHILD relation captures all but the size constraint and the mixed constraint. The cost of the size constraint is estimated as $O(n^3)$. Operationalization can not generate a procedure for the mixed constraint, so we will say that its cost is infinite.

The PARENTS function (along with PARENT-SET) only captures the size constraint. The cost of the irreflexivity constraint is estimated as $O(n)$. The cost of antitransitivity is estimated as $O(n^2)$ as follows. The procedure is executed n times to build the average problem situation. Each time it must check to see if y is a parent of any individual in the existing problem situation. If we assume that there are $O(n)$ individuals in the problem situation (which is reasonable since there have been $O(n)$ additions of a fact of the form $x \in parents(y)$), then this check takes $O(n)$ time. If y is an element of a parent set, then the consequent is performed and since parent sets have a fixed size, this operation takes constant time to perform. Therefore, this procedure takes, on average, $O(n)$ time to perform and is performed n times, giving a total complexity of $O(n^2)$. Again the cost of the mixed constraint is infinite.

Initially, the CHILDREN function (along with CHILD-SET) captures only what was the mixed constraint in the other formulations. Irreflexivity is estimated to be $O(n)$, antisymmetry is estimated as $O(n^2)$, and both antitransitivity and the size constraint are estimated as $O(n^3)$.

Next the uncaptured statements are systematically operationalized starting with statements with lower cost estimates. Each time the statements of some complexity level are operationalized, capture verification is rerun to see if any of the statements of higher complexity have been captured. This process does not change the cost estimate of *children*, so it remains $O(n^3)$.

The next step is for the representation design system to compare the cost of all subsets of the three concepts that have finite cost estimates. There are three such subsets in this example: *children*, $\{children, parents\}$, and $\{child, children\}$. The total cost

of $\{children, parents\}$ is determined to be $O(n)$ as follows. The only statements left uncaptured by both concepts are irreflexivity, antisymmetry, and antitransitivity. Irreflexivity is linear in both formulations. Antisymmetry is linear when expressed in terms of *parents* and quadratic when expressed in terms of *children*. Therefore, the cost of this constraint for $\{children, parents\}$ is considered to be linear. Antitransitivity is quadratic in both conceptualizations, however, it is captured by the PARENTS once irreflexitivity is operationalized. To show this I argue that assuming antitransitivity to be false in a situation in which parent sets are disjoint and *parents* is irreflexive causes a contradiction. Consider a situation in which antitransitivity is false, i.e., $x \in parents(y)$, $y \in parents(z)$, and $x \in parents(z)$. In this situation, x and y are both parents of z. But parent sets are disjoint and therefore since x is a parent of y and x and y are both parents of z, x is a parent of himself. This contradicts the irreflexivity of *parents*.

Finally the cost of the constraint between *parents* and *children*, i.e.,

$$\forall x \forall y [x \in parents(y) \Leftrightarrow y \in children(x)],$$

is estimated to be $O(n)$.

The cost of $\{child, children\}$ is determined, in a similar manner, to be quadratic since neither concept captures the size constraint.

Finally the costs of the three alternatives are compared: $\{children\}$ has cost $O(n^2)$, $\{child, children\}$ has cost $O(n^2)$, and $\{children, parents\}$ has cost $O(n)$. Thus the $\{children, parents\}$ alternative is chosen.

Figure 5.10 shows the collection of statements associated with this alternative. The statements in this figure are those whose cost estimates were used to calculate the total cost of the alternative. Note that since irreflexivity and antisymmetry have the same cost estimates in both conceptualizations, one conceptualization is chosen arbitrarily.

Irreflexivity:	$\forall x\ x \notin parents(x)$
Antisymmetry:	$\forall x \forall y[x \in parents(y) \Rightarrow y \notin parents(x)]$
Antitransitive:	$\forall x \forall y \forall z[x \in parents(y) \wedge y \in parents(z) \Rightarrow \neg x \notin parents(z)]$
Size constraint:	$\forall x \forall y \forall z \forall w[(y \in parents(x) \wedge z \in parents(x) \wedge y \neq z \wedge w \in parents(x)) \Rightarrow w = y \vee w = z]$
A mixed constraint:	$children(A) = \emptyset$
Interconcept constraint:	$\forall x \forall y[y \in children(x) \Leftrightarrow x \in parents(y)]$

Figure 5.10: Formulation in terms of both *parents* and *children*.

5.4 Other Types of Introduction

There are a few introduction rules used in the current version of the system that are not exploratory. I have not yet been able to develop as clear a picture of these rules as I have for exploratory rules. More research is necessary to develop a better understanding of them.

One of these introduction rules can be viewed as adding new concepts that aid in simplification of problem statements (figure 5.11).

IF	C is being classified
AND	COLLECTION C OF **fixed-size-set**(1,C1)
AND	F: **function**(C2,C)
THEN	introduce the function f' as,
	$\forall x \forall y[x = f'(y) \Leftrightarrow x \in f(y)]$
AND	introduce the representation F' as,
	F': **function**(C2,C1)

Figure 5.11: Rule that introduces a function into individuals when an existing function is into sets of size 1.

The rule is attached to the **fixed-size-set** node in the type taxonomy. It is paraphrased:

> "If the collection C is being classified, it is represented as a collection of sets of size 1, and there is a representation for a function F mapping into C, then introduce a new function f' mapping directly into the individuals in these sets."

An example of this rule's use is given in the next section during the classification

of *spouse-set*. Since non-empty spouse sets all have size one, the rule introduces a function, which we call *spouse*, from an individual to that individual's spouse.

One reason this is not an exploratory introduction is that it is attached to a non-leaf node in the taxonomy. In the current system, when we reach this node in the taxonomy, the rule is tested, and if its precondition is met, the concept that was being classified is replaced by the new concept. Unlike exploratory introduction, there is no comparison of alternative formulations. For example, when *spouse* is introduced, the problem is reformulated in terms of it and statements referring to *spouses* are discarded. Perhaps with a more fine grained cost model, i.e., one able to discriminate between

$$\forall x \forall y [x \in spouses(y) \Leftrightarrow y \in spouses(x)]$$
$$\forall x \forall y [x = spouse(y) \Leftrightarrow y = spouse(x)],$$

a more principled approach could be used for this type of introduction.

The second non-exploratory introduction rule in the current system is attached to the node **disjoint-set**. When the members of a collection are found to be disjoint sets, this rule introduces a function from the members of those sets to those sets. For example, members of *parent-set* are found to be disjoint and so this rule introduces a function from individuals to the parents set that an individual is a member of. For instance, if A and B are parents of C, then $parent\text{-}set(A) = \{A, B\}$.

This rule is not exploratory because the concept it introduces is not an alternative for any existing concept. Yet the concept is usually a useful adjunct to a representation. For example, in the FAMILIES problem it provides direct access from an individual to the parent set he/she is a member of. Introducing *parent-set* in the FAMILIES problem reformulates the statement,

$$\forall x \forall y \forall c [x \in parents(c) \wedge y \in parents(c) \wedge x \neq y \Rightarrow x \in couple(y) \wedge x \neq y]$$

as

$$\forall x \forall y \forall c [parent\text{-}set(x) = parents(c) \wedge parent\text{-}set(y) = parents(c) \wedge x \neq y \Rightarrow x \in couple(y) \wedge x \neq y].$$

From this statement, the simplifier produces

$$\forall x \forall y \forall c[\textit{parent-set}(y) \neq \perp \Rightarrow \textit{parent-set}(x) = \textit{parent-set}(y) \wedge x \neq y \Rightarrow x \in \textit{couple}(y) \wedge x \neq y].$$

The symbol $\perp$ is a special symbol that the system uses in the logic to indicate that there is no image of y under *parent-set*.

The above statement is then simplified to

$$\forall x \forall y \forall c[\textit{parent-set}(y) \neq \perp \Rightarrow \textit{parent-set}(y) = \textit{couple}(y)].$$

5.5 Extended Classification of Married

This section illustrates the extended classification process with the classification of *married*. We begin with a summary of the main steps in the process.

After *married* is classified as an irreflexive, symmetric, antitransitive relation, the function *spouses* is introduced and classified. During this classification, the concept of non-empty spouse sets is introduced. Then *spouses* is replaced by a partial function, called *non-empty-spouses*, from family members to non-empty spouse sets. Next, since members of the collection *non-empty-spouse-set* all have size one, the concept *spouse* is introduced. This is a function from family members to family members. *Spouse* is classified as a partial one-to-one function. It is also determined to be its own inverse. This causes the collection *married-couple* to be introduced. Members of this collection are sets of size two of family members that are married to each other.

5.5.1 Introduction of Spouses

As discussed in the last chapter, *married* is classified as a symmetric, antitransitive relation, placing us at the shaded node in figure 5.4. As noted earlier, the system finds and applies the rule associated with this node which is shown in figure 5.12.

The result of applying this rule is to add to the problem formulation the following

IF		R is being classified
	AND	R: **relation**(S,S).
THEN		introduce F_R as,
		$\forall x \forall y[y \in F_R(x) \Rightarrow R(x,y)]$
	AND	introduce the collection *R-left-proj* as,
		SORT($F_R(x)$, *R-left-proj*)
	AND	introduce R-LEFT-PROJ as,
		COLLECTION R-LEFT-PROJ OF sets
	AND	introduce F_R as,
		F_R: **function**(S,R-LEFT-PROJ)

Figure 5.12: Introduction rule associated with the node reached classifying *married*.

definitions:

$\forall x \forall y[y \in spouses(x) \Leftrightarrow married(x,y)]$
SORT(*spouses*(x), *spouse-set*)
COLLECTION SPOUSE-SET OF **set**(FAMILY)
SPOUSES: **function**(FAMILY,SPOUSE-SET).

The logical definition above is used to generate a formulation of the problem in terms of *spouses*. The original formulation of the problem is shown in figure 5.13 and the formulation in terms of *spouses* is shown in figure 5.14.

$married(N,P)$
$\forall x \forall y \exists c[child(x,c) \wedge child(y,c) \wedge x \neq y \Rightarrow married(x,y)]$
$\forall x \forall y[married(x,y) \Leftrightarrow married(y,x)]$
$\forall x \forall y \forall z[married(x,y) \wedge married(y,z) \Rightarrow married(x,z)]$

Figure 5.13: Statements from FAMILIES problem relevant to *child*

$P \in spouses(N)$
$\forall x \forall y \exists c[child(x,c) \wedge child(y,c) \wedge x \neq y \Rightarrow y \in spouses(x)]$
$\forall x \forall y[y \in spouses(x) \Leftrightarrow x \in spouses(y)]$
$\forall x \forall y \forall z[x \in spouses(y) \wedge y \in spouses(z) \Rightarrow x \notin spouses(z)]$

Figure 5.14: Statements from FAMILIES problem rewritten in terms of *spouses*

5.5.2 Classification of Spouses

Next, classification of *spouses* is initiated. This requires prior classification of *spouse-set* because the collection a concept is defined over must be classified before the concept. Since members of the collection *spouse-set* are sets, classification

begins at the **set** node in the collection taxonomy. The first question is whether the members of the collection all have a fixed size. The representation design system can not find any information about this in the problem statement, so it asks the following question:

> **Do sets of the form {y| married(x,y)} all have the same size?**
> No.

Continuing down in the taxonomy, the next question is whether all members have the same maximum size. Again the system asks.

> **Do sets of the form {y|married(x,y)} all have the same maximum size?** Yes, 1.

Next is the question of whether their are empty spouse sets:

> **Can sets of the form {y| married(x,y)} be empty?** Yes.

At this point classification has reached the leaf node for empty members (figure 5.15) which is checked for introduction rules. The system finds the rule shown in figure 5.16 which can be paraphrased:

> "When a collection S of sets is being classified and some it contains empty members, introduce a partial function into the non-empty members of S."

The rule introduces a collection whose members are non-empty spouse sets with the definition,

$$\forall x \forall y [x = \textit{non-empty-spouses}(y) \Leftrightarrow x = \textit{spouses}(y) \land \textit{spouses}(y) \neq \emptyset].$$

This causes the formulation of the problem shown in figure 5.17 to be generated and classification of *non-empty-spouse-set* now begins. The first question that classification raises is whether individuals of this collection all have the same size. The representation design system deduces that they all have size one from the following three facts:

1. *non-empty-spouse-set* was derived from *spouse-set*
2. members of the collection *spouse-set* have maximum size one

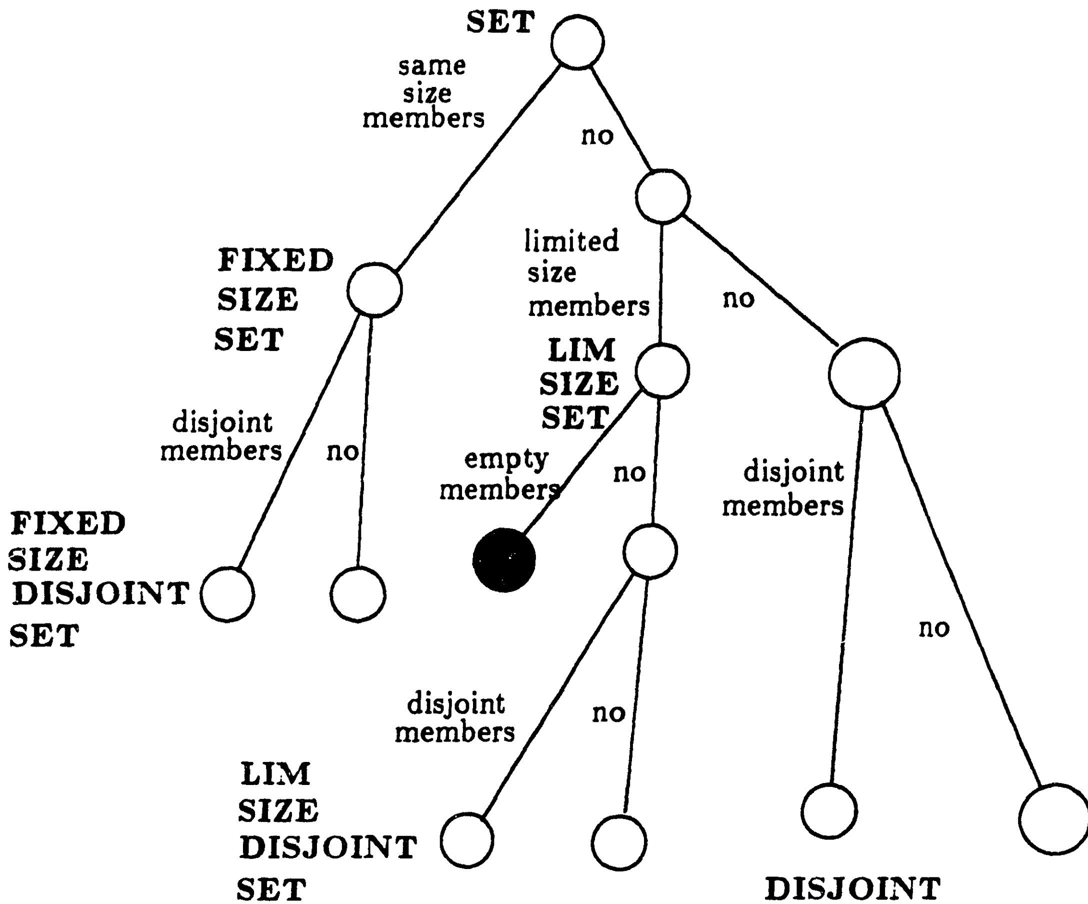

Figure 5.15: Node reached while classifying *spouse-set*

3. members of the collection *non-empty-spouse-set* are non-empty

5.5.3 Introduction of Spouse

This classification effort has now reached the node for fixed size sets (shown shaded in figure 5.18). The simplifying introduction rule of figure 5.11 is associated with this node. The rule applied in this case introduces the function *spouse* and rewrites the problem in terms of it. The rewritten version of the problem is shown in figure 5.6.

Notice that this rule would not have been applied if members of the collection

IF S is being classified
AND COLLECTION S OF **set**(S)
AND SORT(f(x),S)
AND F: **function**(S1,S),
THEN introduce the concept *non-empty-f* as,
$\forall x \forall y [y = non\text{-}empty\text{-}f(x) \Rightarrow y = f(x) \wedge f(x) \neq \emptyset]$
$\forall x \forall y [non\text{-}empty\text{-}f(x) = _ \Rightarrow y = f(x) \wedge f(x) = \emptyset]$
AND introduce the collection *non-empty-S* as.
SORT(*non-empty-f(x)*, *non-empty-S*)
AND introduce NON-EMPTY-S as,
COLLECTION NON-EMPTY-S OF S
AND introduce the representation NON-EMPTY-S as,
NON-EMPTY-F: **function**(S1,NON-EMPTY-S)

Figure 5.16: Rule introducing a collection of sets with non-empty members.

$P \in non\text{-}empty\text{-}spouses(N)$
$\forall x \forall y \exists c [child(x, c) \wedge child(y, c) \wedge x \neq y \Rightarrow y \in non\text{-}empty\text{-}spouses(x)]$
$\forall x \forall y [y \in non\text{-}empty\text{-}spouses(x) \Leftrightarrow x \in non\text{-}empty\text{-}spouses(y)]$
$\forall x \forall y \forall z [x \in non\text{-}empty\text{-}spouses(y) \wedge y \in non\text{-}empty\text{-}spouses(z) \Rightarrow x \notin non\text{-}empty\text{-}spouses(z)]$

Figure 5.17: The problem rewritten in terms of *non-empty-spouses*

non-empty-spouse-set had any other size but one. In that case, NON-EMPTY-SPOUSE-SET would be redefined in terms of `fixed-size-set`.

5.5.4 Classification of Spouse

The function *spouse* is classified as individual valued, partial, and one-to-one. The system deduces that *spouse* is partial from the fact that *non-empty-spouses* is. It deduces that *spouse* is one-to-one from the following problem statement, derived by the introduction process,

$$\forall x \forall y [x = spouse(y) \Leftrightarrow y = spouse(x)].$$

It can also tell from this statement that *spouse* is its own inverse.

This classification effort terminates at the node for a partial one-to-one function (shaded in figure 5.19).

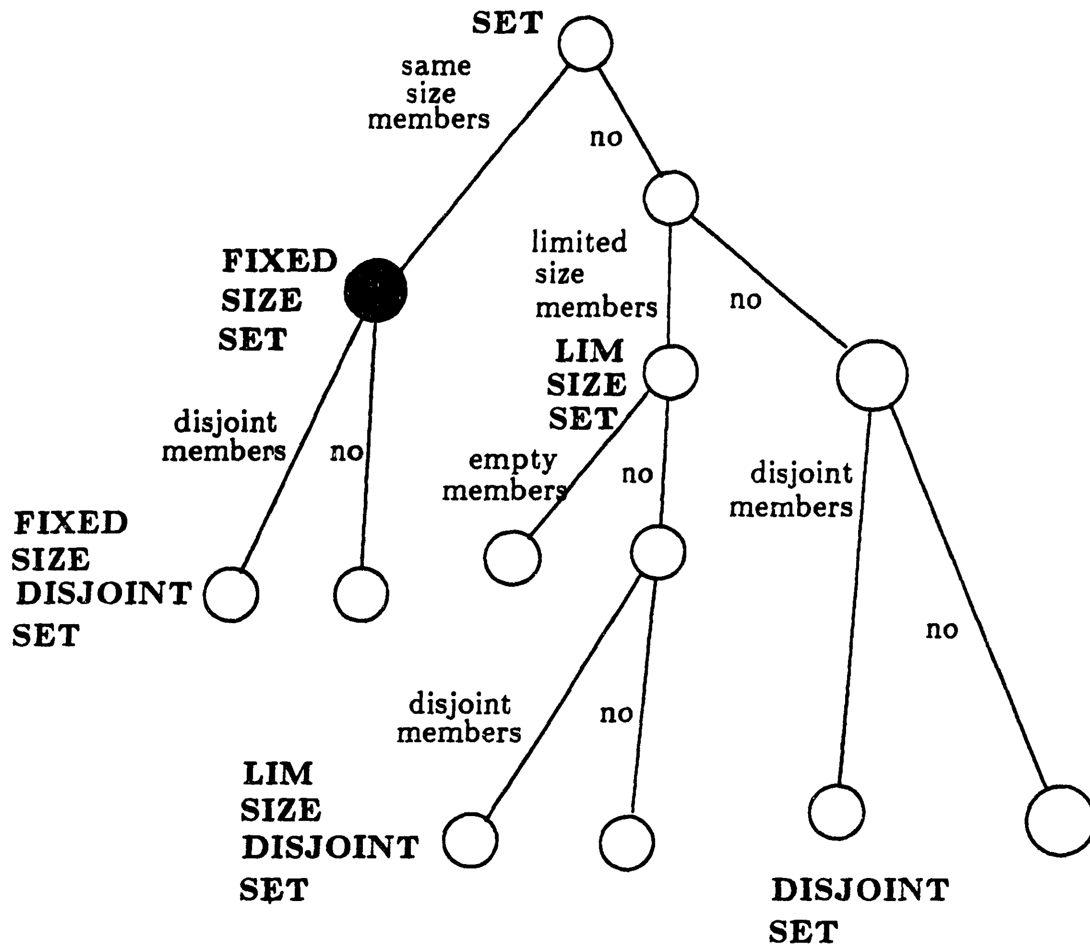

Figure 5.18: Collection taxonomy with node for fixed size set shaded.

5.5.5 Introduction of Couple

The node for partial function that is not one-to-one has the exploratory introduction rule shown in figure 5.20 associated with it. This rule can be paraphrased:

> "If a function is classified as one-to-one and it is its own inverse, introduce the collection of sets of size two which are the domain range pairs of the function. Also introduce a function from an individual into its pair."

This is done because the combination of the function being one-to-one and it being its own inverse means that the sets so introduced are disjoint from each other.

This introduces the concept of married couple and produces the formulation of the problem shown in figure 5.21.

All of the statements in this formulation, except the one mentioning *child*, are captured by the combination of the representation COUPLE and the collection of the range elements of *couple*.

The next step is the classification of *couple* and its range collection. The classification of the range collection is trivial since the associated representation is already defined to have fixed sized disjoint members. The classification of *couple* yields the following definition

COUPLE: **partial-function**(FAMILY,MARRIED-COUPLE).

Couple is determined to be partial from the fact that *spouse* is.

The final step in this example is to compare the alternative formulations: when *couple* is compared with *spouse*, the system determines that *couple* is cheaper; *couple* is also found to be cheaper than *married*.

One final note. The fact that *couple* is partial is not used in this problem. However, suppose the problem also contained the statement, "A is not married," i.e.,

$\forall x \neg married(A, x)$.

When *spouses* is introduced, the following form of this statement is added to the problem,

$spouses(A) = \emptyset$.

Then when *non-empty-spouses* is introduced, the following new variant statement is produced,

$non\text{-}empty\text{-}spouses(A) = \perp$,

which is eventually rewritten as

$couple(A) = \perp$.

When a problem situation is created from a statement like this one, the library type **partial-function** records that the domain element involved has no image under

this instance. Then if an attempt it made to assign *non-empty-spouses*(A) a value, a contradiction is signalled. [4]

5.5.6 Summary of Extended Classification of Married

Recall that before the extended classification of *married* our running example problem was as shown in figure 5.22. Also recall the following statements are added by knowledge acquisition during the classification of *married* (and before introducing *spouses*):

$$\forall x \neg married(x, x)$$
$$\forall x \forall y [married(x, y) \Leftrightarrow married(y, x)]$$
$$\forall x \forall y \forall z [married(x, y) \land married(y, z) \Rightarrow \neg married(x, z)].$$

For the sake of clarity in the example of the extended classification of *married* given above, we assumed that the these statements were already in the problem. We also assumed that the statement of the size constraint was present. Note that the size constraint is not present in figure 5.22 and is not acquired during classification of *married*. After *spouses* is introduced, the classification of *spouse-set* uncovers the size constraint.

Figure 5.23 shows the state of the problem after extended classification of *married* and figure 5.24 shows the specialized representation. The overall effect of this has been to acquire missing constraints and to capture several statements with the specialized representation of *couple*. The captured statements are enclosed in a box in figure 5.23.

5.6 Extended Classification of Child

Section 5.3.2 presented an example classifying *child*. This section takes up the extended classification of *parents* and *children*, concepts introduced during the classification of *child*. In addition to presenting another example of extended classification,

[4] Or if *non-empty-spouses*(A) already has a value when this statement is added to a problem situation, a contradiction is signalled.

this section illustrates that there can be interaction between the classification of closely related concepts. This can affect classification and knowledge acquisition. In this example, we will see that the classification of members of the collection *parent-set* as disjoint affects the classification of and knowledge acquisition for *child-set*.

We begin with a summary. The *child* relation is classified as irreflexive, antisymmetric, and antitransitive. The node reached as a result of this has two introduction rules associated with it which introduce *parents* and *children*. Since the concepts are introduced at the same node, *parents* is arbitrarily chosen to be classified first. The classification of *parent-set* (the collection of the range elements of *parents*) as having fixed size disjoint individuals.

Recall that when the two new concepts are introduced, a formulation of the problem in terms of both of them is also introduced. This formulation contains a statement expressing the relationship between *parents* and *children*. The representation design system uses this relationship and the fact that members of the collection *parent-set* are disjoint to deduce that individuals of type *child-set* are also disjoint. Eventually the functions *parents'* and *children'* are introduced. *Parents'* is a one-to-one function from *child-set* to *parent-set*, while *children'* is a one-to-one function from *parent-set* to *child-set*. These two are found to be inverses of each other. These discoveries contribute to the preference for a formulation of the problem in terms of both *parents'* and *children'*. The details of this example follow.

5.6.1 Classification of Parents

Recall that *parents* is a function mapping family members to their sets of parents. Figure 5.25 gives the problem statement rewritten in terms of *parents*. Before *parents* can be classified, *parent-set* must be because the collections that a concept is defined over must be classified before the concept. This begins at the top of the collection taxonomy. Members of the collection *parent-set* are sets so the system asks whether

parent sets have a fixed size (yes, everyone has two parents).

Continuing to follow the taxonomy down, we come to the question of whether parent sets are disjoint (yes). This places classification at the node for fixed size disjoint sets (shown shaded in figure 5.26). The simplifying introduction rule shown in figure 5.27 is associated with this node. This rule can be paraphrased as:

> "For a collection of disjoint sets, create a function mapping an individual into the unique set in that collection that the individual is an element of."

This is done by finding a function that the system already knows about that maps into the collection and defining the new function in terms of the existing function. In this case, the rule introduces the function *parent-set-of* a mapping from a family member to the parent set he/she is a member of. Along with this introduction is a rule that rewrites terms of the form $x \in parents(y)$ to the form $parent\text{-}set\text{-}of(x) = parents(y)$. The result of rewriting the problem is shown in figure 5.28.

This rule also rewrites the constraint between *parents* and *children* in the formulation of the problem involving both. This statement was

$$\forall x \forall y [x \in children(y) \Leftrightarrow y \in parents(x)].$$

It is rewritten to

$$\forall x \forall y [x \in children(y) \Leftrightarrow parent\text{-}set(y) = parents(x)].$$

This completes the classification of *parent-set*, so the representation design system returns to the classification of *parents* which now classified as a function that is not one-to-one.

5.6.2 Introduction of Children'

Classification of *parents* ends at the node for a partial function that is not one-to-one (shown shaded in figure 5.29). This node has the exploratory introduction rule shown in figure 5.30 associated with it. We can paraphrase this rule as:

"If a function maps into a collection of disjoint sets, then introduce the collection of sets of domain individuals that map to the same member of the range collection. By definition, the sets in this new collection are also disjoint. Therefore, introduce a 1–1 function from the original range collection to members of the new collection."

In the current case, the function *parents* maps into the disjoint collection *parent-set*. So the rule introduces the collection *children-collection* whose members are sets of family members with the same parents. It then introduces a 1–1 function that we call *children'* which maps parent sets to children sets such that if an individual x is a member of the parents of y, then the *children-collection* of y equals the *children'* of the $parent\text{-}set\text{-}of(x)$, i.e.,

$$\forall x \forall y [y \in children'(x) \Leftrightarrow x = parents(y)].$$

As usual, this definition is used as a rewrite rule that generates a new formulation of the problem. This formulation is shown in figure 5.28. It also generates a new version of the statement,

$$\forall x \forall y [x \in children(y) \Leftrightarrow parent\text{-}set\text{-}of(y) = parents(x)].$$

(Recall that this is the result of rewriting the constraint between *parents* and *children*, done when the function *parent-set* was introduced.) The new version of this statement is

$$\forall x \forall y [x \in children(y) \Leftrightarrow x \in children'(parent\text{-}set\text{-}of(y))].$$

This states that the children of an individual are the same as the children of the parent set of the individual. This means that the representation design system has introduced another version of the *children* function. An important inference is made from this fact: the range of *children'* is determined to be the same as the range of *children* (i.e., the *child-set* = *children-collections*). From this fact, the representation design system chooses one of the collections to be the representative for both of them. Note that, because of this, the representation design system now knows that members of the collection *child-set* are disjoint.

5.6.3 Classification of Children'

The representation design system now begins the classification of *children'* which, as usual, requires prior classification of its range. This has just been determined to be *child-set*. Thus, the system begins with the classification of *child-set*. Members of this collection do not have a fixed size nor do they have the same maximum size. However, they are disjoint.

Just as in the classification of *parent-set*, when members of the collection *child-set* are determined to be disjoint, the representation design system uses the rule shown in figure 5.27 to introduce the function *child-set*, a function that maps each family member to the child set he is a member of. This concept is introduced with the following definition,

$$\forall x \forall y[\mathit{child\text{-}set}(x) = \mathit{children}(y) \Leftrightarrow x \in \mathit{children}(y)].$$

Also, because the range of *children'* is *child-set*, the following rewrite rule is introduced

$$\forall x \forall y[\mathit{child\text{-}set}(x) = \mathit{children}'(y) \Leftrightarrow x \in \mathit{children}'(y)].$$

These two rules rewrite the problem statement,

$$\forall x \forall y[x \in \mathit{children}(y) \Leftrightarrow x \in \mathit{children}'(\mathit{parent\text{-}set\text{-}of}(y))]$$

as

$$\begin{aligned}\forall x \forall y[\mathit{child\text{-}set}(x) &= \mathit{children}(y) \Leftrightarrow \\ &\mathit{child\text{-}set}(x) = \mathit{children}'(\mathit{parent\text{-}set\text{-}of}(y))]\end{aligned}$$

This completes the classification of *child-set*; the classification of *children'* can now be completed. It is a partial one-to-one function.

Since *children'* was an exploratory introduction from *parents*, a cost analysis of both concepts is performed. The estimates come out the same, so for the moment the system records a preference for *parents* (because *parents* is closer to an initial concept than *children'*).

5.6.4 Classification of Children

Now classification of *children* is initiated (note that *child-set* has already been classified). The concept *children* is a partial function (because child sets can be empty) that is not one-to-one. This places classification at the node for a partial function that is not 1–1 (shown shaded in figure 5.29) where we have been before. Recall that there is an exploratory introduction associated with this node that introduces a one-to-one function *parents'* with the following definition,

$$\forall x \forall y[y \in \mathit{parents}'(x) \Leftrightarrow x = \mathit{children}(y)].$$

This is used to rewrite the problem statement

$$\begin{aligned}\forall x \forall y[\mathit{child\text{-}set}(x) &= \mathit{children}(y) \Leftrightarrow \\ &\mathit{child\text{-}set}(x) = \mathit{children}'(\mathit{parent\text{-}set\text{-}of}(y))]\end{aligned}$$

as

$$\begin{aligned}\forall x \forall y[y \in\ & \mathit{parents}'(\mathit{child\text{-}set}(x)) \Leftrightarrow \\ & \mathit{child\text{-}set}(x) = \mathit{children}'(\mathit{parent\text{-}set\text{-}of}(y))]\end{aligned}.$$

Recall that earlier the collection *parent-set* was introduced. Members of this collections are sets of parents. The rule that introduced *parent-set* also introduced a partial function *parent-set-of*, mapping a family member into the *parent-set* he/she is a member of. The rewrite rule derived from the definition of *parent-set-of* now rewrites the statement above as

$$\begin{aligned}\forall x \forall y[\ & \mathit{parent\text{-}set\text{-}of}(y) = \mathit{parents}'(\mathit{child\text{-}set}(x)) \\ & \Leftrightarrow \mathit{child\text{-}set}(x) = \mathit{children}'(\mathit{parent\text{-}set\text{-}of}(y))]\end{aligned}.$$

This statement is recognized by classification as defining *parents'* and *children'* as inverses. Recall that earlier the system recorded a preference for *parents* over *children'*. Now the system has determined that *children'* is the inverse of the concept *parents'* and that a formulation in terms of both of these is preferable to *parents*. This is the final result of the extended classification of *child*. It was first reformulated in terms of *parents* and *children* and then these two concepts were reformulated as *parents'* and *children'* and this formulation is preferred.

5.6.5 Summary of Extended Classification of Child

This section summarizes the effect of extended classification of *child* on our running example problem. The state of the problem after extended classification of *married* is shown in figure 5.31.

Knowledge acquisition during classification of *child* adds the following statements:

$\forall x \neg child(x,x)$
$\forall x \forall y [child(x,y) \Rightarrow \neg child(y,x)$
$\forall x \forall y \forall z [child(x,y) \wedge child(y,z) \Rightarrow \neg child(x,z)$.

Parents and *children* are introduced; knowledge acquisition during classification of *parents* adds the statement

$$\forall x \forall y \forall z \forall w [y \in parents(x) \wedge z \in parents(x) \wedge y \neq z \wedge w \in parents(x).$$

The extended classification continues yielding the problem statement shown in figure 5.32 and the specialized representation shown in figure 5.33. The two boxes enclose the statements captured by COUPLE, PARENT', and CHILDREN'.

5.7 Deriving New Mixed Constraints

Chapter 3 explained that when a concept has a restriction on it, an representation included for that concept even if it is not primitive.[5] For example, the presence of the statement $\forall x \neg brother(M,x)$ in the FAMILIES problem causes an instance to be defined for *brother* even though it is defined in terms of *sibling* and *male*. Representations are included for these concepts because restrictions are captured by reformulating during extended classification and only concepts with representations get classified.

The system identifies concepts with restrictions by looking for mixed statements of certain forms in a problem. This approach will not necessarily identify all the concepts

[5]Recall that a restriction is a mixed statement that restricts the number of individuals that can stand in some relation to a specific individual.

with restrictions on them, because a problem statement can imply a restriction on a concept and not include a mixed statement involving that concept. For example, the statements

$\forall x \neg sibling(M, x)$
$\forall x \forall y [brother(x, y) \Leftrightarrow sibling(x, y) \wedge male(y)]$

imply a restriction on *brother* even though neither is a mixed statement involving *brother*.

The representation design system detects restrictions that are not explicit in a problem when problem statements are reformulated. For example, the presence of the first statement above will cause the problem to be reformulated in terms of the function *siblings*, a mapping from an individual to its set of siblings. This will cause the second statement to be rewritten to

$$\forall x \forall y [brother(x, y) \Leftrightarrow y \in siblings(x) \wedge male(y)].$$

When the concept *siblings* is determined to be preferable to *sibling*, the system introduces a new restriction for *brother* which, in turn, causes *brother* to be classified.

The idea behind identifying new restrictions is that any time a concept is reformulated to remove a restriction (i.e., turn it into a specific statement), the system adds restrictions for any concepts that define subsets of the original concept. For example, $brothers(x)$ is a subset of the $siblings(x)$.

5.8 Detecting Redundant Introductions

It is common for more than one introduction rule to introduce the same concept during a design. For instance, recall the example in section 5.3.2 where the concepts *children* and *parents* were introduced for *child*. It turns out that there is an exploratory introduction rule associated with the node for functions that are not 1–1 (i.e., the shaded node in figure 5.29). This rule, shown in figure 5.34, introduces a different incarnation of *parents* when classification of *children* reaches the node

in figure 5.29. In addition, it introduced a different incarnation of *children* when classification of *parents* reaches that node.

The representation design system will not name the two incarnations of *children* (or *parents*) the same and will not know initially that they denote the same concept.

Formally we can define two concepts, c_1 and c_2, to be *equivalent* just in case all the general statements that refer to c_1 follow from the general statements that refer to c_2 and vice versa. Of course, this definition assumes that the problem statement is complete.

It would not be difficult conceptually to check the equivalence of introduced concepts by interpreting this definition literally. This is because the sets of general statements referring to equivalent introduced concepts will be identical except for the name. However, as a practical matter, it is too costly to check for equivalence in this way.

A simple scheme is used to detect equivalence between two introduced concepts. A derivation is kept for each introduced concept giving the sequence of introductions done to arrive at the concept. The representation design system detects the equivalence of two introduced concepts by comparing their sequences.

A set of equivalence reductions is defined over these sequences so that all sequences will be transformed into a unique shortest sequence. This shortest sequence is the canonical member of an equivalence class of sequences denoting an equivalence class of introduced concepts. Two concepts are equivalent if their canonical sequences are equal. For example, the derivation given for the version of *children* introduced directly from *child* is,

(left-proj *child*)

and the derivation of the version introduced from *parents* is,

(swap right-proj *child*)

One reduction states that (swap right-proj) should be reduced to (left-proj). This

reduces the second derivation of *children* so that it is equal to the first.

Of course, this technique does not help with the more difficult issue of detecting equivalence between two concepts appearing in the initial problem.

5.9 Chapter Summary

This chapter has explained how concept introduction is used to extend classification by introducing new concepts giving classification new ways to view existing concepts. Introduction rules are attached to nodes in the classification taxonomies and when these nodes are reached the rules are used. They introduce new concepts, defining them in terms of the concept that was being classified.

The most important (and best understood) kind of introduction is exploratory introduction. These rules are attached to leaf nodes in the taxonomy and are, therefore, used when the classification of a concept completes. If analytical reasoning problem statements were complete, then the system would only have to use these rules if uncaptured statements remained that mention the concept being classified. However, since these problems are incomplete, exploratory introductions are always tried because classification the concepts they introduce can uncover additional missing constraints.

The concepts added to a problem by exploratory introduction are alternatives for the concepts they are defined from. The system classifies the new concepts and compares the alternative representations designed for them. This comparison is based on cost estimates for the statements left uncaptured by the alternative representations. The cost of a statement is computed from estimates of the complexity of procedures generated by operationalization that capture the statement.

Figure 5.19: Node reached in classifying *spouse*.

IF	F is being classified
AND	F: **1-1-function**(S,S)
AND	F is its own inverse,
THEN	introduce the function f' as,
	$\forall x \forall y [f'(y) = f'(x) \wedge x \neq y \Rightarrow x = f(y)]$
AND	introduce the collection f'-*pair* as,
	SORT($f'(x)$, f'-*pair*)
AND	introduce **F'-PAIR** as,
	COLLECTION **F'-PAIR** OF **fixed-size-disjoint-set**(2,S)
AND	introduce the representation **F'** as,
	F': **function**(S,**F'-PAIR**)

Figure 5.20: Introduction rule associated with the 1-1 function node.

$$couple(P) = couple(N) \wedge P \neq N$$
$$\forall x \forall y \exists c [child(x,c) \wedge child(y,c) \wedge x \neq y \Rightarrow couple(x) = couple(y) \wedge y \neq x]$$
$$\forall x \forall y [couple(x) = couple(y) \wedge y \neq x \Leftrightarrow couple(y) = couple(x) \wedge x \neq y]$$
$$\forall x \forall y \forall z [(couple(y) = couple(x) \quad \wedge x \neq y \wedge couple(z) = couple(y) \wedge y \neq z) \Rightarrow couple(z) \neq \{x, z\} \vee x = z]$$

Figure 5.21: The problem rewritten in terms of *couple*

sort(P, $FAMILY$-$MEMBER$), sort(Q, $FAMILY$-$MEMBER$),
sort(R, $FAMILY$-$MEMBER$), sort(S, $FAMILY$-$MEMBER$)
$grandchild(Q, S)$
$\forall x \; child(P, x) \Leftrightarrow x = R$
$married(Q, P)$
$\forall x \forall y [grandchild(x,y) \Leftrightarrow \exists z (child(x,z) \wedge child(z,y))]$
$\forall x \forall y [child(x,y) \Leftrightarrow parent(y,x)]$
$\forall x \forall y \forall c [child(x,c) \wedge child(y,c) \wedge x \neq y \Rightarrow married(x,y)]$
$\forall x \forall y \forall c [married(x,y) \wedge child(x,c) \Rightarrow child(y,c)]$
Query: find-all x: $parent(S, x)$

Figure 5.22: A small problem about families.

$sort(P, FAMILY\text{-}MEMBER)$, $sort(Q, FAMILY\text{-}MEMBER)$,
$sort(R, FAMILY\text{-}MEMBER)$, $sort(S, FAMILY\text{-}MEMBER)$
$grandchild(Q, S)$
$\forall x\ child(P, x) \Leftrightarrow x = R$
$couple(P) = couple(Q) \wedge P \neq Q$
$\forall x \forall y[grandchild(x, y) \Leftrightarrow \exists z(child(x, z) \wedge child(z, y))]$
$\forall x \forall y[child(x, y) \Leftrightarrow parent(y, x)]$
$\forall x \forall y \exists c[child(x, c) \wedge child(y, c) \wedge x \neq y \Rightarrow couple(x) = couple(y) \wedge y \neq x]$
$\forall x \forall y \forall c[couple(y) = couple(x) \wedge y \neq x \wedge child(x, c) \Rightarrow child(y, c)]$

$\forall x \neg[couple(x) = couple(x) \wedge x \neq x]$
$\forall x \forall y[couple(x) = couple(y) \wedge y \neq x \Leftrightarrow couple(y) = couple(x) \wedge x \neq y]$
$\forall x \forall y \forall z[(couple(y) = couple(x) \wedge x \neq y \wedge couple(z) = couple(y) \wedge y \neq z) \Rightarrow couple(z) \neq \{x, z\} \vee x = z]$

Query: find-all x: $parent(S, x)$

Figure 5.23: A small problem about families.

```
COLLECTION* FAMILY-MEMBER OF unique-individual
CHILD:  relation(FAMILY-MEMBER,FAMILY-MEMBER)
PARENT:  relation(FAMILY-MEMBER,FAMILY-MEMBER)
COLLECTION MARRIED-COUPLE OF
fixed-size-disjoint-set(2,FAMILY-MEMBER)
COUPLE:  partial-function(FAMILY-MEMBER,MARRIED-COUPLE)
```

Figure 5.24: Representation of example problem after extended classification of *married*.

Irreflexivity:	$\forall x\ x \notin parents(x)$
Antisymmetry:	$\forall x \forall y[x \in parents(y) \Rightarrow y \notin parents(x)]$
Antitransitive:	$\forall x \forall y \forall z[x \in parents(y) \wedge y \in parents(z) \Rightarrow \neg x \notin parents(z)]$
Size constraint:	$\forall x \forall y \forall z \forall w[y \in parents(x) \wedge z \in parents(x) \wedge y \neq z \wedge w \in parents(x) \Rightarrow w = y \vee w = z]$
A mixed constraint:	$\forall x\ A \notin parents(x)$

Figure 5.25: Example problem rewritten in terms of *parents*

Figure 5.26: Node reached in classifying *parent-set*.

IF the collection C is being classified
AND COLLECTION S OF `disjoint-set`(S1)
AND SORT($f(x), S$),
THEN introduce the function *map-S* as,
$\forall x \forall y[map\text{-}S(x) = f(y) \Leftrightarrow x \in f(y)]$
AND introduce the representation MAP-S as,
MAP-S: `function`(S1,S)

Figure 5.27: Rule that, given a collection of disjoint sets, introduces a function mapping an individual to the unique disjoint set it is a member of.

Irreflexivity: $\forall x\; \textit{parent-set}(x) \neq \textit{parents}(x)$

Antisymmetry: $\forall x \forall y [\textit{parent-set}(x) = \textit{parents}(y) \Rightarrow \textit{parent-set}(y) \neq \textit{parents}(x)]$

Antitransitive: $\forall x \forall y \forall z [\textit{parent-set}(x) = \textit{parents}(y) \wedge \textit{parent-set}(y) = \textit{parents}(z) \Rightarrow \textit{parent-set}(x) \neq \textit{parents}(z)]$

Size constraint: $\forall x \forall y \forall z \forall w [\textit{parent-set}(y) = \textit{parents}(x) \wedge \textit{parent-set}(z) = \textit{parents}(x) \wedge y \neq z \wedge \textit{parent-set}(w) = \textit{parents}(x) \Rightarrow w = y \vee w = z]$

A mixed constr: $\forall x\; \textit{parent-set}(A) \neq \textit{parents}(x)$

Figure 5.28: Example problem rewritten in terms of *parent-set*

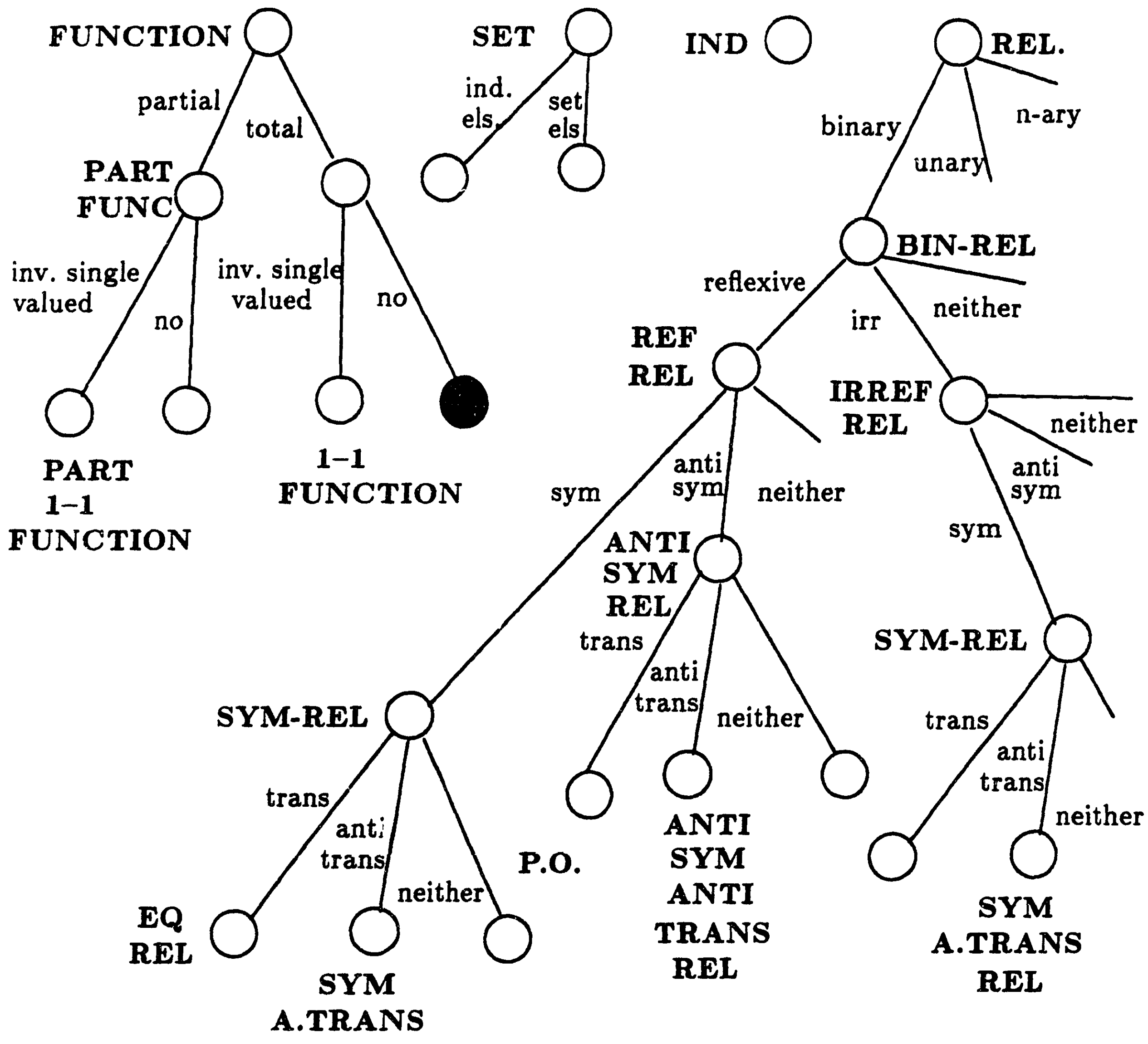

Figure 5.29: The collection taxonomy with the node reached in classifying *parents* shaded.

IF		f is being classified
	AND	F: **function**(S1.S2)
	AND	COLLECTION S2 OF **disjoint-set**(S3).
THEN		introduce the 1–1 function f' as,
		$\forall x \forall y [y \in f'(x) \Rightarrow x = f(y)]$
	AND	introduce the collection *s1-collection* as,
		SORT($f'(x)$, *s1-collection*)
	AND	introduce type S1-COLLECTION as,
		COLLECTION S1-COLLECTION OF **disjoint-set**(S1)
	AND	introduce the representation F' as,
		F': **function**(S2,S1-COLLECTION)

Figure 5.30: Rule introducing 1–1 function for a function that is into disjoint sets.

sort($P, FAMILY\text{-}MEMBER$), sort($Q, FAMILY\text{-}MEMBER$),
sort($R, FAMILY\text{-}MEMBER$), sort($S, FAMILY\text{-}MEMBER$)
$grandchild(Q, S)$
$\forall x\ child(P, x) \Leftrightarrow x = R$
$couple(P) = couple(Q) \land P \neq Q$
$\forall x \forall y [grandchild(x, y) \Leftrightarrow \exists z(child(x, z) \land child(z, y))]$
$\forall x \forall y [child(x, y) \Leftrightarrow parent(y, x)]$
$\forall x \forall y \exists c [child(x, c) \land child(y, c) \land x \neq y \Rightarrow couple(x) = couple(y) \land y \neq x]$
$\forall x \forall y \forall c [couple(y) = couple(x) \land y \neq x \land child(x, c) \Rightarrow child(y, c)]$

$\forall x \neg [couple(x) = couple(x) \land x \neq x]$
$\forall x \forall y [couple(x) = couple(y) \land y \neq x \Leftrightarrow couple(y) = couple(x) \land x \neq y]$
$\forall x \forall y \forall z [(couple(y) = couple(x) \land x \neq y \land couple(z) = couple(y) \land y \neq z) \Rightarrow couple(z) \neq \{x, z\} \lor x = z]$

Query: find-all x: $parent(S, x)$

Figure 5.31: State of example problem after extended classification of *married*

$\mathit{children}'(\mathit{parent\text{-}set\text{-}of}(P)) = \{R\}$
$\mathit{couple}(Q) = \mathit{couple}(P) \wedge Q \neq P$
$\exists z [\mathit{children}'(\mathit{parent\text{-}set\text{-}of}(Q)) = \mathit{child\text{-}set}(z) \wedge \mathit{children}'(\mathit{parent\text{-}set\text{-}of}(z)) = \mathit{child\text{-}set}(S)]$

$\forall x \forall y [\mathit{child\text{-}of}(x, y) \Rightarrow \mathit{parent}(y, x)]$
(*) $\forall x \forall y [\mathit{parent\text{-}set\text{-}of}(x) = \mathit{parent\text{-}set\text{-}of}(y) \Rightarrow \mathit{couple}(x) = \mathit{couple}(y)]$
(*) $\forall x \forall y [\mathit{parent\text{-}set\text{-}of}(x) \neq \perp \wedge \mathit{couple}(x) = \mathit{couple}(y) \Rightarrow \mathit{parent\text{-}set\text{-}of}(x) = \mathit{parent\text{-}set\text{-}of}(y)]$
$\forall x\; x \in \mathit{parent\text{-}set\text{-}of}(x)$
$\forall x\; x \in \mathit{couple}(x)$

$\forall x \neg [\mathit{couple}(x) = \mathit{couple}(x) \wedge x \neq x]$
$\forall x \forall y [\mathit{couple}(x) = \mathit{couple}(y) \wedge x \neq y \Leftrightarrow \mathit{couple}(y) = \mathit{couple}(x) \wedge x \neq y]$
$\forall x \forall y \forall z [\mathit{couple}(x) = \mathit{couple}(y) \wedge x \neq y \wedge \mathit{couple}(y) = \mathit{couple}(z) \wedge y \neq z \Rightarrow \mathit{couple}(x) \neq \mathit{couple}(z) \vee x = z]$
$\forall x \forall y \forall z [\mathit{couple}(x) = \mathit{couple}(y) \wedge x \neq y \wedge \mathit{couple}(x) = \mathit{couple}(z) \wedge x \neq z \Rightarrow y = z]$

$\forall x\; \mathit{child\text{-}set}(x) \neq \mathit{children}'(\mathit{parent\text{-}set\text{-}of}(x))$
$\forall x \forall y [\mathit{child\text{-}set}(y) = \mathit{children}'(\mathit{parent\text{-}set\text{-}of}(x)) \Rightarrow \mathit{child\text{-}set}(x) \neq \mathit{children}'(\mathit{parent\text{-}set\text{-}of}(y))]$

$\forall x \forall y \forall z [(\mathit{child\text{-}set}(y) = \mathit{children}'(\mathit{parent\text{-}set\text{-}of}(x)) \wedge \mathit{child\text{-}set}(z) = \mathit{children}'(\mathit{parent\text{-}set\text{-}of}(y))) \Rightarrow \mathit{child\text{-}set}(z) \neq \mathit{children}'(\mathit{parent\text{-}set\text{-}of}(y))]$

$\forall x \forall y \forall z \forall w [\mathit{child\text{-}set}(x) = \mathit{children}'(\mathit{parent\text{-}set\text{-}of}(y)) \wedge \mathit{child\text{-}set}(x) = \mathit{children}'(\mathit{parent\text{-}set\text{-}of}(z)) \wedge y \neq z \wedge \mathit{child\text{-}set}(x) = \mathit{children}'(\mathit{parent\text{-}set\text{-}of}(w)) \Rightarrow w = y \vee w = z]$

Query: find-any x: $\mathit{parent}(S, x)$

Figure 5.32: Formulation of example problem after extended classification of *child*.

COLLECTION* FAMILY-MEMBER OF unique-individual
COLLECTION PARENT-SET OF fixed-size-disjoint-set(2,FAMILY-MEMBER)
COLLECTION CHILD-SET OF disjoint-set(FAMILY-MEMBER)
PARENT-SET-OF: partial-function(FAMILY-MEMBER,PARENT-SET)
CHILD-SET-OF: function(FAMILY-MEMBER,CHILD-SET)
PARENTS': 1-1-function(CHILD-SET,PARENT-SET)
CHILDREN': 1-1-function(PARENT-SET,CHILD-SET)
PARENT: relation(FAMILY-MEMBER,FAMILY-MEMBER)
COLLECTION MARRIED-COUPLE OF
fixed-size-disjoint-set(2,FAMILY-MEMBER)
COUPLE: partial-function(FAMILY-MEMBER,MARRIED-COUPLE)

Figure 5.33: Representation of example problem after extended classification of *child*.

```
IF          F: function(T1,T2)
      AND   COLLECTION T2 OF set(T3),
THEN        introduce the new concept F' as,
            ∀x∀y[x ∈ F(y) ⇔ y ∈ F'(x)]
      AND   introduce the collection F-swap as,
            SORT(F'(x), F-swap)
      AND   introduce F-SWAP as,
            COLLECTION F-SWAP OF set(T1)
      AND   introduce the new representation F' as,
            F-SWAP: function(T3,F-SWAP)
```

Figure 5.34: Rule associated with the node for functions that are not 1–1

Chapter 6

Operationalization

Operationalization is the representation design system's way to capture the constraints of statements that classification fails to capture.

In describing how operationalization captures constraints, it is useful to view a specialized representation as a black box. Inside the box there is a collection of data structures that represent problem situations. The user of the box creates problem situations by "telling" the representation about the specifics of a problem. For example, in creating a situation for the FAMILIES problem, the user "tells" the representation that N is married to P, Q is the grandfather of S, etc.

The collection of structures inside the black box represents a problem situation because the general constraints on the problem are captured by the procedures that create and inspect the structures.

Now suppose a general problem statement is not captured by the representation. To be concrete, let us suppose that the following statement is not captured:

$$\forall x \forall y [x \in siblings(y) \Leftrightarrow y \in siblings(x)]. \qquad [1]$$

In this case, problem situations can be created in which an individual x is in the $siblings(y)$ but in which $y \notin siblings(x)$.

One way to extend the representation so that it also captures this statement is to

add a procedure that "watches" for the execution of an operation that can affect sibling set and, in response, takes any steps necessaryto maintain the constraint. For example, each time an individual x is added to the $siblings(y)$ during the creation of a problem situation. the additional action adds y to the $siblings(x)$.

In this extended representation. adding elements to sibling sets will not violate constraint [1] above. The new procedure makes it impossible to create a problem situation in which $x \in siblings(y)$ and $y \notin siblings(x)$.

Operationalization captures constraints in this way. Each constraint left uncaptured after extended classification is "turned into" one or more procedure that enforces the constraint. The number of procedures created per constraint is a function of the number of operations that can affect the constraint. For example, in addition to an operation that adds elements to sibling sets, there is another operation that allows all the members of a sibling set to be specified by an assignment, as in $siblings(A) = \{B, C\}$. To fully capture the constraint of [1], operationalization must also write a procedure that responds appropriately when this assignment operation is executed.

To capture the constraint, operationalization must do two things: (i) determine what operations of a representation can affect the constraint and (ii) write procedures extending the representation that enforce the constraint whenever one of those operations is executed.

For example, the constraint

$$\forall x \forall y [x \in siblings(y) \Leftrightarrow y \in siblings(x)]$$

is captured as follows. Operationalization first identifies the operations in the FAMILIES problem class that can affect sibling sets. The representation SIBLING-SET is defined in terms of **set**, which has two operations that get used in building FAMILIES problems: ADD-ELEMENT(y,x) (add x to the set y) and EQUATE-TO-CONSTANT(x,y) (assign the set x the value of a constant set y).

Operationalization the writes two procedures. One procedure responds to

the execution of an ADD-ELEMENT(siblings(x),y) by performing an ADD-ELEMENT(siblings(y),x). The other procedure responds to an ASSIGN-TO-CONSTANT(siblings(y),x) by performing an ADD-ELEMENT(siblings(z),y) for each z in the constant set x.

Since ADD-ELEMENT and ASSIGN-TO-CONSTANT are the only operations that can affect a sibling set, these two procedures capture the original constraint because, with them in place, it is not possible to create a problem situation in which the constraint is violated.

Thus, operationalization converts a general statement into a collection of procedures that respond to the execution of representation operations by executing other operations. Since executing an operation corresponds to adding a specific fact to a problem situation, these procedures can be thought of as compiled lemmas that draw conclusions when new specific facts are added to problem situations. These conclusions are additional specific facts that must be added to the problem situation to maintain a constraint.

6.1 Overview of the Operationalization Procedure

The problem specification that is left when classification terminates includes all the specific statements and the general statements left uncaptured. Many, if not all, of these statements are expressed in terms of sorts and concepts introduced during extended classification.

The new version of the problem is preprocessed by translating the specific statements in it into anonymous statements, then expanding out definitions. The anonymous version of the problem statement left after classification for the FAMILIES problem is shown in figure 6.1. Recall that defined concepts are not represented unless they have restrictions on them. The defined concepts expanded away are those for which

$\exists x \exists y\ couple(x) = couple(y)$
$\exists y \exists y[children'(x) = \{y\}]$

$\forall x \forall y[parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y) \Rightarrow couple(x) = couple(y)]$
$\forall x \forall y[parent\text{-}set\text{-}of(x) \neq \perp \wedge couple(x) = couple(y) \Rightarrow parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y)]$ $\forall x[parent\text{-}set\text{-}of(x) \neq \perp \Rightarrow x \in parent\text{-}set\text{-}of(x)]$
$\forall x[couple(x) \neq \perp \Rightarrow x \in couple(x)]$
$\forall x\ x \in child\text{-}set\text{-}of(x)$

Figure 6.1: Anonymous versions of the specific statements and the uncaptured general statements for the small FAMILIES problem.

the system never found such constraints. These concepts never got representations defined for them by earlier processes. The concept *grandchild* is an example of a concept appearing in the anonymous version of FAMILIES which does not appear in the representation. Its definition is used to expand the statement $\exists x \exists y\ grandchild(x,y)$ into the statement:

$$\begin{aligned}\exists x \exists y \exists z[\ & parent\text{-}set(x) = parents(child\text{-}set(z)) \wedge \\ & parent\text{-}set(z) = parents(child\text{-}set(y))]\end{aligned}$$

Let us call the result of these preprocessing steps a problem's *operationalization set.* The FAMILIES operationalization set is shown in figure 6.2.

$\exists x \exists y\ couple(x) = couple(y)$
$\exists y \exists y[children'(x) = \{y\}]$

$\forall x \forall y[parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y) \Rightarrow couple(x) = couple(y)]$
$\forall x \forall y[parent\text{-}set\text{-}of(x) \neq \perp \wedge couple(x) = couple(y) \Rightarrow parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y)]$ $\forall x[parent\text{-}set\text{-}of(x) \neq \perp \Rightarrow x \in parent\text{-}set\text{-}of(x)]$
$\forall x[couple(x) \neq \perp \Rightarrow x \in couple(x)]$
$\forall x\ x \in child\text{-}set\text{-}of(x)$

Figure 6.2: The operationalization set for the small FAMILIES problem.

The first step in the operationalization of a constraint is to generate lemmas from it. This begins by determining the set of all representation operations that can be executed in the problem class. This is done by looking at the unconditional statements the operationalization set. The literals in these statements that correspond to operations are called the *operational literals* of a problem. The literal in the statement

$\exists x \exists y\ x \in siblings(y)$

is an example of an operational literal from the FAMILIES problem. It is operational because SIBLING-SET supports the operation ADD-ELEMENT(siblings(y),x). The system knows this because associated with each procedure of a representation there is a schema for the predicate calculus literal corresponding to that operation. For example, associated with the ADD-ELEMENT procedure of **set** is a schema of the form $x \in S$. This schema is used by plugging in the term of a literal for S. For example, the above literal is recognized as operation by plugging $siblings(y)$ in for S in the schema.

Not all literals are operational because not all literals correspond to operations. For example, the statement

$\forall x \forall y[siblings(x) = child\text{-}set(x) - \{x\}]$

is an example of a literal in the FAMILIES problem that is not operational. The treatment of these literals is discussed later in this chapter.

For each conditional statement in the operationalization set and each operational literal, the system tries to simplify the statement by assuming the literal in it. If assuming the literal simplifies the statement, operationalization starts an *operationalization sequence* with new the statement:

$assumption_1 \Rightarrow simplified\text{-}statement_1$.

For example, assuming the literal $x_1 \in siblings(y_1)$ in the statement

$\forall x \forall y[x \in siblings(y) \Leftrightarrow y \in siblings(x)]$,

yields

$\forall x_1 \forall y_1[x_1 \in siblings(y_1) \Rightarrow true \Leftrightarrow y_1 \in siblings(x_1)]$,

which simplifies to:

$$\forall x_1 \forall y_1[x_1 \in siblings(y_1) \Rightarrow y_1 \in siblings(x_1)]. \quad [2]$$

Therefore, an operationalization sequence is begun with this statement.

The process of assuming a literal is repeated on the consequent of the new statement, producing a new lemma of the form:

$$assumption_1 \Rightarrow (assumption_2 \Rightarrow simplified\text{-}statement_2).$$

This is continued until a statement is produced which is a right associated sequence of implications. with the final consequent being an operational literal. For example, when operationalization tries to make an assumption in the consequent of [2] above, it finds that it is an operational literal. Therefore, operationalization of this statement is complete. A statement in this form is said to be in *operational form.*

All unconditional statements are also considered to be in operational form. Nothing is done to unconditional general statements in this step.

The second step of operationalization is to compile a procedure for each statement in operational form. Conditional statements are compiled into daemons that respond when operations are executed that correspond to their antecedents.

Unconditional specific statements are compiled by translating them into tell operations that an unconditional general statement is compiled into a daemon that responds to the creation of new individuals with the same sort as the variables in the statement. For example, the procedure generated for the statement,

$$\forall x \; x \in couple(x),$$

responds to the creation of a new couple by adding an element. For instance, suppose the statement $A \in couple(B)$ appears in a problem. Then when a problem situation is created, the tell operation ADD-ELEMENT(couple(B),A) will be executed. In response, the procedure that enforces the constraint of the above general statement executes the operation, ADD-ELEMENT(couple(B),B).

Here is an example of operationalization at work. The formula $x_1 \in siblings(y_1)$ is the declarative form of a tell operation that affects the constraint of,

$$\forall x \forall y [x \in siblings(y) \Leftrightarrow y \in siblings(x)].$$

Assuming $x_1 \in siblings(x_2)$ in this statement yields,

$$\forall x_1 \forall y_1[x_1 \in siblings(y_1) \Rightarrow y_1 \in siblings(x_1)].$$

Which is in operational form because the consequent is the declarative form of ADD-ELEMENT($siblings(x_1), y_1$).

The operational form is then compiled into the procedure:

WHEN ADD-ELEMENT(siblings(y).x) DO ADD-ELEMENT(siblings(x),y).

This example included one operationalization sequence. There is another operationalization sequence involving the same statement and the same operational literal because $x_1 \in siblings(y_1)$ can be assumed in the statement in two different ways: on the left and right side of the byconditional. In this case, both sequences end up producing the same operational form.

In general, for a constraint to be captured, every operationalization sequence begun for the statement must result in a lemma in operational form.

Before continuing to discuss each of the three steps of operationalization in more detail, two important points are discussed. The first point is that the procedure,

WHEN ADD-ELEMENT(siblings(y),x) DO ADD-ELEMENT(siblings(x),y),

does not fully capture the constraint

$$\forall x_1 \forall y_1[x_1 \in siblings(y_1) \Rightarrow y_1 \in siblings(x_1)],$$

because $x \notin siblings(y)$ is a different operation than $x \in siblings(y)$. If a problem contains the operational literal $x \notin siblings(y)$, a separate operationalization sequence is started with the statement,

$$\forall x \forall y[x \in siblings(y) \Leftrightarrow y \in siblings(x)],$$

which yields:

$$\forall x \forall y[x \notin siblings(y) \Leftrightarrow y \notin siblings(x)].$$

Often only one sense of a literal is used in a problem statement. For example, the

FAMILIES problem does not every use $x \notin siblings(y)$. Being able to detect this allows the representation design system to save effort during operationalization. This savings can be quite substantial because eliminating an operational literal from consideration allows operationalization to avoid at least one operationalization sequence for every statement whose constraint can be affected by the eliminated literal. There is also a commensurate savings in the representation machinery when an operational literal is eliminated from consideration.

Second, operationalizing a statement may uncover an operational literal that did not appear in the initial operationalization set. This can happen in two ways. First, a literal that appears only in conditional problem statements is not considered operational because we can not tell which sense of that literal will be used. Second, operationalization can derive literals that do not appear anywhere in the initial operationalization set (this is just a generalization of the first case).

As an example of how can happen, consider a problem with the following two statements in its operationalization set,

$$\forall x \forall y [x \in brothers(y) \Leftrightarrow x \in siblings(y) \wedge sex(x) = male]$$
$$\exists x \; siblings(x) = \emptyset$$

and suppose that there is no mention of the literal $brothers(x) = \emptyset$ in the problem statement. Even though there no mention of this literal, the statement $\exists x \; brothers(x) = \emptyset$ follows from the problem and is, therefore, in the problem class. Literals such as these are uncovered as a natural result of the operationalization process. For example, an operationalization sequence that begins by assuming $siblings(x) = \emptyset$ in the general statement above uncovers $brothers(x) = \emptyset$.

6.2 Identifying the Operational Literals in a Problem Class

The operational literals of a problem class are identified by applying the certain rules to the literals in the unconditional statements of a problem's operationalization set.

For this purpose, literals are divided into *special* and *normal* literals. Special literal are those whose relation symbol has a special meaning to the representation design system. The symbols $\in$ and $=$ are examples of relation symbols with special meanings. In general, special symbols correspond to operations provided by any of the library types. Thus, the reason that $\in$ has a special meaning is that the library type **set** has an operation that "puts" elements "in" sets.

The relation symbol of a special literal, is called the *operation* and one of the argument positions is distinguished as the *operation argument*. For example, in a literal whose operation is $\in$ the second argument is the operation argument. Hence, in the literal $A \in siblings(B)$, $siblings(B)$ is the operation argument.

A special literal is operational when the sort of its operation argument has an associated type defined in terms of a library type that supports the literal's operation. Hence, if $A \in siblings(B)$ is operational because the sort of $siblings(B)$ is *sibling-set* and SIBLING-SET is defined in terms of **set** which supports ADD-ELEMENT.

A normal literal is one whose relation symbol denotes a relation from a problem domain, e.g., *married*. A normal literal is operational if every one of its terms is one of the following: a constant, a variable, a term whose function symbol has an associated instance definition.

6.3 Operationalization Sequences

Operational form is actually slightly more general than the discussion thus far has indicated. A statement is in operational form when it is a right associated sequence of implications, i.e.,

$$\phi_1 \Rightarrow (\phi_2 \Rightarrow \cdots (\phi_{n-1} \Rightarrow \phi_n))$$

in which each of the ϕ_i is either an operational literal or a *test literal.* A test literal is a special literal whose operation argument is a constant. The literal $x \in S$ is

an example of a test literal. Operationalization treats $x \in S$ as a test to determine whether x is an element of S.

$\neg true \gg false$
$\neg false \gg true$
$true \wedge P \gg P$
$false \wedge P \gg false$
$true \vee P \gg true$
$false \vee P \gg P$
$true \Rightarrow P \gg P$
$P \Rightarrow true \gg true$
$false \Rightarrow P \gg true$
$P \Rightarrow false \gg \neg P$
$P \Leftrightarrow true \gg P$
$P \Leftrightarrow false \gg \neg P$

Figure 6.3: Simplification Rules

The operationalization procedure works as follows:

1. For each uncaptured statement ϕ and each operational literal that can simplify ϕ begin an operationalization sequence. Assume that there are n literals that can simplify ϕ and denote these literals by $\alpha_1, \ldots, \alpha_n$. Note that assumptions that are equalities are treated as a special case which is discussed below. In the general case, an operationalization sequence is begun by assuming α_i in ϕ and then deriving an implication for each way to make the assumption. This is done as follows:

   (a) Find all subexpressions in ϕ that unify with α_i. The result of one successful unification is a most general unifier.

   (b) For each unifier, θ, a new statement is constructed whose form is,

   $$\alpha_i[\theta] \Rightarrow \phi[\theta](\alpha_i[\theta]/true).$$

   The notation $\psi[\theta]$ stands for the result of making the substitutions given in θ in ψ and $\phi(\alpha/\beta)$ is the result of substituting β for every occurrence of α in ϕ. Thus the right hand side of the statement constructed is the result

of substituting *true* for α_i in ϕ after making the substitutions in the most general unifier θ in both ϕ and α.

(c) The resultant statement is simplified according to the rules given in figure 6.3.

2. If the new statement is in operational form, then this sequence is complete. The consequent of the new statement is either an operational literal, the constant *true*, or the constant *false*. If it is the constant *true*, then this sequence is considered complete but the new statement is discarded. The reason for this is that the only way a statement of the form,

$\phi_1 \Rightarrow \cdots (\phi_n \Rightarrow true)$

could have been produced is if the constraint,

$\phi_1 \wedge \cdots \wedge \phi_n$

is already captured. If this is the case, there is no need to include the new statement.

If the final consequent of the new statement is *false* then the compiled form of this statement will signal a contradiction whenever its conditions are true. If the final consequent is an operational literal then it is checked to see if it is new. If so, one of two things is done:

(a) If the new operational literal $\alpha_i(x_1, \ldots, x_m)$ can be used to simplify other uncaptured statements in the problem then, the add $\exists x_1, \ldots, x_m(\alpha_i)$ to the problem's operationalization set.

(b) Otherwise, if the negation of α_i is an existing operational literal, replace the consequent of the new statement with $\neg\alpha_i \Rightarrow false$.

3. Otherwise, operationalization tries to continue the sequence by making an assumption in the final consequent of the new statement. If no assumptions can be made then operationalization of ϕ fails.

4. If it finds an assumption in β in the statement.

$$\alpha_1 \Rightarrow (\alpha_2 \Rightarrow \cdots (\alpha_n \Rightarrow \beta))$$

then the sequence is continued with,

$$\alpha_1 \Rightarrow (\alpha_2 \Rightarrow ...(\alpha_n \Rightarrow (\alpha_{n+1} \Rightarrow \beta\prime))),$$

where α_{n+1} is the new assumption and $\beta\prime$ is the result of substituting *true* for α_{n+1} in β.

The above procedure is modified slightly when an assumption is an equality with a constant term, e.g., $F(x) = \varphi$. Instead of unifying the assumption with subexpressions of a statement being operationalized, unifications are found between the non-constant term ($F(x)$) and subexpressions of the statement being operationalized. Then for each unification, the simplified form of the statement's consequent has φ substituted for $F(x)$. Thus, for each unification θ found, the new formula generated for a statement of the form,

$$\alpha_1 \Rightarrow (\alpha_2 \Rightarrow \cdots (\alpha_n \Rightarrow \beta)),$$

has β replaced by,

$$F(x) = \varphi[\theta] \Rightarrow \beta(F(x)[\theta]/\varphi).$$

Here is an example in which the following statement from the FAMILIES problem is operationalized:

$$\forall x \forall y[x \in siblings(y) \Leftrightarrow y \in siblings(x)].$$

Two of the operational literals found in the operationalization set for FAMILIES can simplify this statement: $x_1 \in siblings(y_1)$ and $siblings(x_2) = \varphi$.

1. An operationalization sequence is started by assuming $x_1 \in siblings(y_1)$ in the above statement. This is done by finding unifications of the assumption with subexpressions of the statement. One such unification results in,

   $$\forall x_1 \forall y_1[x_1 \in siblings(y_1) \Rightarrow (true \Leftrightarrow y_1 \in siblings(x_1))],$$

   which is simplified to,

$$\forall x_1 \forall y_1[x_1 \in siblings(y_1) \Rightarrow y_1 \in siblings(x_1)].$$

Since this statement is in operational form, the first sequence is complete.

2. Another operationalization is begun with the other possible unification of $x_1 \in siblings(y_1)$ with a subexpression of the original statement, i.e., $y \in siblings(x)$. This results in the same statement as the first sequence.

3. Operationalization then assumes $siblings(x_2) = \varphi$ in,

$$\forall x \forall y[x \in siblings(y) \Leftrightarrow y \in siblings(x)],$$

by unifying $siblings(x_2)$ with subexpressions of the statement and then substituting φ for $siblings(x_2)$ in the statement. The result of one unification is

$$\forall x \forall x_2[siblings(x_2) = \varphi \Rightarrow (x \in \varphi \Leftrightarrow x_2 \in siblings(x))].$$

This statement is not in operational form so operationalization looks for an assumption to make in the consequent. It assumes $x \in \varphi$ [1] and derives the formula,

$$\forall x \forall x_2[siblings(x_2) = \varphi \Rightarrow (x \in \varphi \Rightarrow (True \Leftrightarrow x_2 \in siblings(x)))],$$

which is simplified as,

$$\forall x \forall x_2[siblings(x_2) = \varphi \Rightarrow (x \in \varphi \Rightarrow x_2 \in siblings(x))],$$

4. Operationalization finds another way to assume $siblings(x_2) = \varphi$ in the statement. The sequence that results from this ends in the same statement as the last sequence.

When an operationalization sequence produces a statement whose final consequent is a non-operational literal, it continues to make assumptions in an effort to simplify it to an operational literal. That is, it continues just as though the final consequent of the statement was not a literal. Unconditional general statements that appear in an

[1] It turns out that it does not have to make any assumptions for $siblings(x)$. The reason will be explained later.

operationalization set are also dealt with in this way. For example, the FAMILIES problem operationalization set contains the statement:

$$\forall x[siblings(x) = child\text{-}set(x) - \{x\}].$$

This is operationalized like any conditional general statement. Suppose, for instance, that the problem contains the operational literal $siblings(x) = \emptyset$. An operationalization sequence is started by assuming this literal in the statement which yields:

$$siblings(x) = \emptyset \Rightarrow child\text{-}set(x) - \{x\} = \emptyset.$$

The next section explains how this statement is simplified to

$$siblings(x) = \emptyset \Rightarrow child\text{-}set(x) = \{x\},$$

which is in operational form.

The current version of operationalization handles only non-operational literals that are equalities (like the example above) or membership relations. These are the only cases that come up in the problems I have studied. If a problem contains a unconditional general statement that is some other non-operational literal, the current system will fail to operationalize that statement.

6.3.1 Rewriting Intermediate Statements

The system simplifier looks for ways to simplify each statement generated in an operationalization sequence. If a statement is simplified, the operationalization sequence continues with the simplified version.

For example,

$$\forall x[siblings(x) = \emptyset \Rightarrow child\text{-}set(x) - \{x\} = \emptyset]$$

is rewritten to

$$\forall x[siblings(x) = \emptyset \Rightarrow child\text{-}set(x) = \{x\}],$$

while

$$\forall x \forall y \forall z[x = spouse(y) \wedge y = spouse(z) \Rightarrow x \neq spouse(z)]$$

can be simplified if we assume $spouse(c_1) = c_2$:

$$spouse(c_1) = c_2 \Rightarrow c_1 = spouse(z) \Rightarrow c2 \neq spouse(z)$$

Rewrite rule uses the fact that *spouse* is one-to-one to rewrite to,

$$spouse(c_1) = c_2 \Rightarrow z = spouse(c_1) \Rightarrow c2 \neq spouse(z)$$

and then to,

$$spouse(c_1) = c_2 \Rightarrow z = c_2 \Rightarrow c2 \neq spouse(z)$$

and then to,

$$spouse(c_1) = c_2 \Rightarrow c2 \neq spouse(c_2)$$

6.3.2 Operationalizing Statements that Contain Existential Quantifiers

A general statement can contain existential quantifiers in the scope of its universal quantifiers. Operationalization handles these existential quantifiers in a way that is similar to skolemization.

When a general statement contains an existentially quantified variable, operationalization replaces that variable with a new function of the universal quantifiers in the surrounding scope. For example, the existential z is removed from the following statement,

$$\forall x \forall y[grandchild(x, y) \Rightarrow \exists z(child(x, z) \wedge child(z, y))],$$

by introducing a new function of x and y, $F(x, y)$, and substituting it for z in the statement, yielding

$$\forall x \forall y[grandchild(x, y) \Rightarrow child(x, F(x, y)) \wedge child(F(x, y), y))].$$

So far this is skolemization. The representation design system also introduces a representation for the new function, e.g., assuming that x is a member of a collection

s_1 and that y is a member of a collection s_2,

```
F: function(s1,s2).
```

Then operationalization of the skolemized statement proceeds normally. For example, operationalization of the statement above yields,

$$grandchild(x,y) \Rightarrow child(x,F(x,y))$$
$$grandchild(x,y) \Rightarrow child(F(x,y),y)$$

The instances associated with the skolem functions introduced now serve to equate individuals denoted by a skolem function with the same constants as arguments. For example, suppose A is made the grandchild of B in some problem situation, i.e., an operation corresponding to $grandchild(B,A)$ is executed. When the procedure generated for the first statement above is executed it will create an anonymous individual to be $F(B,A)$. The same element will be accessed by the procedure generated for the second statement above.

In this way, the two function expressions will denote the same piece of data structure in situations created.

6.3.3 Optimizations of Operationalization

The operationalization process is very expensive to perform and can generate a large number of procedures to operationalize statements. One way the representation design system tries to minimize the use of operationalization is by capturing the constraints of statements during classification. Operationalization has one technique for reducing the number of sequences that are generated for a particular statement. It also has several techniques for detecting when statements that are the results of sequences are unnecessary. When it detects that a statement is unnecessary, no procedure is generated for it.

The technique that reduces the number of operationalization sequences generated exploits the following fact. Separate statements with the same consequents are equiv-

alent if their sequences of antecedents are permutations of each other. For example, the following two statements are equivalent:

$$P \Rightarrow Q \Rightarrow R$$
$$Q \Rightarrow P \Rightarrow R.$$

It is also the case that the procedures that operationalization will generate for these two statements are equivalent.

This fact is used to reduce the number of operationalization sequences that will be generated for a statement.

There are three techniques that the representation design system currently uses to detect that statements it has generated are unnecessary. The first technique removes statements generated by operationalization when they are subsumed by others. This method removes any statement of the form,

$$\phi_1 \Rightarrow \cdots (\phi_{n-1} \Rightarrow \phi_n),$$

if operationalization has generated a statement whose consequent is ϕ_n and whose "antecedents" are a subset of $\{\phi_1, \ldots, \phi_{n-1}\}$.

The reason this can be done is that a statement of the second form subsumes a statement of the first form. What this means with respect to operationalization is that including a procedure that is the compiled form of a statement of the first form will not change the behavior of the representation. For example, suppose operationalization of a problem has generated the two statements:

$$Q \Rightarrow R$$
$$P \Rightarrow Q \Rightarrow R.$$

After adding the procedure for the first statement to a representation, there is not reason to add a procedure for the second because it will only add R in a subset of the problem situations in which the first procedure will.

Here is another example from the operationalization of the FAMILIES problem of a statement being discarded by subsumption. The statement being operationalized is,

$$\forall x \forall y [y \in brothers(x) \Leftrightarrow y \in siblings(x) \wedge sex(x) = male].$$

There are several relevant operational literals:

$x_1 \in brothers(y_1)$
$x_2 \in siblings(y_2)$
$sex(x_3) = male$

1. Assuming $x_1 \in brothers(y_1)$ yields,

   $$\forall y_1 \forall x_1 [x_1 \in brothers(y_1) \Rightarrow x_1 \in siblings(y_1) \wedge sex(x_1) = male].$$

   This is split into the two statements,

   $\forall y_1 \forall x_1 [x_1 \in brothers(y_1) \Rightarrow x_1 \in siblings(y_1)$
   $\forall y_1 \forall x_1 [x_1 \in brothers(y_1) \Rightarrow sex(x_1) = male].$

2. Assuming $x_2 \in siblings(y_2)$ yields,

   $$\forall y_2 \forall x_2 [x_2 \in siblings(y_2) \Rightarrow x_2 \in brothers(y_2) \Leftrightarrow sex(x_2) = male].$$

   There are two assumptions that can be made in this statement, so the sequence splits into two:

   (a) Assuming $x_1 \in brothers(y_1)$ yields,

   $$\forall y_1 \forall x_1 [x_1 \in siblings(y_1) \Rightarrow x_1 \in brothers(y_1) \Rightarrow sex(x_1) = male].$$

   This statement is removed because it is subsumed by one of the results of the first sequence above.

   (b) Assuming $sex(x_3) = male$ yields,

   $$\forall y_1 \forall x_3 [x_3 \in siblings(y_1) \Rightarrow sex(x_3) = male \Rightarrow x_3 \in brothers(y_1)].$$

6.4 Generating Procedures from Statements in Operational Form

A procedure is generated for each statement in operational form. The basic idea is to compile a statement of the form,

$$\phi_1 \Rightarrow \cdots (\phi_{n-1} \Rightarrow \phi_n),$$

where all the ϕ_i are operational literals, into a sequence of daemons. Each daemon in the sequence, except the last, waits for the execution of an operation and

then creates a new daemon. The last daemon, e.g., the daemon corresponding to $\phi_{n-1} \Rightarrow \phi_n$ above responds to the execution of an operation corresponding to its antecedent (e.g., ϕ_{n-1}) by executing an operation corresponding to its consequent (e.g.. ϕ_n).

Suppose that the operation corresponding to each of the ϕ_i above is Φ_i. Then the basic form of the procedure generated for the statement above is,

WHEN Φ_1 DO
    WHEN Φ_2 DO
        ⋮
        WHEN Φ_{n-1} DO Φ_n

When an operation Φ_1 is executed, this procedure generates a new daemon of the form,

WHEN Φ_2 DO
    ⋮
    WHEN Φ_{n-1} DO Φ_n

It is a new daemon because the bindings for the arguments of Φ_1 are substituted into $\Phi_2, \ldots, \Phi_n$.

Here is a concrete example. One of the statements generated during operationalization of the FAMILIES problem statement,

$$\forall x \forall y [x \in brothers(y) \Leftrightarrow x \in siblings(y) \wedge sex(x) = male]$$

is

$$x \in siblings(y) \Rightarrow (sex(x) = male \Rightarrow x \in brothers(y)).$$

The procedure generated for this statement is,

```
WHEN ADD-ELEMENT(siblings(y),x) DO
     when ASSIGN-CONSTANT(sex(x),male) DO
          ADD-ELEMENT(brothers(y),x)
```

When a particular operation is executed that adds an individual to the siblings of another individual, say ADD-ELEMENT(siblings(B),A), this daemon responds by creating a new daemon of the form,

```
WHEN ASSIGN-CONSTANT(sex(A),male) DO
      ADD-ELEMENT(brothers(B),A)
```

The form of the procedures generated is complicated by two factors. First, some of the literals in statements in operational form are test literals. Second, a procedure like the one above must work correctly regardless of the order in which the operations of its antecedents are executed. For example, the procedure just discussed must work if the operation ASSIGN-CONSTANT(sex(A),male) is executed before ADD-ELEMENT(siblings(B),A).

The first complication is addressed as follows. Test literals are compiled into conditionals instead of daemons. For example, the literal $x = C$, where C is a constant, is compiled into the program fragment,

IF $x = C$ THEN ...

Some test literals are compiled into an iteration construct. For example, the literal $x \in \varphi$, where φ is a set constant, is compiled into the program fragment,

FOR EACH $x \in \varphi$ DO...

The problem of making procedures generated by operationalization work no matter what order the operations are executed is solved by compiling all but the first operational literal differently than has been suggested. Embedded operational literals are compiled into a test operation followed conditionally by a daemon. For instance, the procedure generated for the example above involving *siblings* and *brothers* is actually compiled into,

```
WHEN ADD-ELEMENT(siblings(y),x) DO
        IF sex(x)=male THEN ADD-ELEMENT(brothers(y),x)
        ELSE WHEN ASSIGN-CONSTANT(sex(x),male) DO
                  ADD-ELEMENT(brothers(y),x)
```

I now give an abstract version of the procedure **compile** which takes predicate calculus statements (in operational form) as input and produces procedures. For example,

compile($x \in siblings(y) \Rightarrow y \in siblings(x)$) =
    WHEN ADD-ELEMENT($siblings(y), x$) DO ADD-ELEMENT($siblings(x), y$)

The **compile** procedure is defined as:

(**com** $\alpha \Rightarrow \beta$) =
if operational(α) **then**
  '(WHEN ,(**gen-op** α) DO ,(**compile1** β))
else (**gen-test-op** α (**compile1** β))
(**compile** α) = (**map** '**gen-op** (**literals** α))

The procedure **gen-op** generates a tell operation corresponding to the operational literal it is given as an argument. For example,

(**gen-op** $x \in siblings(y)$) = ADD-ELEMENT(siblings(y),x)

The procedure **gen-test-op** generates a test operation corresponding to the test literal it is given. It also decides whether the test should be encapsulated in an IF or a FOR EACH. For example,

(**gen-test-op** $x = C\ \beta$) = '(IF x=C THEN ,(**compile** β))
(**gen-test-op** $x \in \varphi\ \beta$) = '(FOR EACH $x \in \varphi$ DO (**compile** β))

The second statement in the definition of COMPILE handles unconditional general statements **Compile** treats the first antecedent differently than embedded antecedents by employing the procedure COMPILE1 to handle all but the first antecedent. **Compile1** is defined as:

```
(com α ⇒ β) =
if operational(α) then
  `(IF ,(gen-test-op α) THEN ,(compile1 β)
   ELSE WHEN ,(gen-op α) DO ,(compile1 β))
else gen-test-op α (compile1 β))
(com α) = (gen-op α)
```

6.5 Soundness of Operationalization

Given a statement ϕ and the collection of statements generated operationalizing it Φ, we must show that $\Phi \Rightarrow \phi$. The argument proceeds as follows. First, we show that the antecedents of each operational form generated from ϕ correspond to combinations of specific facts whose addition to a situation cause ϕ to become *false*. Then we show that the system either generates an operational form for each such combination or indicates that it has failed to capture ϕ. We conclude that if ϕ is captured then no situation can be created that violates the constraint of ϕ.

Chapter 7

Representation Machinery

This chapter details how problems are solved with specialized representations of the kind generated by my representation design system. First, it explains two underlying mechanisms that the representations rely on in building problem situations. Then, it explains how the library types **relation**, **function**, **set**, and **individual** are implemented. Finally, it gives an example of a specialized representation and shows how it is used to build one problem situation.

Recall that library structures are implemented as abstract data types (ADTs). Each ADT is a prototype for creating representations. Representations of concepts are created from ADTs in the concept taxonomy by instantiation. For example, a structure that results from instantiating **relation** is used to represent a particular relation like *married*. This structure is created by the declaration

MARRIED: **relation**(FAMILY-MEMBER,FAMILY-MEMBER).

Representations of collections are created from ADTs in the collection taxonomy by defining subtypes. For example, a representation of the collection *FAMILY-MEMBER* is created by the definition

COLLECTION FAMILY-MEMBER OF **individual**.

Instances of the ADT FAMILY-MEMBER are used to represent family members like N.

7.1 The Equality System and Anonymous Individuals

Two facilities support the specialized representations generated by my system: an equality system and a mechanism that creates anonymous individuals. These are closely integrated in the process of creating situations, so they are described together.

The equality system is similar to RUP's [McAllester82]. It maintains equivalence classes of terms known to be equal and supports an operation that merges equivalence classes as new terms are found to be or asserted to be equal. It also supports a retraction mechanism similar to RUP's, but this is not used by the representations in the current implementation so it is not discussed here.

Unlike RUP, this equality system requires that each equivalence class have an object associated with it that the terms in that class denote. This object is a data type instance that represents an individual in the problem. For example, in the situation representing

$$mother(A) = B, mother(C) = mother(A),$$

the terms $mother(A)$ and $mother(C)$ are placed in the same equivalence class. The object associated with this class is B.[1]

When the object denoted by an equivalence class is unknown at the time the class is created, an object representing an anonymous individual is created and associated with the class. For example, given only the statement

$$mother(C) = mother(A),$$

an equivalence class containing $mother(C)$ and $mother(A)$ is created and an anonymous object is created and associated with the class. The type of the anonymous object (i.e., what collection it is a member of) is determined from the range of $mother$.

[1] Presently, I will explain that the object associated with this class is actually an instance of individual that represents the individual B.

The equality system contains another facility that is not included in RUP, but is a simple extension. It allows two equivalence classes to be marked to indicate that they are known to be disjoint (i.e., the individuals that the two classes denote are known to be unequal). Any attempt to equate individuals in classes so marked results in the equality system signalling a contradiction.

When the equality system merges two equivalence classes, it creates a new class which is the union of the terms in the two original classes. To determine what object to associate with the new class, the system sends a signal to the objects associated with the original classes notifying them that they are being equated. This is called an EQUATE notification. The objects must respond with a single object which the equality system associates with the new class.

Accordingly, the library ADTs that are used to create objects representing individuals are required to respond to the EQUATE notification. In the current library, the ADTs that do this are **individual**, **set**, and their specializations.

To illustrate the merging process, consider the situation created to represent

$$mother(A) = B, mother(C) = D.$$

It will contain two equivalence classes: one containing the term $mother(A)$ with the associated individual B and the other containing the term $mother(C)$ with the associated individual D. Now suppose the statement $mother(A) = mother(C)$ is added. The two classes are merged and the objects representing B and D are sent an EQUATE signal. They respond with a single object that can be referred as B or D

7.2 Implementation of Library ADTs

This section describes how the library ADTs **individual**, **relation**, **function**, and **set** are implemented. The rest of the library types are implemented as specializations

of these.

The ADT **individual** is used to represent domain individuals. It is used as a prototype to create representations of collections of individuals by creating subtypes such as

COLLECTION* FAMILY-MEMBER OF **individual**.

Individual is able to represent the fact that an individual can have more than one name. It does this with a single data field, called **name**, which is used to store a list of constants that name the individual represented by an instance. For example, the instance representing the individual B has the name field (B). **Individual** has several procedures associated with it (answers several messages). When an instance of individual receives an EQUATE message of the form

EQUATE(other-object)

it responds by returning a single instance whose name field is the union of its name field and other-object's name field. For instance, in the situation representing

$mother(A) = B$
$mother(C) = D$

there are objects representing each of the individuals B and D. When the the statement $mother(A) = mother(C)$ is added to this situation, one of these objects, say B, is sent the message

EQUATE(D)

(actually the argument is the object representing D). The response to this message is to combine the objects representing B and D and return the single object whose name field contains the list (BD).

Individual also supports an EQUAL? message of the form

EQUAL?(other-object).

When an instance receives such as message, it responds with *true* if its name field shares any elements with other-object's name field. If the instance and other-object

are associated with equivalence classes that are marked disjoint, the instance responds with *false*. Otherwise it responds with *unknown*.

In the FAMILIES problem, the different names given for individuals of sort *FAMILY* stand for different individuals. Therefore, the collection FAMILY-MEMBER is specialized to **unique-individual**. This ADT also has a **name** field, however, it can only contain a single name. An instance of **unique-individual** responds to an EQUAL? message with *true* if the names are the same or they are anonymous but are associated with the same equivalence class; *false* if they have different names or are associated with equivalence classes that are marked disjoint; and *unknown* otherwise.

An instance of **unique-individual** responds to an EQUATE message as follows. If both instances are anonymous, it does not matter which one is returned so the instance returns itself. If one of the individuals is anonymous and the other is named, then the named individual is returned. If both individuals are named and the names are different, a contradiction is signalled.

Note that the effect of equating an anonymous individual with a named individual is to name the anonymous individual. A typical scenario is that as situations get created, anonymous individuals get named.

The ADT **relation** is implemented as two lists of ordered n-tuples. Instances of **relation** are created to represent particular relations. An instance R created for a relation R contains two lists used to store n-tuples of individuals. One list in R is used to store the n-tuples of individuals known to stand in the relation R. This list is called *R-list*. The other list is used to store n-tuples of individuals known not to stand in the relation R. It is called $\overline{R}$*-list*. As a problem situation is created n-tuples get added to these lists.

The semantics of each of the library ADTs discussed in the rest of this section are given by a collection of axioms describing their behavior. The axioms for **relation** are:

$\forall \vec{x}\ [\mathcal{R}(R(\vec{x})) \Leftrightarrow element(<\vec{x}>, R\text{-}list)]$
$\forall \vec{x}\ [\mathcal{R}(\neg R(\vec{x})) \Leftrightarrow element(<\vec{x}>, \overline{R}\text{-}list)]$
$\forall \vec{x}\ [element(<\vec{x}>, R\text{-}list) \Rightarrow \neg element(<\vec{x}>, \overline{R}\text{-}list)]$
The term $\vec{x}$ stands for a list of variables, i.e.. $x_1, \ldots, x_n$.

The first axiom defines what it means for the fact $R(\vec{x})$ to be represented ($\mathcal{R}(statement)$ is read. "*statement* is represented"). It states that such a fact is represented in a situation just in case the *R-list* contains the n-tuple $<\vec{x}>$ in that situation. The third axiom states the consistency requirement that a situation can not represent both $R(\vec{x})$ and $\neg R(\vec{x})$.

Instances of the ADT `function` are used to represent functions. An instance that is created to represent a particular function, say *mother*, contains a single list to store ordered pairs of the form $< x, mother(x) >$. As a problem situation is created, ordered pairs of individuals are added to this list. Function terms can appear in problem statements that do not give the names of the individuals they denote and creating situations for these statements requires creating anonymous individuals. For example, the statement

$married(mother(A), father(B))$,

is represented by creating an anonymous family member, say x, to stand for the $mother(A)$, adding the pair $< A, x >$ to MOTHER, creating another anonymous individual, say y, adding the pair $< B, y >$ to FATHER, and then adding the pair $< x, y >$ to MARRIED.[2]

The axioms specifying the semantics of `function` are:

$\forall \vec{x}\ \forall y[\mathcal{R}(F(\vec{x}) = y) \Leftrightarrow element(<<\vec{x}>, y >, F)]$
$\forall \vec{x}\ \forall y \forall z[element(<<\vec{x}>, y >, F) \wedge element(<<\vec{x}>, z >, F) \Rightarrow y = z]$

The ADT `set` is used to represent individual sets by instantiating it and to representation collections of sets by defining subtypes of it. As an example of its use to represent a collection, consider *brother-set*, the collection of sets of brothers. It is

[2]This assumes, of course, that *married* is represented as a relation.

represented with a subtype whose instances are sets of brothers. **Set** has a data field for storing a list of the individuals known to be members of a particular set. It also has a field to indicate whether a set is closed (i.e., all the members of that set are known). It has the following procedures associated with it:

1. ADD-ELEMENT adds a new individual to a set unless the set is closed in which case a contradiction is signalled.

2. ASSIGN-TO-CONSTANT makes the list inside an instance equal to a given list and marks the instance closed. If the instance already contains elements, they must all be members of the constant set or a contradiction is signalled.

3. EQUAL? takes another set instance as an argument and returns *true* if both instances are closed sets and have the same members, returns *false* is they are closed and do not contain the same members, otherwise it returns *unknown*.

Set also answers the EQUATE message. When an instance of **set** receives a message of the form

EQUATE(other-object),

it responds with a single instance that contains the union of the original instance's members and the members of other-object unless one (or both) of the instances are closed. If one of the instances is closed, the EQUATE procedure of **set** checks that all members of the open instance are also members of the closed instance and returns the closed instance if they are, otherwise it signals a contradiction. If both instances are closed they must be equal.

Here are the axioms defining the full semantics for sets:

$\forall x \forall y[\mathcal{R}(x \in y) \Leftrightarrow element(x, y)]$

$\forall x \forall \varphi[\mathcal{R}(x = \varphi) \Leftrightarrow \forall z(z \in \varphi \Rightarrow z \in x) \wedge closed(x)]$,

where φ is a constant set

$\forall x \forall y[x = y \Rightarrow \forall z(z \in x \Leftrightarrow z \in y)]$

$\forall x \forall y \forall z[closed(x) \wedge closed(y) \wedge (z \in x \Leftrightarrow z \in y) \Rightarrow x = y]$

7.3 Solving the Example Problem

This section explains how the small FAMILIES problem is solved with the representation designed for it. The initial problem statement is shown in figure 7.1. The final problem statement is shown in figure 7.2. The boxes enclose statements in the figure captured by the specialized representation and the statements marked by (*) are captured by operationalization. The final representation is shown in figure 7.3.

SORT($P, FAMILY\text{-}MEMBER$), SORT($Q, FAMILY\text{-}MEMBER$),
SORT($R, FAMILY\text{-}MEMBER$), SORT($S, FAMILY\text{-}MEMBER$)
$grandchild(Q, S)$
$\forall x\ child\text{-}of(P, x) \Leftrightarrow x = R$
$married(Q, P)$
Query: find-all x: $parent(S, x)$

Figure 7.1: The small FAMILIES problem.

The system creates a problem situation from the three specific statements found in the final formulation of the problem. Let us illustrate how it does this beginning with the statement

$$children'(parent\text{-}set\text{-}of(P)) = \{R\}.$$

To create a situation representing this statement, an instance of FAMILY-MEMBER is created whose name is P; an instance of PARENT-SET is created, say $parent\text{-}set_1$; and P is added to it, then the pair $< P, parent\text{-}set_1 >$ is added to PARENT-SET-OF. The system has, so far, created a situation representing $parent\text{-}set\text{-}of(P)$. Next, a child-set is created, call it $child\text{-}set_1$, assigned the constant value $\{R\}$, and the pair $< parent\text{-}set_1, child\text{-}set_1 >$ is added to CHILDREN'. A diagrammatic version of the structure built for this statement is shown in figure 7.4.

Next the system creates a representation of the statement

$$couple(Q) = couple(P) \wedge Q \neq P$$

and adds it to the problem situation. To accomplish this, it creates an individ-

$children'(parent\text{-}set\text{-}of(P)) = \{R\}$
$couple(Q) = couple(P) \wedge Q \neq P$
$\exists z[children'(parent\text{-}set\text{-}of(Q)) = child\text{-}set(z) \wedge children'(parent\text{-}set\text{-}of(z)) = child\text{-}set(S)]$

$\forall x \forall y[child\text{-}of(x, y) \Leftrightarrow parent(y, x)]$
$(*)\ \forall x \forall y[parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y) \Rightarrow couple(x) = couple(y)]$
$(*)\ \forall x \forall y[parent\text{-}set\text{-}of(x) \neq \perp \wedge couple(x) = couple(y) \Rightarrow parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y)]$
$\forall x\ x \in parent\text{-}set\text{-}of(x)$
$\forall x\ x \in couple(x)$

$\forall x \neg[couple(x) = couple(x) \wedge x \neq x]$
$\forall x \forall y[couple(x) = couple(y) \wedge x \neq y \Leftrightarrow couple(y) = couple(x) \wedge x \neq y]$
$\forall x \forall y \forall z[couple(x) = couple(y) \wedge x \neq y \wedge couple(y) = couple(z) \wedge y \neq z \Rightarrow couple(x) \neq couple(z) \vee x = z]$
$\forall x \forall y \forall z[couple(x) = couple(y) \wedge x \neq y \wedge couple(x) = couple(z) \wedge x \neq z \Rightarrow y = z]$
$\forall x\ child\text{-}set(x) \neq children'(parent\text{-}set\text{-}of(x))$
$\forall x \forall y[\ child\text{-}set(y) = children'(parent\text{-}set\text{-}of(x)) \Rightarrow child\text{-}set(x) \neq children'(parent\text{-}set\text{-}of(y))]$

$\forall x \forall y \forall z[(child\text{-}set(y) = children'(parent\text{-}set\text{-}of(x)) \wedge child\text{-}set(z) = children'(parent\text{-}set\text{-}of(y))) \Rightarrow child\text{-}set(z) \neq children'(parent\text{-}set\text{-}of(y))]$

$\forall x \forall y \forall z \forall w[\ child\text{-}set(x) = children'(parent\text{-}set\text{-}of(y)) \wedge child\text{-}set(x) = children'(parent\text{-}set\text{-}of(z)) \wedge y \neq z \wedge child\text{-}set(x) = children'(parent\text{-}set\text{-}of(w)) \Rightarrow w = y \vee w = z]$

Query: find-the $parents'(child\text{-}set\text{-}of(S))$

Figure 7.2: Final formulation of the example FAMILIES problem.

COLLECTION* FAMILY-MEMBER OF **unique-individual**
COUPLE: **function**(FAMILY-MEMBER,MARRIED-COUPLE)
COLLECTION MARRIED-COUPLE OF **fixed-size-disjoint-set**(2,FAMILY)
COLLECTION CHILD-SET OF **disjoint-set**(FAMILY-MEMBER)
CHILD-SET-OF: **function**(FAMILY-MEMBER,CHILD-SET)
COLLECTION PARENT-SET OF **fixed-size-disjoint-set**(2,FAMILY-MEMBER)
PARENT-SET-OF: **partial-function**(FAMILY-MEMBER,PARENT-SET)
CHILDREN': **1-1-partial-function**(PARENT-SET,CHILD-SET)
PARENTS': **1-1-function**(CHILD-SET,PARENT-SET)

Figure 7.3: Final representation of the FAMILIES problem.

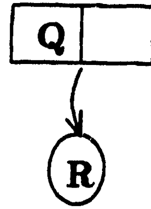

Figure 7.4: Diagram of representation for $children'(parent\text{-}set\text{-}of(P)) = \{R\}$.

ual named Q and a couple containing Q. Then, it creates another couple and adds the individual P to it. Next, it EQUATES the two couples just created produces a single couple containing P and Q. Note that since Q and P are represented as instances of `unique-individual`, the second conjunct is already represented.

Because $couple(Q) = couple(P)$, the procedure that operationalization generates to capture the constraint between couples and parent sets now makes $parent\text{-}set\text{-}of(P) = parent\text{-}set\text{-}of(Q)$. This causes a single PARENT -SET to be created containing P and Q. This situation is diagrammed in figure 7.5. Note that in the diagrams, the box with two slots is overloaded: it is used to represent both couples and parent sets.

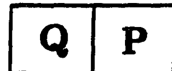

Figure 7.5: Diagram of representation for $couple(Q) = couple(P)$.

The final step in creating the problem situation is to add the representation of the statement

$\exists z[children(parent\text{-}set\text{-}of(Q)) = child\text{-}set(z) \wedge children(parent\text{-}set\text{-}of(z)) = child\text{-}set(S)]$.

The representation of this statement alone is shown in figure 7.6. The existential variable z is represented in the actual structure with an anonymous instance of FAMILY-MEMBER, call that instance sc z. As the system creates the structure representing this statement, the specialized procedure associated with CHILDREN' that

enforces the "unique range element" constraint equates the *child-set*(z) with the *child-set*(R). Since an individual is always a member of his own child set and since *child-set*(R) $= \{R\}$, the anonymous individual z is equated with R. This results in the structure shown in figure 7.7.

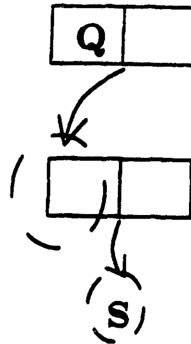

Figure 7.6: Diagram of representation for $\exists z[children(parent\text{-}set\text{-}of(Q)) = child\text{-}set(z) \wedge children(parent\text{-}set\text{-}of(z)) = child\text{-}set(S)]$.

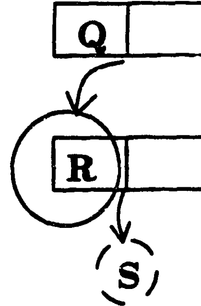

Figure 7.7: Diagram of representation for all three statements.

The system now answers the problem query by inspecting the parents of S which is a *parent-set*. The system knows that this set has two individuals but that it only knows one of them, R. So it answers the question with R and an indication that there are other members that it doesn't know.

Chapter 8

Analysis

This chapter discusses the following loosely related issues involved in developing a more formal theory of representation design:

1. The semantics of situations
2. Attempts to formally characterize the class of problems the system can design representations for
3. The system's expected coverage of analytical reasoning problems
4. The soundness of the representation design process
5. Complete query procedures

8.1 Semantics of Situations

When a situation is expressed in one of the representations designed by my system, a collection of data structures is created. This section shows how to treat this collection as a model of the problem. That is, we show that if a designed representation captures all the constraints of a problem class, then we can define a structure whose domain is the collection of data structures created in building the problem situation and that

structure will be a model of the problem, i.e., it will satisfy every statement in the problem.

We first develop the notion of an l-structure whose domain is a collection of data structures created in building a problem situation. Then we will show how to define that it means for an l-structure to be a model.

An l-structure is a triple

$$\mathcal{M} = < A, R, c >,$$

where

(1) A is a non-empty collection of instances of **set**, **relation**, **function**, and **individual**, or their specializations. A is called the *domain* of $\mathcal{M}$;

(2) R is a mapping from relation symbols in $\mathcal{L}$ into instances of **relation** or one of its specializations in A;

(3) c is a mapping from constant symbols in $\mathcal{L}$ into instances of **set**, **individual** or a specialization of one of these in A.

In general, an l-structure will provide only partial information about the truth of statements in $\mathcal{L}$. Therefore, the satisfaction relation is three valued: An l-structure may satisfy a formula in $\mathcal{L}$, it may satisfy the negation of that formula, or it may not satisfy either.

We define a satisfaction relation ($\overset{d}{\models}$) between our new type of l-structure and sentences in a problem statement as follows. [1]

Let $\mathcal{M}$be an l-structure and let $\sigma = < a_0, a_1, ... >$ be an assignment in $\mathcal{M}$. For all $\mathcal{L}$-formulas ϕ we define the relation σ *satisfies* ϕ in $\mathcal{M}$ ($\mathcal{M} \overset{d}{\models}_\sigma \phi$) by induction on the degree of ϕ:

(1) For terms $\mathbf{t_1}$, $\mathbf{t_2}$ of $\mathcal{L}$,

$$\mathcal{M} \overset{d}{\models} \mathbf{t_1} = \mathbf{t_2} \Leftrightarrow \textit{eq-class}(b_1) = \textit{eq-class}(b_2),$$

[1] This definition is similar to the definition given in [Bell & Machover 77, p.163].

where if $\mathbf{t_n}(n = 1, 2)$ is the variable $\mathbf{v_k}$ then b_n is a_k, while if $\mathbf{t_n}$ is the constant $\mathbf{C}$ then b_n is $c(\mathbf{C})$. The function *eq-class* is explained below.

(2) For terms $\mathbf{t_1}, \mathbf{t_2}$ of $\mathcal{L}$,

$$\mathcal{M} \overset{d}{\models} \mathbf{t_1} \neq \mathbf{t_2} \Rightarrow eq\text{-}class(b_1) \neq eq\text{-}class(b_2),$$

where if $\mathbf{t_n}(n = 1, 2)$ is the variable $\mathbf{v_k}$ then b_n is a_k, while if $\mathbf{t_n}$ is the constant $\mathbf{C}$ then b_n is $c(\mathbf{C})$.

(3) For the terms $\mathbf{t_1}, \ldots, \mathbf{t_n}$ of $\mathcal{L}$,

$$\mathcal{M} \overset{d}{\models}_\sigma \mathbf{R}(\mathbf{t_1}, \ldots, \mathbf{t_n}) \Leftrightarrow element(< b_1, \ldots, b_n >, R(\mathbf{R})),$$

where if $\mathbf{t_i}(i = 1, \ldots, n)$ is the variable $\mathbf{v_k}$ then b_n is a_k, while if $\mathbf{t_n}$ is the constant $\mathbf{C}$ then b_n is $c(\mathbf{C})$.

(4) For the terms $\mathbf{t_1}, \ldots, \mathbf{t_n}$ of $\mathcal{L}$,

$$\mathcal{M} \overset{d}{\models}_\sigma \mathbf{R}(\mathbf{t_1}, \ldots, \mathbf{t_n}) \Leftrightarrow element(< b_1, \ldots, b_n >, \overline{R}(\mathbf{R})),$$

where if $\mathbf{t_i}(i = 1, \ldots, n)$ is the variable $\mathbf{v_k}$ then b_n is a_k, while if $\mathbf{t_n}$ is the constant $\mathbf{C}$ then b_n is $c(\mathbf{C})$.

(5) $\mathcal{M} \overset{d}{\models}_\sigma \phi \Rightarrow$ *not* $\mathcal{M} \overset{d}{\models}_\sigma \neg\phi$.
(6) $\mathcal{M} \overset{d}{\models}_\sigma \phi \wedge \psi \Leftrightarrow \mathcal{M} \overset{d}{\models}_\sigma \phi$ and $\mathcal{M} \overset{d}{\models}_\sigma$ **psi.**
(7)$\mathcal{M} \overset{d}{\models}_\sigma \exists \mathbf{v_n}\phi \Leftrightarrow \mathcal{M} \overset{d}{\models}_{\sigma(n\underline{)}} \phi$ for some b in A. And the individual b is an instance of **set** or **individual** (or one of their specializations) in A.
(8) $\mathcal{M} \overset{d}{\models}_\sigma \forall \mathbf{v_n}\phi \Leftrightarrow \mathcal{M} \overset{d}{\models}_{\sigma(n\underline{)}} \phi$ for every b, an instance of **set** or **individual**(or one of their specializations), in A.

Eq-class is a function from instances of **individual**, **set**, (or one of their specializations) to equivalence classes maintained by specialized representations. Two instances are equal if there images under *Eq-class* are equal; they are unequal if there images

are explicitly marked unequal; otherwise the status of an equality relation is unknown.

Note that just as traditional model theory presupposes a consistent theory of sets, we must presuppose a consistent theory for the types in the library. Section 7.2 provided an informal semantics for these types.

8.1.1 Discussion

These semantics give a non-standard interpretation for universal quantification. The fact $\forall x\ P(x)$ is true if P is true of every individual in a situation, but does not claim to be a statement about *all* individuals. Hence, a universal statement can be true in a situation, but cease to be true after new facts are added. For example, $\forall x\ P(x)$ is true of a situation containing the individuals A and B if it contains the facts $P(A)$ and $P(B)$. But we can add $\neg P(C)$ to this situation, creating a new situation in which the general statement is false.

This non-standard interpretation is the reason for defining the notion of being captured in terms of all situations that can be expressed in a representation. The idea is that a representation captures a constraint when it can not be used to create a situation in which the constraint is false. For instance, since the representation in the example above is used to create a situation in which $\forall x\ P(x)$ is not satisfied, that representation does not capture that statement.

8.2 Attempts to Characterize The Class of Problems

This section discusses attempts to formally characterize a class of problems that my methodology can handle. I also provide an example of a problem outside of this class. Note that since the class of analytical reasoning problems does not appear to be formally characterizable, there is no clear relationship between it and the class

defined in this section.

Given the definition of l-structure from the last section, we can characterize the way in which specialized representations are used to solve problems as building a model of the specific situation described in the problem and then inspecting that model to answer questions. In this view, we should only expect to be able to solve a problem if it has at least one finite model. If there are no finite models of a problem, then when the problem situation is expressed in the specialized representation, an infinite structure will be created.

This notion of class appears to mesh quite nicely with ordinary intuitions about the way people solve analytical reasoning problems: the process is often described as constructing a "model" of a problem situation, then inspecting it to solve the problem. This notion is helpful in understanding the theoretical limits of the current representation design method.

The finite model characterization is helpful in understanding when problems are beyond the scope of this methodology. An example of a problem outside the scope will help make this clear. Any problem that requires reasoning by induction is outside the scope. Induction problems violate the syntactic restriction that queries may not ask about general facts (facts that do not contain individuals). More importantly, induction theorems are questions about the properties of infinite well orderings. All models of such orderings are infinite, so such problems are outside the scope of my methods.

8.3 Coverage of Analytical Reasoning Problems

Section 1.3 provided an informal characterization of the class of analytical reasoning problems. I believe that the current system can easily be extended to provide reasonably high coverage of analytical reasoning problems. I characterize the coverage of the existing knowledge by answering the following two questions: First, how well

does the existing library cover a large body of analytical reasoning problems? Second, how much overlap is there in the knowledge used on different problems?

There is evidence to suggest that the current structure library population is very close to a sufficient collection for designing representations for most analytical reasoning problems. The current system was designed after a survey of approximately two hundred analytical reasoning problems. From these, I compiled a set of twenty representative problems. From the representative set, I chose three problems that I found among the most difficult to solve. I call these the *paradigm* problems. I then studied the representations that I and two other people used in solving the paradigm problems. The current structure library population is the result of that study.

Examination of the other seventeen representative problems showed that the existing structure library was sufficient: the problems did not use any additional structures.

This research would not be interesting if each problem solved used a disjoint subset of the knowledge. This has proven to be far from the case.

The current representation design system generates representations for the three paradigm problems and some variations (eight problems total). One of the paradigm problems is the FAMILIES problem. The others are shown in figure 8.1 and figure 8.2.

Even though the paradigm problems appear quite different, all the constraints on the problems can be captured in representations that use only structures from the existing library. Furthermore, figure 8.3 shows that most of the structures are used in more than one problem. Thus there is significant overlap in the structure library knowledge used in designing representations for the paradigm problems.

There is also overlap in the introduction knowledge used in generating representations for these problems. There are a total of ten rules in the introduction knowledge base. Figure 8.4 shows the number of different rules used in designing each paradigm problem and the number of total rule firings that occurred in the design efforts. All of the rules in WAITERS were also used in the other two problems. Four of the rules

Eight law professors are housed in a single wing of a building. The wing contains ten offices, numbered 1 to 10, in that order; each professor is assigned to a different office, and two offices are left empty for use as meeting rooms. The professors are named Boswell, Dyer, Garrett, Harrelson, Kranepool, Ryan, Taylor, and Weis.
Dyer is four offices away from Kranepool.
There is one empty office and one occupied office between Taylor and Harrelson.
Ryan is in an office next to Boswell.
Dyer is in an office next to Garrett.
Kranepool is between an occupied office and an empty office.
Weis is in office 2.
Garrett is in office 7.
Who is in office 4?
Which offices are unoccupied?
Is Ryan, Dyer, Garrett, Taylor a possible sequence of offices?

Figure 8.1: The PROFESSORS problem

in PROFESSORS were used in FAMILIES.

8.4 Soundness of Representation design

One question that arises about the representation design process concerns how much faith we should put in the answers that we get with the representations designed by the system. This is the question of whether or not the representation design process is sound.

I have shown that if the system produces a fully constrained representation and that representation halts building a problem situation, then the answers produced with that representation are always answers to the original problem.

There are two kinds of lemmas that must be proved to obtain the soundness result. First, since representation design changes problem statements, we must prove that all transformations done on the predicate calculus problem statement are sound.

The purpose of the second kind of lemma is to show that the process of capturing constraints in a representation is sound. In particular, we must show that when the

A restaurant employs eight waiters, D, E, F, G, H, I, J, and K, each of whom works four days a week. The restaurant is open every day except Monday. On Friday and Saturday, a staff of six waiters is needed. On all other days when the restaurant is open, a staff of five is needed.
D cannot work on Tuesday or Thursday.
E cannot work on Wednesday.
G cannot work on Thursday or Saturday.
H cannot work on Friday.
J cannot work on Tuesday or Sunday.
K cannot work on Wednesday or Friday.

Is D,E,F,I,K a possible staff of waiters for a Tuesday?
Is Tuesday, Wednesday, Thursday, Sunday a possible work week for G?

Figure 8.2: The WAITERS problem

Problem	*Structures Used*

FAMILIES	`function, 1-1 function,`
	`partial-function, partial-1-1-function,`
	`disjoint-set, fixed-size-disjoint-set`

WAITERS	`function, fixed-size-set, set` `antitrans-antisymm-irref-rel`
PROFESSORS	`1-1-function, 1-1-partial-function, set,` `fixed-size-set`

Figure 8.3: Library structures used in each paradigm problem.

constraints of a set of general statements Φ are captured by a representation, the following is the case. If a set of specific facts Ψ is added to that representation, then the problem situation created will contain every specific fact that follows from the union of Φ and Ψ. Since the allowable queries can be answered given the specific facts that follow from a problem, this result is sufficient to ensure correct answers.

All the pieces of the soundness proof have appeared the preceding chapters. This section provides a summary with pointers to the individual pieces.

Problem	*# Different Rules*	*# Total Rule Firings*
FAMILIES	9	31
WAITERS	3	7
PROFESSORS	5	11

Figure 8.4: Data on introduction rule usage in paradigm problems.

8.4.1 The Irrelevance Filter

Soundness of the irrelevance filter was discussed in section 3.2.2. The idea of the proof is that the filter will remove only concepts that can not appear in a proof of a problem query. Therefore, removing them can not change the answer to the query.

8.4.2 Concept Introduction

Soundness of introduction is discussed in section 5.2. The main points of the argument are the following.

A restriction is placed on each introduction rule to ensure soundness of the overall process. For the purposes of this restriction, introduction is viewed as adding a new symbol to the language of a problem and adding a new statement to the problem defining the symbol. Each introduction must be shown to extend all models of a problem to models that satisfy the new statement. This is done (by hand) by interpreting the statement that a rule introduces as an abstract procedure to perform on models of a problem to extend them to models of the statement that the rule introduces.

For example, given a relation R, one rule introduces a function F_R with the statement

$$\forall x \forall y [y \in F_R(x) \Leftrightarrow R(x, y)].$$

We show that the rule meets the soundness restriction by showing how to treat this statement as a procedure that extends any model of the original problem to be

a model of the statement. Take any model of the original problem and add F_R to the set of function symbols of the model. Next, for each element, x, in the domain F_R, create a pair of the form $< x, \{y \mid R(x,y)\} >$. Add the union of these pairs to the domain of the model and designate this new set by the symbol F_R. The new model is an extension of the original model.

8.4.3 Classification

Classification moves constraints from predicate calculus statements into abstract data types. The only time a statement is removed from a problem during classification is when it can be shown that the representations referred to capture the constraint of the statement. The soundness of classification follows from the the validity of capture verification (shown in section 4.3).

8.4.4 Operationalization

There are two parts to operationalization. Given a statement, the first part generates a collection of statements in operational form and the second part generates procedures that enforce the constraint of each of the statements in operational form.

We must show that the operational forms are generated by sound inference. Statements in operational form are derived from conditional statements by making a series of assumptions in those statements. The assumption making process consists of repeatedly making an assumption (σ) in a statement (ϕ), simplifying the statement, and then constructing the new statement

$$\sigma \Rightarrow \phi(\sigma/true),$$

where $\phi(\sigma/true)$ is the result of substituting *true* for all occurrences of σ in ϕ. This is shown to be sound in section 6.3. Since each of these steps is sound, it follows that if Φ is a set of statements generated in operationalizing a given statement ϕ, then $\phi \Rightarrow \Phi$.

For the second part of operationalization, we must show the other direction of the above implication, i.e., that $\Phi \Rightarrow \phi$. This is done in section 6.5. The argument is summarized as follows. First, we show that the antecedents of each operational form generated from ϕ correspond to combinations of specific facts whose addition to a situation cause ϕ to become *false*. Then we show that the system either generates an operational form for each such combination or indicates that it has failed to capture ϕ. We conclude that if ϕ is captured then no situation can be created that violates the constraint of ϕ.

8.5 Complete Procedures for Answering Queries

As noted in chapter 1, specialized representation can be used to answer the following types of queries:

1. $\Box\phi$, where ϕ is a specific statement
2. $\Diamond\phi$, where ϕ is a specific statement
3. find-all $x : \phi$, where ϕ is a mixed statement
4. find-the τ, where τ is a function term

Here is how each of these is currently answered:

1. To determine if some specific fact is true in a problem situation, the system inspects the situation to see if the fact is present. To determine if a mixed statement is true, the statement is expanded into a conjunction of specific statements and a necessity query is done to check each of these. Then the system does a possibility query. For example, to answer the question, "Are A, B, and C the only children of S?" the system first checks that A, B, and C are children of S and then it checks to see if it is possible for any other individual to be a child of S.

2. To determine if it is possible for some specific fact to be true in a problem situation, the system adds that fact to the situation. If no contradiction results, the system concludes that the fact is possible.

   Possibility queries about mixed statements are expanded in a way similar to necessity queries about mixed statements. The difference is that each specific statement in the conjunct obtained by expanding is added to the problem situation. If a contradiction occurs from any of these additions, the query is answered negatively. For example, to answer the question, "Is is possible that A, B, and C are the only children of S?" the system adds *child*(S, A), etc. to the problem situation. Then it does a find-all query to see if there are any other individuals in the collection referred to by the mixed statement. Continuing the example, the system finds all the children of S. If there are any children other than A, B, and C, the query is answered "no." Otherwise, it is answered "yes."

3. To find all the individuals that stand in some relation to a specific individual, the system searches the problem situation looking for all facts relating individuals to the specific individual.

4. A find-the query can only ask for the value of a functional expression like "find-the *siblings*(S)." These questions are answered by retrieving from the problem situation the image of an individual under the **function**. For example, the query, "find-the *siblings*(S)," is answered by retrieving from the problem situation the image of S under SIBLINGS.

   As noted above, the representation design system can transform find-all queries into find-the queries by reformulating a problem. The reformulation has the effect of adding a new kind of individual to the problem. For example, the query "find-all x : *sibling*(S, x)" is transformed into "find-the *siblings*(S)," where the range elements of *siblings* are individuals that are themselves sets of individuals.

These query procedures are not complete because a problem situation can represent a fact in its procedures without recording it in its data structure. Consequently, correct answers to some necessity queries can only be obtained by adding the negation of the fact to the situation and seeing if a contradiction results. Consider, for example, the following problem:

$P(a)$
$\forall x[P(x) \Rightarrow Q(x)]$
$\forall x[\neg P(x) \Rightarrow Q(x)]$
Query: $\Box\, Q(b)$.

Suppose that in the specialized representation, P and Q, are represented in terms of **relation** and that the two general statements are captured by operationalization. The following specific facts are true in the created situation:

$P(a)$, $Q(a)$.

$Q(a)$ is added by one of the procedures operationalizing the first general statement.

Thus the system will answer the query, "no." However, notice that because both $P(x)$ and $\neg P(x)$ imply $Q(x)$, that $\forall x\ Q(x)$ is true. Thus $Q(b)$ is true and hence $\Box\, Q(b)$ is true. The incompleteness comes from the fact that the knowledge about $\forall x\ Q(x)$ being true is embedded in the procedures of the specialized representation.

The complete versions of the query procedures work by "forcing" the procedures of a representation to report contradictions to the that a fact is not possible in a situation. Continuing the above example, the procedure for answering the query $\Box\, Q(b)$ adds $\neg Q(b)$ to the problem situation. Two procedures execute in response. One is part of the operationalization of

$\forall x[P(x) \Rightarrow Q(x)]$.

It adds $\neg P(b)$ to the situation. The other procedure is part of the operationalization of

$\forall x[\neg P(x) \Rightarrow Q(x)]$.

It responds by adding $P(b)$ to the situation. This causes a contradiction to be sig-

nalled. Consequently, $\Box Q(b)$ is true.

Here are complete versions of the query procedures:

1. For a specific fact ϕ, the system answers $\Box\phi$ by adding $\neg\phi$ to the situation. If a contradiction occurs, $\Box\phi$ is true. When ϕ is a mixed fact, it is expanded into a conjunction of specific statements whose negations are added to the problem situation. If all of these cause a contradiction, then this is followed by a possibility query. For example, to answer the question, "Are A, B, and C the only children of S?" the system first adds $\neg child(S, A)$, etc. to the problem situation. If each of these causes a contradiction, then it checks to see if it is possible for any other individual to be a child of S. If not, the system reports that $\Box\phi$ is true.

2. $\Diamond\phi$ is answered in the same way as before.

3. Find-all $x : \phi$ is answered by searching the problem situation for all individuals for which ϕ is true and then trying to prove that there can be no others. Proving that there can be no additional individuals is accomplished by creating an anonymous individual, asserting that it is unequal to the individuals already found, and adding the new individual to the situation. If this causes a contradiction, we know that it is not possible for there to be any additional individuals with property ϕ, hence we have found them all. If adding the new individual does not cause a contradiction, then it is possible for there to be other individuals. Unfortunately, it is also possible, in this situation, for the problem to imply other specific individuals with property ϕ without our being able to identify them. Thus, we can not guarantee completeness in this case.

4. The procedure for answering find-the queries begins in a manner similar to the incomplete version: the system retrieves the image of the individual under the function. If the image is an open set, then the system must check to see if

it is possible for it to contain additional individuals. For example, given the query "find-the *siblings*(S)," if this set is open then the system tries to add an additional member. If doing this causes a contradiction, the set does contain all of its elements but the situation does not reflect this. If no contradiction occurs we are in the same incompleteness situation that can occur with find-all queries.

Chapter 9

Related Work

This section highlights the differences between my research and previous work in the following areas:

- Systems that solve word problems
- Automatic programming
- Research on good representations
- Problem reformulation

9.1 Previous Systems that Solve Word Problems

Since my representation design system solves word problems, one might expect there to be an interesting relationship between my system and previous systems that did this. This section discusses the relationship between my research and two previous efforts to solve word problems and conclude that the relationship between my system and these two previous systems is only superficial. All of the systems solve word problems, but the underlying research objectives are very different.

In [Bobrow68], the author reports on a program called STUDENT that solves high school algebra word problems. Bobrow was interested in issues of translating problems

stated in natural language to simple algebraic equations. He was also interested in a psychological model of high school student's performance in this activity and used his model to make predictions about human performance.

The most important difference between STUDENT and my work is that the objective of STUDENT was to translate a problem, stated in English, into a single preestablished target representation. My system is concerned with designing good target representations. For example, in most cases, confronted with a problem like those that STUDENT solved, I would expect my system to select algebraic equations as the target representation. [1] For high school algebra word problems, there would not be much representation design to do. However, given a different kind of problem, my system should (and does) design different kinds of representations.

Another important difference between Bobrow's and my work is that he was interested in the psychological implications of his model. I have not concentrated on this in my work. Like many other efforts in AI the methods my system employs are inspired by human performance, but my emphasis has been on designing good representations regardless of whether or not I have captured the design process that people employ.

The other word problem system I compare my system to is the work reported in [Novak76]. This system solved physics word problems in the area of rigid body statics. Like Bobrow's, this work was concerned with translating a problem, stated in English, into a single target representation and then solving it. However, Novak points out that physics problems of this type are not deductive. The hard part of solving them is figuring out what assumptions to make so that the problem decomposes into idealized pieces to which physical principles can be applied directly. By contrast, the problems that my system works on are deductive and do not require making assumptions to simplify the situation presented in the problem.

[1] I have not tried my system on problems like those that STUDENT solves. However, my system does design representations in terms of equations. For example, part of the representation that is designed for the FAMILIES problem is a collection of equations relating set of children, siblings, brothers, and sisters.

Another important difference between both of the research efforts discussed here and mine is that they considered the natural language translation problem an important part of what their systems did. I have not considered natural language translation in my research because I believe that analytical reasoning problems are stated in a restricted enough subset of English that the translation problem can be solved by "off the shelf" technology.

9.2 Relationship to Automatic Programming

Since my system generates programs, it can be viewed as an automatic programming system. This section argues that the major difference between my work and other automatic programming systems is that they perform tasks such as algorithm design, data structure selection, and optimization (of algorithms and data structures) within a fixed representation which is chosen by a person prior to the point where the automatic programming system gets involved. By contrast my system is concerned with those earlier steps in the problem solving process during which a representation is designed. There are a number of ways to demonstrate this distinction. One way is to compare the input to automatic programming systems with the input to my system.

Automatic programming systems begin with a specification of a program as input. In contrast, my system starts with a specification of a problem. For instance, the SAFE system is an automatic programming system that accepts an informal specification of a program and formalizes it. Figure 9.1 is an example of a specification given to SAFE. This specification is process oriented. Analytical reasoning problem specifications clearly are not.

I chose SAFE as one effort to discuss because, unlike most work in automatic programming, it is concerned with completing informal specifications. Like my system, SAFE tries to acquire missing information. The techniques it uses identify incom-

```
((THE SOL)
 (IS SEARCHED)
 FOR
 (AN ENTRY FOR (THE SUBSCRIBER)))

(IF ((ONE)
     (IS FOUND))
    ((THE SUBSCRIBER'S (RELATIVE TRANSMISSION TIME))
     (IS COMPUTED) ACCORDING TO ("FORMULA-1"))

((THE SUBSCRIBER'S (CLOCK TRANSMISSION TIME))
 (IS COMPUTED) ACCORDING TO ("FORMULA-2"))

(WHEN ((THE TRANSMISSION TIME)
       (HAS BEEN COMPUTED))
      ((IT)
       (IS INSERTED)
       AS (THE (PRIMARY ENTRY))
       IN (A (TRANSMISSION SCHEDULE))))

(FOR (EACH PATS ENTRY)
     (PERFORM)
     (: ((THE RATS'S (RELATIVE TRANSMISSION TIME))
         (IS COMPUTED) ACCORDING TO ("FORMULA-1"))
        ((THE RATS'S (CLOCK TRANSMISSION TIME))
         (IS COMPUTED) ACCORDING TO ("FORMULA-2")))))

((THE RATS (TRANSMISSION TIMES))
 (ARE ENTERED)
 INTO (THE SCHEDULE))
```

Figure 9.1: Example of a specification given to SAFE.

pleteness in a natural language specification based on syntactic cues in the text. In contrast, my system relies on the semantic properties of library structures to guide a search for missing information.

Another point of distinction between the two systems has to do with irrelevance. Informal program specifications do not appear to contain irrelevant information and SAFE does not consider this possibility at all.

The distinction I have drawn between specifying programs and problems seems clear enough in the example just given. However, it becomes less clear for systems such as [Cohen86] whose input is a predicate calculus specification of a set to generate or a condition to test. Figure 9.2 gives an example of the input to Cohen's system which is called AP5. This specification describes more of what a programmer wants the machine to do than how it is to do it.

```
(DeclareRel Sex 2) ; a binary relation
; e.g., (Sex Sam Male)
(DeclareRel Parent 2) ; (Parent child parent)
(DefineRel Sibling (x y)
   (and (not (eq x y))
        (Exists (parent)
                (and (Parent x parent)
                     (Parent y parent)))))
(Defun list-nephews (person)
  (loop for nephew s.t.
    (and (Sex nephew 'male)
         (Exists (sibling)
           (and (Sibling person sibling)
                (Parent nephew sibling))))
    collect nephew))
```

Figure 9.2: A specification input to AP5.

Note that we could easily define a similar analytical reasoning problem that provides a definition of the *sibling* and *nephew* relation and then asks the question

find-all $x : nephew(P, x)$.

However, AP5 and my system are trying to address very different problems. AP5 is given a specification and a collection of annotations that select fixed representations for the primitive relations in the specification. It compiles the specification given information it has about the costs of different ways of testing relations and for generating n-tuples of individuals standing in some relation.

Once the person observes the behavior of the program that AP5 generates, he/she can choose better representations for the primitive relations and use AP5 to recompile the specification. The important point to notice is that the person selects the representations and AP5 produces the best code it can based on those selection. My system chooses representations.

I once conceived of my system producing a specification instead of a program. I had in mind that this specification would then be handed to an automatic programming system such as [Barstow79] or possibly AP5 which would then make data structure selection and algorithm optimization decisions. I was subsequently persuaded to produce a program instead. Still, the code generation part of my system does not try to perform data structure selection or algorithm optimization. For example, my system is concerned with deciding that some problem concept is best represented as a set. When a program is generated, the representation design system uses a default implementation for sets instead of trying to choose from among alternative implementations (e.g., a list or a bit vector) as, for example, [Rovner76] does.

More conventional automatic programming systems such as QA3 (described in [Green68]) take the approach that automatic programming is a theorem proving activity in which the system tries to prove the existence of some entity that we want a program to produce. A human provides the system with a theory of the operations available for writing programs and when the system uses these to prove a theorem, it produces the desired program as a by-product. For example, to generate a program that sorts a list, QA3 tries to prove a theorem stating that for all (finite) lists

there exists a sorted version. Since it has been provided with axioms that describe list operations and operations for comparing numbers, a by-product of the proof is a program that sorts lists.

As Green and others have pointed out, the formulation of the set of axioms that describe both the operations available for programming and the desired program to be written can have a dramatic effect on whether the theorem proving approach succeeds. Finding a good formulation of a set of axioms is a large part of what my research is about.

Operationalization is the activity that my system engages in that is most similar to conventional automatic programming in that is transforms predicate calculus statements into code fragments. Operationalization is a restricted form of automatic programming that is used as a last resort in capturing problem constraints. It can not produce recursive programs and can only produce simple forms of iteration.

9.3 Research on Good Representation

As I have already stated, the principle difference between my research and previous efforts to understand what makes a representation good that they were concerned with recognizing the properties of good representations and my research is about generating such representations prospectively. Still much of this work provided me with a good starting point and also helped me to sort out what the important issues about representation are. Several works give good explanations of what it means for a representation to be direct (or analogical) and what the consequences of having such a representation are (see [Sloman71, Sloman85, Hayes74, Lenat & Brown 84]).

Pylyshyn's work, reported in [Pylyshyn75], helped me to understand a phenomenon that I observed in the representations that people design to solve analytical reasoning problems. Specifically that it does not make sense to talk about the directness of the structure of a representation devoid of the semantic interpretations functions that

give a semantics to that structure. For example, even in an obvious case like using a structure with two slots to represent married couples, it is really the interpretation provided by the procedures that manipulate couples that preserve the relationship between the representation and the real word object couple.

9.4 Problem Reformulation

Some early work on problem reformulation can be found in [McCarthy64, Newell65], and [Newell66]. Instead of trying to do an exhaustive comparison with previous work in problem reformulation, this section I concentrates on three works in the area. One work reported in [Amarel68] is included because of its influence in the area. The two other works [Korf80, Subramanian87] are included because they illustrate recent work in the area. Special attention is paid to [Korf80] because the work reported there is the most direct ancestor of mine.

Amarel's work is a paper and pencil study leading us through a successive refinement of representations for the familiar Missionaries and Cannibals (M&C) problem. The work differs from mine in two ways. First, it considers a substantially different domain: reasoning about the effects of actions. Second, it does not seriously consider issues in automating the search for a refinement sequence. This results in a loose description of the transformations used and little feeling for the space of possible representations being searched through. In contrast, these issues have been at the forefront of my concerns.

Amarel leads us through a sequence of refinements and, at each step, identifies some interesting properties of the problem and then discusses a refined representation that capitalizes on those properties. Many of the transformation discussed are informally motivated by arguments about the search space generated in finding solutions. For example, in the first version of the problem, the operators encode the possible moves and the non-cannibalization conditions are expressed as general constraints on the

problem. In the next version, the general constraints are "compiled" into the operators so that they are applicable only when they produce a non-cannibalized state. The resultant representation (with constraints compiled into the operators) is better because the search space is smaller.

In a number of places in his argument, Amarel fails to define what is involved in the transformations he offers. For example, he begins with representations in terms of production systems in which rules construct new states from old ones. At one point he switches to a reduction system. This takes a statement of the form

initial state $\Rightarrow$ *final state*,

which is interpreted as "the final state is attainable from the initial state." The process of solving the problem involves "reducing" the statement relating the initial and final states to a sequence of states attainable by operators. The problem with the switch from production systems to reduction systems is that we are not told how such a switch is made or when it is advantageous to do so.

In another part of the paper, problem representations are described in terms of state space graphs. At one point, Amarel structures these graphs by viewing them as juxtaposed two dimensional grids. Unfortunately, the reader is left with little feeling as to what the precise nature of this transformation is and the conditions under which such a transformation is useful. In this presentation he exploits our visual abilities to notice certain properties. But what are these properties? And how did Amarel identify them in the problem?

Korf's work, reported in [Korf80], has a similar mind set to that of Amarel's. However, Korf went much further to develop a formalism for describing representations and transformations between different representations. He also provided a clean characterization of two dimensions along which different transformations effect representations: viewing them as homomorphic and isomorphic transformations in a space of possible representations. Homomorphic transformations preserve structure and re-

duce information content. Isomorphic transformations preserve information content but change a representation's structure.

An important contribution of his work is that he demonstrated how representation changes that had been previously viewed as "leaps of insight" could, in fact, be modeled as gradual refinement involving transformations of the type he identifies.

One of his examples is the mutilated checker board. He describes transformations along the way to a representation consisting of two integers, one representing the number of uncovered black squares and one representing the number of uncovered red squares. One mapping involves assuming that the squares are indistinguishable. Then any situation in which the same number of squares are uncovered can be thought of as the same. This can be modeled as a homomorphic transformation that maps board situations into sets of indistinguished uncovered squares. Since all that is important about the sets of squares is their cardinality, the set representation can be transformed into one in which the sets are replaced by integers representing their cardinality. This transformation can be modeled as an isomorphic mapping between sets and integers representing their cardinality.

In some ways my work can be seen as an extension of Korf's. He was concerned with characterizing a space of possible representations and types of transformations on them. My work is concerned with *how to choose the right transformations to do to arrive at a good problem representation.* I have identified some of the essential properties of representations and given a method to design representations with those properties.

Korf (and Amarel) viewed problem solving as state space search and observed that changes in representation (i.e., the description of a problem state) affect the size of the space. The focus of my work has been explaining *how* representations do this and how to design representations that yield smaller search spaces. The claim is that when a representation captures more constraints in its structure and behavior the

problem solving space is reduced.

One point that Korf appears to have missed is that he actually describes two different kinds of homomorphic transformation: transformations on state spaces and transformations on state descriptions. An example of a transformation on a state space is introducing uninterruptible operator sequences (macro-operators. lemmas, etc.) and then removing the original operators. This has the effect of reducing a search space by "skipping over" intermediate states derived by the original operators. An example of a transformation on a state description is the collapsing of a check board into two integers representing the cardinality of the set of red and black squares respectively. This has the effect of reducing a search space by throwing away distinctions in a state description and thereby forming equivalence classes of states.

These two different kinds of homomorphisms have the same effect on a search space: the overall size of the space is reduced. However, the their focus is different. My research has only considered one method of collapsing state descriptions in its use of the irrelevance filter (described in Chapter 3). The irrelevance filter can be viewed as removing information from a state description that is irrelevant to solving a problem. One can imagine other homomorphic transformations (such as those suggested in Korf's further work section) whose effect is to remove information without changing a problem's solution.

My research does consider homomorphic transformations on state spaces in its identification of specialized inference rules. These rules enforce consequences directly in a representation, skipping over intermediate steps necessary for a theorem prover to deduce those consequences.

[Subramanian87] reports on a study of the use of logic as a tool to investigate properties of irrelevance. The theory that they discuss is applied to three example problems including proving that a particular reformulation is justified because it removes only information that is irrelevant to solving a problem. This example applies the theory

to justify the removal of intermediate links in an ancestral tree for a problem that asks whether two individuals are in the same family. The intermediate ancestral links are irrelevant because all that matters to answering the question is the relationship between a person and the root of his her family tree.

Collapsing ancestral links is viewed as reformulation. The problem given in the paper is initially stated in terms of *father*, *ancestor*, and *samefamily* and is reformulated in terms of *foundingfather* and *samefamily*. This example provides one example of a kind of reformulation different from mine. My reformulations are guaranteed not to change the semantics of a problem. In contrast, the example given in this paper is a reformulation that changes the semantics of the problem in a way that preserves the solution. It would be interesting to pursue the use of this type of reformulation in automatic representation design. This does not, however, appear to be the current emphasis of the work reported in [Subramanian87].

Chapter 10

Summary

The important contributions of this thesis are the development of the idea that good representations capture constraints in there structure and behavior and a computer program that designs good representations by specialization.

A *representation* is a mapping between concepts and structures with behavior. The structures in better representations capture more constraints of a problem. For example, representing "mother" as a function (e.g., $mother\text{-}of(A) = B$) captures more constraints than representing it as a relation (e.g., $mother(A, B)$) because the first representation captures all the constraints that the second does and, in addition captures the single valuedness of "mother."

When a representation captures more constraints, fewer situations can be expressed in it. For example, when "mother" is represented as a relation we can express situations in which an individual has more than one mother; when it is represented as a function this situation can not be expressed. This reduces the space that a problem solver must consider in a specialized representation. This, in turn, results in more efficient problem solving behavior.

Representation design begins with a *problem statement*: a collection of statements in a sorted first order logic along with one or more queries written in a separate query language. The goal of representation design is a representation that is *fully*

expressive and *fully constrained* with respect to a problem's class. The representation design system tries to achieve this by specializing the structures used to represent a problem's concepts (e.g., specializing *mother* to *mother-of*).

Intuitively, two problems are in the same class when the same general constraints are relevant to solving them. They can differ in the individuals they mention, in the particular relationships between those individuals, or in which individuals are constrained in a particular way. For example, a problem in the same class as the FAMILIES problem would refer to the same collection of concepts (e.g., *married*). However, it could mention a different collection of individuals and could have a different number of individuals being married.

A representation is fully expressive with respect to a class if every problem situation in the class is representable. We assume that the representation of the problem given in the problem statement is fully expressive. The representation design system must be sure to preserve this property as a representation is specialized.

In a fully constrained representation, the syntactic structures representing a problem's concepts collectively capture all of the constraints of the problem's class. Another way to say this is that a representation is fully constrained with respect to a problem class if the situations that can be expressed in the representation always satisfy the constraints of the class.

10.1 The Process of Representation Design

The first step in representation design is to develop a description of a problem's initial representation. This is done by identifying the primitive concepts relevant to solving the problem and then stating which syntactic structure each concept is represented with. Because problems are stated in an extension of first order logic including sets, the possible representations initially are individual, relation, function, and set. For example, *married* is initially represented as a relation because it appears in atomic

formulas of the form *married*($term_1$, $term_2$). In this step. the representation design system tries to arrive at the smallest collection of concepts that is sufficient for a fully expressive representation of the problem. This is called the collection of *represented concepts.*

The rest of the process of representation design attempts to create a fully constrained representation without losing full expressivity. The methods for doing this work with a description of the constraints of a problem class, simultaneously capturing problem constraints and removing statements that express those constraints from the class description. This process continues until either all of the constraints of the class are captured or the representation design system exhausts its methods.

Constraints on a problem concept are captured structurally when the concept is represented with a structure having the same properties as the concept. Classification uses a library of structures organized into a taxonomy around the constraints that they capture: Structures that capture more constraints are more specialized. Constraints on a concept are captured structurally by identifying the library structure that captures the most constraints on the concept and then representing the concept with that structure. This is done by classifying concepts in the taxonomy.

Given a problem, classification comes up with a collection of maximally specialized representations for it concepts. However, classification by itself has a serious limitation: Its success depends on the particular vocabulary used to state a problem. The FAMILIES problem, for example, is stated in terms of *married*, which is classified beginning with `relation`. None of the specializations of a `relation` capture the fact that married couples are all of size two. However, if the problem had been stated in terms of *couples*, classification would have been more successful: *couples* are a specialization of `set` and a specialization of `set` takes advantage of size constraints.

In general, the strategy of concept introduction works because different library structures capture different constraints and have different specializations. Representing a

concept differently may be a better fit between the constraints on that concept and the constraints captured by the different representation.

Introducing a new concept affects a problem in two ways. The primary affect is to give the system access to different representation design knowledge: by changing the representation of a concept the system gains access to syntactic structures that enforce different constraints, and have different specializations.

The other effect of introducing a new concept is that it enables reformulation of the problem. This is useful because it often allows new properties to be discovered in the problem statement. Reformulation is accomplished by treating the logical definition of a new concept as a rewrite to perform on problem statements.

As new concepts are introduced, the representation design system explores a space of alternative problem formulations. For example, introducing *couples* for *married*, creates two alternative formulations: one in terms of *married* and one in terms of *spouses*. Alternative problem formulations are maintained because the representation design system can not always tell, when it introduces an alternative concept, whether or not that concept will result in a more specialized representation. For example, the formulation of the problem in terms of *married* can not be compared to the formulation in terms of *spouses* until *spouses* is classified.

Classification extended by concept introduction is called *extended classification.* What is interesting about extended classification is that the two processes that are involved in it are fairly simple, however, the behavior of the combination of classification and introduction is very dynamic and can result in radical reformulations of a problem as a representation is being designed for it.For example, consider the of introductions that result from extended classification of *married*:

1. The concept *spouses* is introduced. This is a function from individuals to the sets of individuals to whom they are married.

2. The concept *non-empty-spouses* is introduced. This is a partial function from individuals to the non-empty sets of individuals to whom they are married.

3. The concept *spouse* is introduced. This is a partial function that captures the fact that individuals have at most one spouse.

4. The concept *couple* is introduced. This is a partial function from individuals to the married couple that they are members of. COUPLE captures the following facts: not all individuals are married, married couples are disjoint from all other married couples, married couples contain exactly two members.

Classification and concept introduction run as coroutines, trying to capture all of the constraints of a problem. As they do this, the statements of those constraints get removed from the problem. They usually fail to capture all the constraints leaving statements of the uncaptured constraints in the problem. Operationalization then tries to capture the constraints of any remaining statements by writing new procedures and using these to specialize the representations created by classification and concept introduction.

For example, suppose the statement

$$\forall x \forall y [x \in siblings(y) \Leftrightarrow y \in siblings(x)]$$

is left uncaptured and that *siblings* is represented as a function from individuals to their set of siblings (i.e., the range elements of *siblings* are represented as **sets**). Operationalization captures the constraint of this statement by writing procedures that watch for the addition of facts that violate the statement's constraint. One such fact has the form $x_1 \in siblings(y_1)$. Accordingly, one procedure that operationalization writes watches for the addition of facts of this form and responds by adding a fact of the form $y_1 \in siblings(x_1)$.

When operationalization succeeds in writing a procedure like the one above for every fact that violate a statement's constraint, the statement is captured (and is, therefore,

removed from the class description).

10.2 Summary of An Example of Representation Design

This section summarizes the design of the representation for the small FAMILIES problem used as an example throughout this thesis. It is shown again in figure 10.1.

SORT($P, FAMILY\text{-}MEMBER$), SORT($Q, FAMILY\text{-}MEMBER$),
SORT($R, FAMILY\text{-}MEMBER$), SORT($S, FAMILY\text{-}MEMBER$)
$grandchild(Q, S)$
$\forall x\ child\text{-}of(P, x) \Leftrightarrow x = R$
$married(Q, P)$
Query: find-all x: $parent(S, x)$

Figure 10.1: The small FAMILIES problem.

The representation design system begins by prompting the user for definitions of the concepts mentioned in the problem statement. The user supplies definitions for *grandchild* and *parent*:

$$\forall x \forall y [grandchild(x, y) \Leftrightarrow \exists z (child\text{-}of(x, z) \wedge child\text{-}of(z, y))]$$
$$\forall x \forall y [child\text{-}of(x, y) \Leftrightarrow parent(y, x)].$$

Then, the system runs the irrelevance filter and finds that *married* is disconnected from the rest of the problem. So it asks whether there are necessary or sufficient conditions connecting it. The user responds with both:

$$\forall x \forall y \forall c [child(x, c) \wedge child(y, c) \wedge x \neq y \Rightarrow married(x, y)]$$
$$\forall x \forall y \forall c [married(x, y) \wedge child(x, c) \Rightarrow child(y, c)].$$

Next, the system derives a description of the problem's initial representation. These are:

```
COLLECTION* FAMILY-MEMBER OF unique-individual
MARRIED: relation(FAMILY-MEMBER,FAMILY-MEMBER)
CHILD-OF: relation(FAMILY-MEMBER,FAMILY-MEMBER)
PARENT: relation(FAMILY-MEMBER,FAMILY-MEMBER)
```

Even though *parent* is not primitive, a definition is included for it because it appears in a find-all query. A definition is included for *parent* so that the system can consider

the cost of answering the query. Including a definition for it causes it to be classified and classifying it allows the cost of answering the query to be considered in alternative formulations of the problem.

Next, the representation design system performs extended classification on the concepts defined in the description of the initial representation. Here we begin (arbitrarily) with *married*. This results in a sequence of several introductions yielding the concept *couple*; details of this were given in section 5.5.

As a result, a representation is defined for *couple* as

COUPLE: **function**(FAMILY-MEMBER,MARRIED-COUPLE)

and the collection *married-couple* is defined as

COLLECTION MARRIED-COUPLE OF **fixed-size-disjoint-set**(2,FAMILY).

The problem statement is also reformulated and a number of statements are added by knowledge acquisition activities. The new problem statement is shown in figure 10.2. Note that the statements enclosed in the box in the figure are captured by the representation of *couple*.

$\mathrm{SORT}(P, FAMILY\text{-}MEMBER)$, $\mathrm{SORT}(Q, FAMILY\text{-}MEMBER)$,
$\mathrm{SORT}(R, FAMILY\text{-}MEMBER)$, $\mathrm{SORT}(S, FAMILY\text{-}MEMBER)$
$grandchild(Q, S)$
$\forall x\ child\text{-}of(P, x) \Leftrightarrow x = R$
$couple(Q) = couple(P) \wedge Q \neq P$
$\forall x \forall y[grandchild(x, y) \Leftrightarrow \exists z(child\text{-}of(x, z) \wedge child\text{-}of(z, y))]$
$\forall x \forall y[child\text{-}of(x, y) \Leftrightarrow parent(y, x)]$
$\forall x \forall y \forall c[child\text{-}of(x, c) \wedge child\text{-}of(y, c) \wedge x \neq y \Rightarrow couple(x) = couple(y) \wedge x \neq y]$
$\forall x \forall y \forall c[couple(x) = couple(y) \wedge x \neq y \wedge child\text{-}of(x, c) \Rightarrow child\text{-}of(y, c)]$

$\neg[couple(x) = couple(x) \wedge x \neq x]$
$\forall x \forall y[couple(x) = couple(y) \wedge x \neq y \Leftrightarrow couple(y) = couple(x) \wedge x \neq y]$
$\forall x \forall y \forall z[couple(x) = couple(y) \wedge x \neq y \wedge couple(y) = couple(z) \wedge y \neq z \Rightarrow couple(x) \neq couple(z) \vee x = z]$
$\forall x \forall y \forall z[couple(x) = couple(y) \wedge x \neq y \wedge couple(x) = couple(z) \wedge x \neq z \Rightarrow y = z]$

Query: find-all x: $parent(S, x)$

Figure 10.2: Example problem reformulated in terms of *couple*

Next, the representation design system performs extended classification on *child-of* which results in the following representations:

COLLECTION CHILD-SET OF disjoint-set(FAMILY-MEMBER)
CHILD-SET-OF: function(FAMILY-MEMBER,CHILD-SET)
COLLECTION PARENT-SET OF fixed-size-disjoint-set(2,FAMILY-MEMBER)
PARENT-SET-OF: partial-function(FAMILY-MEMBER,PARENT-SET)
CHILDREN': 1-1-partial-function(PARENT-SET,CHILD-SET)
PARENTS': 1-1-function(CHILD-SET,PARENT-SET)

Again, statements get added to the problem and it gets reformulated. The result of this is the problem statement in figure 10.3. As in the previous figure, the statements enclosed in the box have their constraints captured by the specialized representations. Also note that the two statements marked by (*) were derived by the rewrite system from the two statements

$\forall x \forall y \forall c[\text{child-of}(x,c) \wedge \text{child-of}(y,c) \wedge x \neq y \Rightarrow \text{couple}(x) = \text{couple}(y) \wedge x \neq y]$
$\forall x \forall y \forall c[\text{couple}(x) = \text{couple}(y) \wedge x \neq y \wedge \text{child-of}(x,c) \Rightarrow \text{child-of}(y,c)]$

Finally, *parent* is classified and the system determines that the query asks for any member of a set that it has already created: a *parent-set*. The query is transformed into

find-the $\text{parents}'(\text{child-set-of}(S))$

and the problem statement defining *parent* in terms of *child-of* is removed.

This leaves following general statements uncaptured

$\forall x \forall y[\text{parent-set-of}(x) = \text{parent-set-of}(y) \Rightarrow \text{couple}(x) = \text{couple}(y)]$
$\forall x \forall y[\text{parent-set-of}(x) \neq \perp \wedge \text{couple}(x) = \text{couple}(y) \Rightarrow \text{parent-set-of}(x) = \text{parent-set-of}(y)]$ $\forall x[\text{parent-set-of}(x) \neq \perp \Rightarrow x \in \text{parent-set-of}(x)]$
$\forall x[\text{couple}(x) \neq \perp \Rightarrow x \in \text{couple}(x)]$
$\forall x\ x \in \text{child-set-of}(x)$.

These get captured by operationalization.

$children'(parent\text{-}set\text{-}of(P)) = \{R\}$
$couple(Q) = couple(F) \wedge Q \neq P$
$\exists z[children'(parent\text{-}set\text{-}of(Q)) = child\text{-}set(z) \wedge children'(parent\text{-}set\text{-}of(z)) = child\text{-}set(S)]$

$\forall x \forall y[child\text{-}of(x,y) \Leftrightarrow parent(y,x)]$
(*) $\forall x \forall y[parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y) \Rightarrow couple(x) = couple(y)]$
(*) $\forall x \forall y[parent\text{-}set\text{-}of(x) \neq \perp \wedge couple(x) = couple(y) \Rightarrow parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y)]$
$\forall x[parent\text{-}set\text{-}of(x) \neq \perp \Rightarrow x \in parent\text{-}set\text{-}of(x)]$
$\forall x[couple(x) \neq \perp \Rightarrow x \in couple(x)]$
$\forall x\ x \in child\text{-}set\text{-}of(x)$

$\forall x \neg[couple(x) = couple(x) \wedge x \neq x]$
$\forall x \forall y[couple(x) = couple(y) \wedge x \neq y \Leftrightarrow couple(y) = couple(x) \wedge x \neq y]$
$\forall x \forall y \forall z[couple(x) = couple(y) \wedge x \neq y \wedge couple(y) = couple(z) \wedge y \neq z \Rightarrow couple(x) \neq couple(z) \vee x = z]$
$\forall x \forall y \forall z[couple(x) = couple(y) \wedge x \neq y \wedge couple(x) = couple(z) \wedge x \neq z \Rightarrow y = z]$
$\forall x\ child\text{-}set(x) \neq children'(parent\text{-}set\text{-}of(x))$
$\forall x \forall y[\ child\text{-}set(y) = children'(parent\text{-}set\text{-}of(x)) \Rightarrow child\text{-}set(x) \neq children'(parent\text{-}set\text{-}of(y))]$

$\forall x \forall y \forall z[(child\text{-}set(y) = children'(parent\text{-}set\text{-}of(x)) \wedge child\text{-}set(z) = children'(parent\text{-}set\text{-}of(y))) \Rightarrow child\text{-}set(z) \neq children'(parent\text{-}set\text{-}of(y))]$

$\forall x \forall y \forall z \forall w[\ child\text{-}set(x) = children'(parent\text{-}set\text{-}of(y)) \wedge child\text{-}set(x) = children'(parent\text{-}set\text{-}of(z)) \wedge y \neq z \wedge child\text{-}set(x) = children'(parent\text{-}set\text{-}of(w)) \Rightarrow w = y \vee w = z]$

Query: find-all x: $parent(S,x)$

Figure 10.3: Formulation of example problem after *child-of* has been classified.

Bibliography

[Amarel68] Amarel, S., "On Representations of Problems of Reasoning About Actions," In Michie, D. (editor), Machine Intelligence 3, pp. 131-171, Edinburgh University Press, 1968.

[Barstow79] Barstow, D., "An Experiment in Knowledge-Based Automatic Programming," Artificial Intelligence, 12, pp.73-119, 1979.

[Bell & Machover 77] Bell, J. and Machover, M., A course in Mathematical Logic, North Holland, 1977.

[Bobrow68] Bobrow, D.G., "Natural Language Input for a Computer Problem-Solving System," in Minsky, M. (editor), Semantic Information Processing, pp.146-226, MIT Press, 1968.

[Cohen86] Cohen, D., "Automatic Compilation of Logical Specifications into Efficient Programs," AAAI86, pp.20-25, 1986.

[Green68] Green, C., "Theorem Proving by Resolution as a Basis for Question-Answering Systems," In Michie, D. (editor), Machine Intelligence 4, Edinburgh University Press, 1969.

[Hayes74] Hayes, P.J., "Some Problems and Non-Problems in Representation Theory," in Brachman, R.J and Levesque, H.J. (editors), Readings in Knowledge Representation, pp. 3-22, Morgan Kaufmann, 1985.

[Johnson-Laird82] Johnson-Laird, P.N., "Ninth Bartlett Memorial Lecture. Thinking as a Skill," Quarterly Journal of Experimental Psychology, 34A, pp. 1-29, 1982.

[Korf80] Korf, R.E., "Toward a Model of Representation Changes," Artificial Intelligence, 14, pp.41-78, 1980.

[Korf83] Korf, R.E., "Learning to Solve Problems by Searching for Macro-Operations," CMU-CS-83-138, 1983.

[Kowalski75] Kowalski, R., "A Proof Procedure Using Connection Graphs," Journal of the ACM, 22, No. 4, 1975.

[Lenat & Brown 84] Lenat, D.B. and Brown, J.S., "Why AM and EURISKO Appear to Work," Artificial Intelligence, 23, pp. 269-294, 1984.

[Marr82] Marr, D., Vision, W.H. Freeman and Company, 1982.

[McAllester82] McAllester, D.A, "Reasoning Utility Package User's Manual," AI Memo 667, M.I.T. Artificial Intelligence Laboratory, 1982.

[McCarthy64] McCarthy, J., "A tough nut for proof procedures," A.I. Project Memo 16, Stanford University, 1964.

[Miller & Schubert 88] Miller, S.A. and Schubert, L.K., "Using Specialists to Accelerate General Reasoning," AAAI88, pp. 161-165, 1988.

[Newell65] Newell, A., "Limitations of the current stock of ideas about problem solving," in Electronic Information Handling, Kent, A,. and Taulbee, O., (editors), Spartan Books, 1965.

[Newell66] Newell, A., "On Representations of Problems," in *Annual research review,* Department of Computer Science, Carnegie-Mellon University, 1966.

[Novak76] Novak, G., "Computer Understanding of Physics Problems Stated in Natural Language," in American Journal of Computational Linguistics, Microfiche 53, 1976.

[Pylyshyn75] Pylyshyn, Z.W., "Do we need images and analogs?" pp. 175-177, TINLAP-1, 1975.

[Rovner76] Rovner, P.D., "Automatic representation selection for associative data structures," Technical Report 10, Univ. of Rochester, Dept. of Computer Science, 1976.

[Sloman71] Sloman, A., "Afterthoughts on Analogical Representations," in Brachman, R.J. and Levesque, H.J (editors), Readings in Knowledge Representation, pp. 431-440, Morgan Kaufmann, 1985.

[Sloman85] Sloman, A., "Why We Need Many Knowledge Representation Formalisms," University of Sussex, Cognitive Science Research Report, 1985.

[Subramanian87] Subramanian, D. and Genesereth, M.R., "The Relevance of Irrelevance," IJCAI87, pp.416-422, 1987.

[Weber83] Weber, K., How to Prepare for the New LSAT, Harcourt Brace Jovanovich, 1983.

[Winston84] Winston, P.H., Artificial Intelligence, Addison-Wesley Publishing Co., 1984.