TR-158

Some Data-base Applications of Constraint Expressions

Richard Weaver Grossman

February 1976

Massachusetts Institute of Technology

Laboratory for Computer Science
(formerly Project MAC)

Cambridge Massachusetts 02139

Abstract

This report presents a novel network-like representation for information, called "constraint expressions" (CE). CE makes use of some of the knowledge-representation techniques developed by Artificial Intelligence research. A CE network consists of points (which represent classes of objects) interconnected by constraints (which represent the relationships which are known to hold among the classes). All constraints are defined in terms of six primitive ones. The data in a CE network is accessed by propagating various kinds of labels through it: Each constraint can be viewed as an active process which looks for certain patterns of labels on some of its attached points, and then propagates new labels to other points when such patterns occur.

The CE representation provides several significant features which are not found in most current data models. First, the same mechanism is used to represent "general" as well as "specific" information. For example, "The sex of Jane Smith is female" is specific, while "Every person has a unique sex which is either 'male' or 'female'" is general.

Second, CE's label-propagation procedure implements logical consistency checking: Data-base integrity can be maintained by checking all new data for consistency with the existing information. Since the data-base can contain general information (representing a "semantic model" of the data-base's application domain), new specific data can be rejected if it is inconsistent with either other specific data or with the general information. Also, the general information can itself be checked for internal consistency.

Third, the CE representation is sufficiently modular and well-defined so that it has a precise formal semantics, which insures that CE's definition contains no hidden ambiguities or contradictions.

Fourth, CE's modularity allows the label propagations to be done in parallel, so that parallel hardware can be used to full advantage.

Acknowledgements

I wish to thank the following persons for contributing to the production of this thesis:

Michael Hammer (thesis advisor) provided many extremely valuable comments regarding both the content and the prose in successive drafts of this document. Also, he and members of his research group (Marvin Essrig and Christopher Reeve) provided many of the examples used here.

Drew McDermott, Robert Moore, Vaughan Pratt, and Guy Steele (of the M.I.T. Artificial Intelligence Laboratory) read an earlier report on this research and provided useful technical comments.

Last but not least, Leah Grossman provided much moral support and comfort during the bad times, and shared my enthusiasm during the good ones -- this document is dedicated to her.

This report reproduces a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science on January 5, 1976, in partial fulfillment of the requirements for the degree of Master of Science.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contracts N00014-75-C-0661 and N00014-75-C-0643.

This research was also supported by the author's National Science Foundation graduate fellowship.

Contents

Part Two -- Epistemology, etc.

Introduction

The work reported in this document concerns a network-like representation for information called "constraint expressions" (CE). This representation has two major features which distinguish it from most others. The first is that it deals comfortably with incomplete information. For example, unlike many other representations, a data-base structured in terms of CE can easily contain information about the class of "all persons" even when the data-base does not contain a complete list of all of them. This feature allows the data-base to contain both "specific" information (such as "Mary Smith is the mother of Jane Smith") and "general" information (such as "each person has a unique mother").

The second major feature is that the CE representation has a well-defined logical semantics, which precisely defines the meaning of every piece of any given CE network. Many other representations lack an adequate logical semantics, which makes it difficult to understand them in a coherent way and hinders one from comparing their relative strengths and weaknesses. In addition, CE's semantics is "procedural" as well as "logical" in that it specifies not only what any given expression means, but also how to compute inferences from it. More specifically, the logical and procedural semantics are specified in terms of how various kinds of "labels" are allowed to propagate through a CE network. The information in a CE data-base is contained in the structure of such a network, and this information is then accessed by moving labels through the net. This kind of semantics encourages one to think of a CE data-base as operating in a highly parallel manner, with each datum acting as an active process which propagates labels.

This document is divided into three main parts. The first gives the technical

details of the CE representation, using examples relating to a hypothetical data-base of census information. The second part is more philosophical than technical -- it compares CE with other representations, discusses why representational issues are important in the first place, and examines some areas in which the CE representation is not adequate. The third and final part is a collection of technical appendices. The topics include such things as parallel hardware and "knowledge about knowledge" (such as "Billy knows who Jane Smith's real father is, and she doesn't know that he knows.") These appendices may be read in any order (or skipped entirely) since each is largely self contained.

The reader should be forewarned that much of part one is not particularly easy reading -- the presentation is organized to minimize the number of forward references, which means that interesting examples occur only after the necessary machinery has been introduced. The main reason for this rather dry "bottom up" format is brevity -- it would be possible to discuss the "big picture" in parallel with the details, but at a cost of perhaps doubling the size of this document. It is expected that the interested reader will skim the entire document first (especially part two) in order to get enough of the big picture to motivate studying the details.

Part One -- Technical Details

The following three sections present the technical details of the CE representation. Section 1 introduces the abstract universe of objects, classes, and constraints in which CE operates, and discusses how information retrieval and inference are accomplished. Section 2 presents the six different primitive constraints which are used (in the current formulation of CE) to structure this universe. Section 3 then uses these primitives to construct more complex constraints, such as those involving boolean functions, transitive relations, and naive probability.

1 The CE universe

1.1 Objects and Classes

The CE universe is composed of atomic objects which can be aggregated to form arbitrary classes. Each particular object either is or is not contained in any given class -- it is impossible for some object to both be in and not be in the same class. Of course, an object may be in more than one class, and a class may have any number of objects in it (from zero to infinitely many). Section 2.2 gives more details about how objects and classes interact.

As a convenient notation, let uppercase names denote classes, and let other names denote objects. These names may contain hyphens and other punctuation. Sometimes a name will be enclosed in single quotes to avoid confusing it with the surrounding text. For example, REGISTERED-VOTERS and OCCUPATIONS name classes, while 'Jane-Smith' and 'lawyer' name objects.

1.2 Constraints

As presented so far, the universe has no structure -- any arbitrary assignment of objects to classes is allowed. Constraints add the necessary structure by constraining the allowable assignments. For example, the constraint that "all REGISTERED-VOTERS are PERSONS" requires that every object assigned to REGISTERED-VOTERS also be assigned to PERSONS. Section 2 defines all the primitive CE constraints by specifying how certain patterns of object assignments (for example, assigning 'Jane-Smith' to REGISTERED-VOTERS) can force other assignments (such as assigning 'Jane-Smith' to PERSONS). These definitions provide the logical semantics for each primitive.

The information in a CE data-base is represented as a network of such constraints connected to the appropriate classes. Both "general" and "specific" information can be represented in the network -- how this feature relates to other data-base schemes is discussed in part two. As a notation, the classes will be drawn as points in the network, and the constraints will be drawn as various kinds of structures connecting the points. The class-points may be named so they can be referred to explicitly in the text (e.g. 'REGISTERED-VOTERS'), but the names themselves are not part of the data-base and do not carry any data-base information. The "meaning" of a class-point such as REGISTERED-VOTERS is carried totally in terms of its connections with the rest of the network, and not at all in terms of whatever external name the class-point has (if any).

1.3 Extension and Intension

Within the abstract universe, the important aspect of a class such as 'REGISTERED-VOTERS' is its extension (the objects which it contains, in this case

presumably all the registered voters). Within the data-base however, the important aspect is its intension (how it is constrained to relate to other classes). In some cases the intension and the extension coincide -- a class in the data-base can be constrained (defined) in terms of an explicit listing of its objects. For example, the class BLOOD-GROUPS can be defined by listing the four blood groups 'a', 'b', 'ab', and 'o'.

However, in many cases the data-base will not contain such complete information about a class (i.e. its extension). For example, consider JANE-SMITH'S-BLOOD-GROUP, which is a one-object class containing an object such as 'ab'. If the data-base does not know Jane Smith's blood group, then it does not know the extension of the class. But, it does know some things about the class in terms of its intension (how it relates to the rest of the network). For one thing, the class can be constrained to be a subclass of the BLOOD-GROUPS class. For another, a person's blood group can be constrained in terms of their parents' blood groups. The important thing to remember is that what the data-base "knows" (via intensions) may only be a small part of what is true in the universe (via extensions). (Note that the meanings of the terms "extension" and "intension" as used in this document are not the same as their meanings in modern mathematical logic. The meanings used here are similar to those used by Pople [1972].)

1.4 Inference

The reason for being concerned with extensions in the first place is that they provide the foundations for the logical semantics of the constraints (as described in section 2), which in turn provide the framework for making inferences from the data-base. Here, "inference" means the process of accessing the data-base in order to acquire information.

The kinds of inference considered here are:

(1) Retrieval -- the user asks questions of the data-base.

(2) Consistency -- the user adds new information to the data-base, and wants to be notified if the new data conflicts with existing data.

(3) Redundancy -- the user adds new data and wants to be notified if the new data is redundant (i.e. is implied by existing data).

This document focuses on the issue of consistency for a couple of reasons. One is that maintaining a consistent data-base in the "real world" is often a very difficult problem, and CE provides one possible solution. The other reason is that **all three forms of inference can be subsumed under consistency checking.** For redundancy checking, a new datum is redundant iff its negation is inconsistent with the existing data-base. For retrieval, a "yes/no" question such as "Is Jane Smith's father's blood group the same as her mother's?" can be answered by checking the assertion that "Jane Smith's father's blood group is the same as her mother's" for both consistency and redundancy: If it is inconsistent, then the answer is "no"; if redundant then the answer is "yes"; if neither then "don't know." Since checking for redundancy means checking the negation for inconsistency, then answer to the above yes/no question is "yes" iff the assertion that "Jane Smith's father's blood group is <u>not</u> the same as her mother's" is inconsistent with the existing data-base. Retrieval for "find" questions such as "Find Jane Smith's hair color" or "Find every retired lawyer living in Nevada" is similar but involves added complications -- this topic is discussed in appendix A.

1.4.1 Labels and Propagation

The consistency-checking inferences are performed by propagating labels through the CE data-base network. A label is an extensional device which relates an object to a class in the network. A label names an object and is "on" a class -- the label can actually be written on a CE network diagram next to a class-point (which "puts the label on the class"). A class may have more than one label on it, and the same label may be used on more than one class. Here are the three kinds of labels ('obj' is some object):

+obj	The class contains 'obj'.
-obj	The class does not contain 'obj'.
=obj	The class contains exactly the one object 'obj' and nothing else.

Note that =obj is a special case of +obj, so anything said below about '+' labels applies equally well to '=' labels.

Labels propagate through the network because each constraint looks for certain patterns of labels on the class-points to which it is attached, and then creates other labels when such a pattern occurs. All of section 2 consists of specifying these patterns for the primitive CE constraints.

1.4.2 Label Collisions

Two labels can "collide" at a common class point in two interesting ways. The first is that **an inconsistency is detected whenever the same class-point gets labeled with both -obj and +obj for some object** -- it is impossible that an object both be in and not be in the same class.

The second kind of collision occurs when a +obj1 collides with an =obj2. In this case, 'obj1' and 'obj2' **have to be the same object.** This is because the class

does contain 'obj1' (from the +obj1 label), and it contains only 'obj2' (from the -obj2 label). Since 'obj1' and 'obj2' are therefore the same, they can be "identified" with each other during the inference. This allows any reference to either to imply a reference to the other. For example, a +obj1 and -obj2 collision will indicate an inconsistency (as above). In general, any class labeled with either object will be implicitly labeled with the other, and these implicitly created labels may propagate in the usual manner.

1.4.3 Initial labelings

To check the consistency of an assertion against the data-base, the assertion is represented as an initial pattern of labels in the network. The labels are then propagated, and an inconsistency is indicated if (for some object 'x') -x and +x (or +x) labels "collide" at the same class-point. If the data-base (without the new assertion) is consistent, then the inconsistency must have arisen either because the new assertion is internally inconsistent or because the assertion is inconsistent with the rest of the data-base (which is the interesting case).

This paragraph introduces two important initial labeling patterns (which are used throughout the rest of this document). Let 'x' be a new object (one that does not already exist in the data-base), and let 'A' and 'B' be two class-points in the net.

(1) If labeling A with +x and B with +x yields an inconsistency, then it means that there is no such 'x'. That is, it is inconsistent to assert that A and B have some object in common. This demonstrates that A and B are mutually exclusive classes.

(2) Similarly, if +x on A and -x on B yields an inconsistency, it means that there is no object which is in A but not in B. This demonstrates that A is a subclass of B.

2 Primitive Constraints

This section discusses the six primitive constraint expressions used in the current formulation of the CE representation.

A CE data-base consists of a set of class-points that are interconnected by a network of these primitive constraints. Since all inferences involving the data-base are performed by propagating labels through the network, a primitive constraint's meaning and behavior can be completely specified in terms of the label patterns it responds to and the labels it propagates on the basis of such patterns. Thus it is easy to add a new kind of primitive constraint without having to worry about possible interactions with previously existing primitives.

In this section, each primitive is described by giving its intuitive meaning, its network symbol, the propagation rules, and some examples. More examples occur in section 3. The more philosophical issues concerning what the primitives "really" mean and how they compare with those of other representations are discussed in part two.

2.1 The Partition Constraint

This constraint represents the partitioning of a superclass into exclusive and exhaustive subclasses. The network symbol for this is shown in figure 2-1a -- the superclass (here, 'A') is drawn on the convex side of the bar, and the subclasses (here, 'B' and 'C') are drawn on the concave side. Note that there may be more (or fewer) than two subclasses, and that the particular left-to-right ordering of the subclasses is unimportant. In figure 2-1a, the class 'A' is partitioned into 'B' and 'C'. This means that every object in 'A' is in exactly one of 'B' or 'C', and that no "extra" objects (those not in 'A') are in 'B' or 'C'.

For a more concrete example, figure 2-1b illustrates the partition of PERSONS into FEMALES and MALES. As an example of a partition with more than two subclasses, figure 2-1c partitions FEDERAL-EMPLOYEES into LEGISLATIVE-EMPLOYEES, EXECUTIVE-EMPLOYEES, and JUDICIAL-EMPLOYEES. Figure 2-1d says that the class of REGISTERED-VOTERS is a subclass of PERSONS -- the unnamed class-point is what is left over (i.e. persons who are not registered voters). Figure 2-1e says that no registered voters are convicted felons (without bothering to name the superclass).

Note that any of the classes may be empty: Figure 2-1 does not say that there are any registered voters. In fact, all the classes in figure 2-1 could be empty (since the empty class can be partitioned into empty subclasses). However, if there are any REGISTERED-VOTERS, then they are constrained to be PERSONS and constrained not to be CONVICTED-FELONS. The notion of subclass such as in 2-1d occurs frequently enough to deserve a simpler symbol: Figure 2-1f uses this symbol, which should be interpreted as an abbreviation for the one used in 2-1d. Similarly, the constraint of figure 2-1g (saying that 'A' is partitioned into exactly 'B' -- they are the same class) has its own symbol, shown in 2-1h.

The five label propagation rules for the partition constraint are diagrammed in figure 2-2. Each of these rules describes a case in which enough information is available (in terms of existing labels on class-points) to enable new labels to be propagated to other class-points; these new labels may then in turn enable further propagations. The left-hand side of each rule gives the relevant pattern of existing labels, and the right-hand side gives the new propagated labels. The ellipses ("...") indicate that there may be more subclasses than are explicitly drawn. In the figure, 'x' is used as the name of the label object -- the

five rules of course apply to any object. They are:

(p1) If an object is in one of the subclasses, then it must be in the superclass and can not be in any of the other mutually exclusive subclasses. In other words, the existing +x label provides the information that 'x' is in one of the subclasses. This information is sufficient to deduce that 'x' must be in the superclass and not in any of the other subclasses. This deduced information is then represented in terms of new labels which are put on the relevant classes. It is in this manner that labels "propagate."

(p2) If an object is not in the superclass, then it can not be in any of the subclasses. This is a consequence of (p1) -- if the object were in any of the subclasses, then it would have to be in the superclass, which is false.

(p3) If an object is in the superclass, and is not in all-but-one of the subclasses, then it must be in the remaining subclass. This is true because the partition is exhaustive.

(p4) If an object is not in any of the subclasses, then it can not be in the superclass. (This too is true because of exhaustiveness).

(p5) If the superclass contains exactly one object, and the object is not in all-but-one of the subclasses, then the remaining subclass must contain exactly that object.

Note from figure 2-2 that (p1) and (p3) are in some sense duals of each other, as are (p2) and (p4). Rather surprisingly, section 3.1 below shows that this one constraint suffices for representing all of the Boolean functions (set-theoretic union, intersection, etc.)

2.2 The Object Constraint

This constraint forces a class to contain exactly one object. The notation for this is to draw the constrained class-point as a small square (instead of a round point). Since

such a class represents exactly one object, it can be named with the lowercase name which is the name of the object.

For example, figure 2-3a states that the president of the US is an executive-branch employee. Figure 2-3b states that there are exactly four BLOOD-GROUPS, and lists them -- this is an example where the data-base has complete (extensional) information about a class. In 2-3a, the data-base certainly does not have complete information about EXECUTIVE-EMPLOYEES -- all it knows is that one of them must be 'the-president-of-the-US'.

The partition constraint is used in 2-3b in order to force the four blood groups to be distinct objects: In CE, the fact that two objects are distinct must be indicated intensionally (i.e. via some constraints in the data-base), instead of just assuming that two objects with different names are different (extensional) objects. Another instance of this distinction (due to Frege) is that "the morning star" and "the evening star" are the same extensional object (the planet Venus) even though they are named differently. Indeed, in some cases it may be difficult or even impossible to decide whether two objects are the same. Note that CE does not require an explicit partition constraint between each pair of distinct objects (which would be quite wasteful) -- all that is required is to be able to infer this distinctness (by, for example, starting a +x label on both the objects and deriving an inconsistency).

The propagation rule for objects is very simple:

(o1) Each object class "broadcasts" an -obj label naming itself.

For example, the object class 'ab' in 2-3b starts an -ab label from itself. The label says that the class contains exactly the one object, which is of course precisely what is forced by the

object constraint. The -obj labels so generated can then interact with other constraints to propagate further. For example, in 2-3b the -ab on 'ab' can propagate a +ab to BLOOD-GROUPS as follows: The -ab on 'ab' entails a +ab on 'ab' (see section 1.4.1), which allows rule (p1) to propagate a +ab to BLOOD-GROUPS.

To prevent a CE data-base from choking itself with too many labels, some technique is needed to have only the "relevant" object classes broadcast their labels during an inference. One solution is to consider an object class to be "relevant" (with respect to a particular inference) iff it gets labeled during that inference. Thus an inference which has nothing to do with blood groups will not cause the blood group objects (e.g. 'ab') to broadcast, since none of them will get labeled.

Note that CE "objects" can be used to represent a wide variety of data-base entities which are not "objects" in the normal English use of the word. In 2-3b, the object 'ab' clearly has no physical existence. In 2-3c, the objects 'female' and 'male' are not particular females or males but are objects denoting these sexes themselves. Since 'female' and 'male' have little meaning in a universe without females and males, there must be a strong connection between the class of FEMALES and the object 'female'. The next subsection discusses what this connection is.

2.3 The Binary Relationship Constraint

This constraint allows the use of binary relations, such as "sex-of", "father-of", and "blood-group-of". Its network symbol is drawn as shown in figure 2-4a, which states that the binary relation SEX-OF holds between the objects 'Jane-Smith' and 'female'. Almost all data-base schemes have some such construct for assigning "values" to

"characteristics" of objects -- what CE's binary relationship constraint does is to extend this notion to encompass arbitrary classes in addition to single objects like 'Jane-Smith'.

This extended case is shown in figure 2-4b. The meaning of it is that the class 'R' (the "range") is the image of the class 'D' (the "domain") under the binary relation 'B'. That is, 'R' contains exactly those objects which are related (by 'B') to some object in 'D'. Figure 2-4c uses this to say that the SEX-OF all FEMALES is 'female'. Using non-object classes for both the domain and the range, figure 2-4d says that the SEX-OF all PERSONS is either 'female' or 'male'.

Note that 'D' in figure 2-4b is not the "domain of the relation 'B'" in the usual sense of the word. Normally, the "domain of a binary relation" is the class of all objects which are related by the relation to something. So in this sense, the "domain of the SEX-OF relation" is all those objects which have a sex. In the sense of "domain" used in this document, the domain is an arbitrary class of objects which act as "input" to the binary relation. The "range" then is the image of the relation restricted to that domain. Furthermore, the "domain" may contain objects which are not related by the relation to anything. For example, figure 2-4d would still be meaningful if 'PERSONS' were replaced by 'THINGS'. Presumably things such as chairs, books, and bacteria have no sex, so they contribute nothing to the range. The changed figure 2-4d would state that the image of THINGS under SEX-OF is the class of 'male' and 'female' -- the fact that bacteria have no sex is irrelevant.

Note that the original figure 2-4d also implies that there is at least one female person and one male person -- the way "image" is defined, an object (e.g. 'female') appears in the range if and only if it is related to some object in the domain (which in this case

must be a female person). If this existential commitment is not desired, the expression can be made weaker as in 2-4e. This one says that the SEX-OF all PERSONS is contained in the class of the two SEXES 'female' and 'male'. The class '%X' may be any subclass of SEXES (including possibly the null class, which would mean either that there are no persons, or that all persons have no sex). As above, it is not required by the definition of "image" that every domain object (a person) be related to at least one range object (a sex) -- one simple means of achieving this kind of symmetry is given in the next subsection (2.4).

One remaining issue is what the 'B' class really represents, and how it may be labeled. Extensionally, a binary relation can be thought of as a class of ordered pairs, each of the form <d,r> where 'd' is an object in the domain, and 'r' is an object in the range. So, for example, SEX-OF can be thought of as a class which contains pairs such as <Jane-Smith,female>, <Billy-Jones,male>, etc. This is just an extensional way of looking at classes such as 'SEX-OF' -- it is not actually necessary that the data-base contain (intensionally) a table of all the ordered pairs occurring in each binary relation.

These <d,r> constructs can be used for labelling binary relation classes. As with all labels, they may be of the form:

+<d,r> The relation so labeled does relate 'd' to 'r'.
-<d,r> The relation definitely does not relate 'd' to 'r'.
=<d,r> The relation relates 'd' to 'r', and relates nothing else.

As with =obj labels, =<d,r> implies +<d,r>. Also note that <d,r>, being an extensional object, can participate in all constraint propagation rules just like 'obj' labels can. For example, in figure 2-4f, a +<some-child,its-mother> label can propagate from MOTHER-OF to PARENTS-OF via rule (p1).

The seven propagation rules for the binary relationship constraint are shown in

figure 2-5. They are:

(b1) If object 'd' is in the domain, and <d,r> is in the relation, then 'r' is in the range. This is an immediate consequence of the definition of "image" given above.

(b2) If an object 'r' is in the range, then there must be some object in the domain which bears the relation to it. It is not known what this object is, so a new name will be used for it (in this case, 'g0037' is being used as the new name). Any implementation of a CE database must contain some provision for generating such new objects. In the rest of this document, these generated object-names will consist of single lowercase letter followed by a 4-digit number. For rule (b2), it is known that the new object is in the domain (hence +g0037 is put on D), and that it bears the relation 'B' to the object 'r' (hence +<g0037,r> is put on B). Some examples below show how this works in practice.

(b3) This is a consequence of (b1). If 'r' is not in the range, and 'd' is in the domain, then <d,r> can not be in the relation. If <d,r> were in the relation, then 'r' would be in the range (using b1), which is false.

(b4) This is another consequence of (b1). If 'r' is not in the range, and <d,r> is in the relation, then 'd' can not be in the domain. Again, if 'd' were in the domain, then 'r' would be in the range (using b1).

(b5) If the domain contains only the one object 'd', and if <d,r> is not in the relation, then 'r' can not be in the range.

(b6) Similarly, if the relation contains only the one ordered pair <d,r>, and the domain does not contain 'd', then the range can not contain 'r'.

(b7) Finally, if the domain contains only the one object 'd', and the relation contains only the one ordered pair <d,r>, then the range must contain only the one object 'r'.

Figures 2-6 and 2-7 diagram some inferences using combinations of these rules. In 2-6a, assume that the data-base contains the information above the dotted line, which states that the sex of Jane Smith is female. If a user of the data-base desires to know the sex of Jane Smith, the user can construct the network below the dotted line. In order to do this, the user must have access to the class-points for 'Jane-Smith' and 'SEX-OF', but the user need not have access to the rest of the network. In particular, the user presumably does not already know about the binary relationship constraint (above the dotted line) which exists in the data-base -- otherwise the user would already know the sex of Jane Smith.

In general, a user interacts with a CE data-base in terms of some fixed set of class-points. The data-base can be viewed as a black box with "terminals" (the set of class-points) with which the user interacts. In simple cases, the user can access the data-base by setting up an initial labeling on some terminals and seeing if the automatic label-propagating inference procedure inside the black box produces an informative result (in terms of label collisions). In more complex cases, the terminals might not directly express the classes in which the user is interested -- in 2-6a there is no terminal for 'the-sex-of-Jane-Smith'. Thus the user must construct the appropriate class (a new terminal) in terms of the existing terminals. This is the purpose of the network fragment below the dotted line in figure 2-6a. In general, a user temporarily adds such a fragment to an existing data-base in order to define whatever new terminals are necessary for the current inference. After the inference, the temporary fragment can of course be deleted.

Having constructed the fragment below the dotted line in 2-6a, the user knows that the object-class 'x' (the name is unimportant) is constrained to be the sex of Jane

Smith. That is, the new terminal 'x' is constrained to be a class containing a single object, that object being the sex of Jane Smith. Now, the object-class 'x' is known (intensionally) to be the sex of Jane Smith, but presumably there is some already existing object in the data-base (in this case, 'female') which is equivalent to 'x' but is more "interesting" (in that the user and/or the data-base know more about 'female' than they do about 'x'). The user's task is to set up an initial labeling such that the propagation procedure can be used to find such an existing object.

An initial labeling (and the subsequent inference) which does this is shown in figure 2-6b, which is a copy of 2-6a with the addition of labels. The number in braces preceding each label is used to indicate the order in which the labels are created by the propagation rules -- these numbers are only for convenience in studying the diagram and are not actually needed by the inference procedure. Following the number is the propagation rule used to create the label, so for example "{2,b1} +x" means that the +x label was created at time 2 via rule (b1). (The following English description of how the labels are propagated is rather cumbersome, but it contains no more information than the labels themselves do.) The {0} +x label is the user's initial labeling which starts the inference -- it says that the terminal 'x' (which is intensionally the sex of Jane Smith) contains exactly the object 'x'. The two labels numbered {1} are created by rule (b2) -- the =x on 'x' automatically entails a +x on 'x' (as in section 1.4.1), which allows (b2) to generate a new object (here, g0038) and create the new labels. The {2,b1} +x label is created by applying (b1) to the two labels just created in step {1}. Rule (o1) allows 'female' to broadcast an =female (label number {3,o1}) which collides with the +x already on 'female'.

As described in section 1.4.2, this kind of collision means that 'x' and 'female' are

the same (extensional) object. The collision can be made known to the user, who thus knows that 'x' (the sex of Jane Smith) is just another name for 'female'. Note that all the user had to do was to create the network below the dotted line and then set up an initial labeling. From that, the label-propagating inference procedure concluded that 'x' and 'female' are the same object, and that conclusion was given to the user. At no time did the user have to be concerned with the structure of network above the dotted line (which represents the existing data-base).

All of this may seem to involve an excessive amount of effort for performing such a simple inference, but exactly the same technique can be used in more complex cases. Figure 2-7 diagrams the inference that Jane Smith's female parent is Mary Smith. Again, assume the structures above the dotted line are part of the data-base, and those below the line are constructed by the user for the purpose of this one inference. Note that the data-base contains both "specific" data (such as that Mary Smith is Jane Smith's mother) and "general" data (such as that MOTHER-OF and FATHER-OF are the two cases of PARENTS-OF). Note also that not all of the allowed label propagations are shown in 2-7 -- it shows only those that are relevant to the inference.

The user-constructed piece of network below the dotted line states that 'x' is a parent of Jane Smith and that 'x' is a female. The question the user wants to ask is "Is 'x' Mary Smith?" Note that this is a simpler question than "find the object(s) in the data-base which are equivalent to 'x'" -- such "find" inferences (akin to figure 2-6's "find the sex of Jane Smith") are discussed in appendix A.

Having constructed the lower piece of network, the user must now set up an initial labeling. The way to infer that 'x' is 'Mary-Smith' is to assume that 'x' is not Mary

Smith and then see if the inference procedure derives an inconsistency (via the collision of some "+" and "-" label). To assume that 'x' is not Mary Smith, the user puts a -x label on the object-class 'Mary-Smith'. The rest of the initial labeling consists of +x on 'x' and +Mary-Smith on 'Mary-Smith'. These are all marked as number {0} on the diagram. The label-propagating inference procedure then uses this initial labeling to derive an inconsistency -- at point 'A' there is a collision of a +x and a -x. Notice of this collision can be given to the user, who then knows that 'x' is indeed 'Mary-Smith', since it is inconsistent to assume otherwise. (Another possible interpretation of this inconsistency is that there is no such 'x', meaning that Jane Smith does not have any female parent. Presumably the user assumes that she does have one, so this interpretation is ruled out.)

Note that the inconsistency is detected via a collision at point 'A'. By propagating the labels in a different order, the collision could have been made to occur somewhere else. Thus the exact place where the inconsistency is detected is not important -- it is only important that it be detected somewhere.

One notational shortcut used in figure 2-7 is that when a "+" label naming some generated-object would be placed on an object-class, the name of the object-class is used instead of the generated-object's name. For example, the {2,b2} +<Jane-Smith,x> label on PARENTS-OF is written down immediately, instead of going through the full process of (1) generating a new object (here, 'g1234'); (2) propagating a {2,b2} +g1234 label to 'Jane-Smith' (and a {2,b2} +<g1234,x> label to PARENTS-OF); (3) creating an -Jane-Smith on 'Jane-Smith' via rule (o1); (4) Colliding the +g1234 with the -Jane-Smith which then identifies the two objects; (5) then finally using 'Jane-Smith' in all labels where 'g1234' is used (which is legal because the two objects have been identified with each other). The

end result is that 'Jane-Smith' is used in place of 'gi234', so there was really no reason to have generated the 'gi234' in the first place.

In general, this shortcut is legal because the generated-object "+" label will immediately collide with the "=" label on the object class, which identifies the generated object with the object-class object. Thus either may be used in place of the other, so it is legal to use the object-class name instead of the generated-object name. In this way, the generated-object name never appears on the diagram. This shortcut will also be used wherever applicable in all the rest of the examples.

2.4 The Inverse Constraint

This construct constrains binary relations to be inverses of each other. For example, PARENTS-OF and CHILDREN-OF are inverse relations in that if 'p' is a parent of 'c', then 'c' is a child of 'p'. The network symbol for this is given in figure 2-8a -- since "inverse" is symmetric, it does not matter which relation is connected to which end of the symbol. The three propagation rules for the inverse constraint are shown in figure 2-8b:

(inv1) If +<d,r> is on one side of the symbol, put +<r,d> on the other side.

(inv2) If -<d,r> is on one side, put -<r,d> on the other. (For example, if 'p' is not a parent of 'c', then 'c' is is not a child of 'p'.)

(inv3) Finally, if =<d,r> is on one side, put =<r,d> on the other.

Things may be said about a binary relation in a symmetric manner by using both the relation and its inverse. For example, figure 2-8a is an improvement over figure 2-4e. Like 2-4e, it states that the SEX-OF all PERSONS is some subclass (here called 'XX') of

the SEXES 'male' and 'female'. As discussed in section 2.3, figure 2-1e does not imply that every person has a sex. However, 2-9a does imply this. Since PERSONS is the image of XX under BEINGS-OF-SEX, for every person there must be some object in XX such that <the-xx-object,the-person> is in BEINGS-OF-SEX (using rule 5E). The inverse constraint in 2-9a then forces that <the-person,the-xx-object> be in SEX-OF. That is, 'the-person' has a sex (which is 'the-xx-object'). Note that none of this requires that a person have one and only one sex -- this topic of "1-1 functions" is discussed in detail in section 2.6.

As another example, figure 2-9b states that SPOUSE-OF is its own inverse, which is means that it is a symmetric relation.

2.5 The World Constraint

So far, we have assumed that the information in a data-base is constant over time and represents a single "point of view." That is, it has been tacitly assumed that the extension of a class such as REGISTERED-VOTERS is static and well-defined. However, in the real world many things change over time. The class of REGISTERED-VOTERS gains and loses objects as new people register and old ones die or let their registration expire. Also, the extension of a class can vary depending on the particular "point of view" which is taken. For example, some users of a census data-base might want to consider all DRAFT-EVADERS to be CRIMINALS, while other users might not.

Fortunately, there is a rather simple mechanism which helps solve both these problems, and which will come to be relied on more and more in following sections of this document. The basic idea is to support multiple "worlds" in the data-base. A world can be thought of as a physical or metaphysical situation, such as the physical situation of "April 3,

1946, at 10:23 am", or the metaphysical situation of "what Jane Smith believes to be true." The latter might be important if Jane Smith is a data-base user who wants to add information to the data-base that conflicts with information that some other user wants to add -- the data-base should then be operating in different worlds when it is being used by Jane Smith and when it is being used by the other user. Section 3.5 discusses many other such applications of worlds.

2.5.1 Extension

To support multiple worlds, it is necessary for the data-base to be able to contain assertions relative to the various worlds and to make the appropriate inferences from them. Since the inference process deals only with propagating labels, the inferences can be relativized by tagging each label with the world that created it. For example, if 'w' is the world of "April 3, 1946, at 10:23 am" then the label "+x/w" will denote a +x label relativized to that world, and similarly for "-" and "=" labels. For example, having a +Jane-Smith/w label on REGISTERED-VOTERS states that Jane Smith is a registered voter in world 'w', without making a commitment one way or the other as to whether Jane Smith is a registered voter in other worlds.

For all of the label propagation rules given so far, it is necessary to add the stricture that two labels may interact only if they are tagged with the same world. For example, in figure 2-10a, rule (p4) can not propagate a -x to class 'A', since the -x labels on 'B' and 'C' belong to different worlds. Also, every label which is propagated during an inference must be given the same world-tag as the label(s) which caused the propagation. For example, in figure 2-10a, if point 'C' were labeled -x/w, then 'A' could be labeled (using

rule p4) with -x/w -- the 'w' world-tag being required to relativize it properly.

Furthermore, world-tags must be taken into account by the object constraint's rule (o1): An object-class such as 'the-president-of-the-US' represents the same intensional object in all worlds, even though extensionally it may be different (there being different presidents at different times). Therefore if 'obj' is an object-class, it is allowed to broadcast an =obj/w label for any world 'w'. As in section 2.2, the data-base can avoid choking itself with spurious labels by having an object-class generate an =obj/w label only when the class is reached by some existing label with a world-tag of 'w'.

By convention, the world-tag used on the initial labeling which starts an inference will be 'inf' (an abbreviation for "current inference"), and every label written without an explicit world-tag will be implicitly tagged with 'inf'.

In addition to using worlds for tags on labels, it is desirable to be able to refer to them explicitly as objects. That is, 'inf', 'w', and all other worlds will be treated in the same manner as other CE objects ('Jane-Smith', 'female', etc). For example, if 'w.1' and 'w.2' are both worlds, then the label +w.1/w.2 on some class means that the world-object 'w.1' is in the class, relative to world 'w.2'. The use of such labels is discussed below; for now, it is first necessary to describe what a world-object such as 'w.1' "really is" -- it is clearly not the same sort of thing as a person (e.g. 'Jane-Smith') or a sex (e.g. 'female'). Since a world is a specification of some state of affairs (physical or metaphysical), and since the CE universe models all information in terms of assigning objects to classes, it follows that a world "really is" such an assignment. These assignments need not be exhaustive -- the world 'w' might assign 'Jane-Smith' to be in REGISTERED-VOTERS, assign 'John-Smith' not to be in REGISTERED-VOTERS, and make no commitment one way or the other regarding

'Mary-Smith'.

For consistency with material to be presented below, a world-object should be viewed as representing a set of "allowable assignments." That is, if world 'w' makes no commitment regarding the assignment of 'Mary-Smith' to REGISTERED-VOTERS, then the set which 'w' represents will contain (as allowable assignments) both "Mary-Smith is in REGISTERED-VOTERS" and "Mary-Smith is not in REGISTERED-VOTERS." In brief, the world 'w' allows both possibilities. Note that viewing the assignment-sets this way means that if one assignment-set is a subset of another one, then the smaller assignment-set corresponds to the stronger world (in that it has fewer cases of allowing both possibilities).

Now, having given some solidity to the rather philosophical concept of "world", consider again how world-tags are used in labels. A label relates an object to a class -- the label "+x/w" on class 'C' means that the object 'x' is contained in 'C' relative to the world 'w'. Another way of stating this is that in world 'w' it is true that 'x' is in 'C'. Now such a thing can be nested: Consider "in world w.1 it is true that in w.2 it is true that 'x' is in 'C'." (A less awkward but rather anthropomorphic way of putting this is "w.1 believes that w.2 believes that 'x' is in 'C'.") This nesting will be represented by using the label "+x/w.2/w.1" on 'C' -- the object 'x' is relative to world 'w.2', which is in turn relative to 'w.1'. The main point of this paragraph is that the world-objects used to relativize objects in labels may themselves be relativized (to any depth), giving rise to arbitrarily long world-tags.

Like other objects, world-objects can be grouped into classes. By convention, such world-classes will have names beginning with "W-". For example, the world-class W-CATHOLIC-FAITH can be defined to contain all worlds which are consistent with orthodox Catholic religious dogma. There may be many different world-objects in such a

world-class -- for instance, in W-CATHOLIC-FAITH two world-objects might agree on all assignments except those involving SAINT-PAUL'S-BLOOD-GROUP (i.e. these two worlds assign Paul's blood group differently). Since Catholic dogma takes no position on what Paul's blood group is, both these world-objects are consistent with the Catholic faith and thus both belong in the world-class. Extensionally, W-CATHOLIC-FAITH will contain a great many such world-objects, differing with each other regarding inessential details. However, all these world-objects will agree on those details which are important in Catholic dogma -- that a divine Christ existed, that Mary was a virgin, that the Pope is infallible, etc.

2.5.2 Intension

As with other classes, it is not necessary that the data-base contain an exhaustive extensional listing of all the world-objects in a world-class in order for the world-class to be useful intensionally. For example, figure 2-10b states that all worlds of W-CATHOLIC-FAITH are also worlds of W-JUDEO-CHRISTIAN-FAITH (which is meant to represent all the beliefs held in common among Judeo-Christian religions: monotheism, the Ten Commandments, etc.) Note that the smaller class (in terms of the subclass constraint in 2-10b) is the stronger one (in terms of what is believed): The Catholic beliefs include all of the general Judeo-Christian beliefs, but not conversely. A more general way of looking at this is that every situation (world-object) which is consistent with the "beliefs" of the smaller (stronger) world-class must necessarily be consistent with the beliefs of the larger (weaker) one. And indeed, a +w (or whatever) label can propagate from the smaller class to the larger via rule (p1). Note that the use of "weaker" and "stronger" here (with regard to

classes of world-objects) parallels the above use of these terms to describe the assignment-sets for individual world-objects. The remainder of this section deals only with world classes -- assignment sets are taken up again in section 2.6.

Now, just as world-tags are needed in labels to relativize them during inferences, some constraint expression is needed to relativize information in the data-base. This constraint expression is the "world constraint," the network symbol for which is shown in figure 2-10c. Its meaning is that 'B' is a subclass of 'A' in exactly those worlds which are in the world-class 'W'. (Section 3.5.3 shows how constraints in addition to the subclass one can be relativized.)

A simple example using the world constraint is shown in figure 2-10d. This states that the Catholic faith believes that Mary is a virgin. More formally, it states that every world which is consistent with the Catholic faith is necessarily one of the worlds in which Mary is a virgin. The subclass constraint used in 2-10d between W-CATHOLIC-FAITH and W-VIRGIN-MARY is needed because there might be worlds in which Mary is a virgin but which are not consistent with the Catholic faith (such as those worlds consistent with Protestant faiths that assert the virginity of Mary but deny the infallibility of the Pope). That is, the Catholic faith imposes stronger constraints than just the virginity of Mary.

The propagation rules for the world constraint are as shown in figure 2-11. The "/..." at the end of the world-tags indicates that the exact nature of the rest of the world-tag is unimportant for these rules, and that any "tail" may be uniformly substituted for the "/...". The propagation rules are:

(w1) If world 'w' is contained in 'W', this means by definition of the world constraint that

'B' is a subclass of 'A' (in world 'w'). Therefore, if some object 'x' is contained in 'B' (in world 'w'), the object 'x' must also be contained in 'A' (in world 'w'). In other words, the +w label on 'W' "enables" the subclass constraint for all objects with world-tag 'w', and this enabled subclass constraint operates in the same manner as rule (p1).

(w2) Similarly, the enabled subclass constraint operates in the same manner as rule (p2): If x/w is not in the superclass, then it can not be in the subclass.

(w3) This rule applies when a world constraint is explicitly disabled by a -w label on 'W'. Rather surprisingly, such negative information can be used in an active manner (instead of just passively refusing to enable the subclass constraint, which is what having no label on 'W' does). Since the -w label means that in world 'w' the class 'B' is not a subclass of 'A', then there must be some object (relative to 'w') which is in 'B' but not in 'A'. As with rule (b2), it is not known which (extensional) object this one might be, so a generated object is used. In this case, the generated object 'g0152' is constrained to be in 'A' via the +g0152/w label, and constrained not to be in 'B' via the -g0152/w.

(w4) If an object (in some world) is contained in 'B' but not in 'A', then 'B' can not be a subclass of 'A' in that world.

(w5) If 'B' contains exactly one object 'x' (in world 'w'), and if that object is contained in 'A' (in world 'w'), then 'A' contains all of 'B' (in 'w'). Thus 'B' is known to be a subclass of 'A' in world 'w'.

Figure 2-12 shows an inference using some of these rules. The task is to show that in all worlds where 'P' is a subclass of 'Q', then in those worlds 'P' must be a subclass of 'Q'. Although it seems trivial, this example does show how the propagation rules interact. 'WA' is the world-class for the first "P is a subclass of Q" and 'WB' is the world-

class for the second one. The task is now to show that every world in 'WA' is necessarily in 'WB' (i.e. that 'WA' is a subclass of 'WB'). As described in section 1.4, the usual manner of doing this is to assume that 'WA' is not a subclass of 'WB'. Then some object 'x' is in 'WA' and not in 'WB' (and its world-tag is 'inf' since it is starting the inference.) After the user sets up this initial labeling, the inference procedure can start propagating. The +x/inf on 'WA' can do nothing for the moment, but the -x/inf on 'WB' can. Rule (w3) is activated by it and thus can create the generated-object labels which assert that 'WA' is not a subclass of 'WB'. Then rule (w1) can propagate the generated "+" label through the enabled left-hand world constraint, causing a collision at point 'Q' which indicates an inconsistency. Note that rule (w2) could have been used instead to propagate the "-" label, which would have caused a collision at point 'P'.

2.6 The Typical-Member Constraint

This constraint allows the data-base to make use of a "typical member" of a class. Consider representing the information that "each person has a unique mother." It is easy enough to state for any particular person such as Jane Smith that she has a unique mother -- this is shown in figure 2-13a. Thus one way to represent "each person has a unique mother" would be to create a structure similar to figure 2-13a for each individual person. Needless to say, using this approach is very expensive if there are more than just a few individual persons, and it has the added disadvantage of requiring complete extensional information concerning all the members of the class PERSONS.

What the typical-member constraint allows is the ability to state that the typical person has a unique mother. Note that "typical" is used here in the sense of "archetype"

and not in the sense of "usual" -- all persons have a unique mother, not just most of them. The network symbol for a simple version of the typical-member (or "t-m") constraint is shown in figure 2-13b. This states that the object 'obj' is a typical member of the class 'CL'. This constraint is used in figure 2-13c to state that "every person has a unique mother." In the following text, the point 'CL' will be referred to as the "input class" of the t-m constraint, and the point 'obj' will be referred to as the "t-m object".

Intuitively, the t-m constraint should have the following behavior: When some "+" label (such as +p/w) occurs on the input class, it is known that 'p' is in that class. Therefore, what is true of the typical member must be true of 'p'. Therefore, the t-m object can be "bound to" 'p' by putting an -p/w label on the t-m object. Then this -p/w can cause further propagations.

The one problem with this behavior is that only one object can be bound to the t-m object at any given time. Violating this restriction can cause severe problems. For one thing, if 'p.1' and 'p.2' are both in the input class (as indicated by having both +p.1/w and +p.2/w labels on the input class-point), then binding them both to the t-m object at the same time would imply that they are both the same object. In a similar manner, all objects in the input class would be simultaneously identified with the t-m object. Furthermore, in figure 2-13c 'the-mother' is obviously not the same (extensional) object for every person -- it too must be "bound" in some manner.

Although the "one binding at a time" restriction avoids these problems, it has its own difficulties. The most obvious is that it may be necessary to refer two (or more) different members of the input class during a single inference. Another difficulty is that some facility is needed for "erasing" all the consequences of one binding (i.e. all the labels

which that binding caused to be propagated) before performing the next binding.

One way to solve this "erasing" problem is to tag all the labels which propagate as a consequence of each particular binding. Then it would be a simple matter to identify which labels should be erased, since these labels would be the ones with the tag. The most straightforward way to implement such tagging is in terms of the existing world mechanism -- each different binding can be tagged with a unique (newly generated) world-tag. Indeed, if this world-tagging is done then it is no longer necessary to keep the "one binding at a time" restriction at all. For example, the above =p.1 and =p.2 labels which come from different bindings (at the same time) can not interact with each other because they will have different world-tags -- they will be something like =p.1/w2034 and =p.2/w2035. Thus it turns out that the "one binding at a time" restriction can be replaced by a "one binding per world" technique, which avoids the difficulties caused by the sequential nature of "one binding at a time."

To properly implement "one binding per world" it is necessary for the inference process to keep a record of which "parent" world each binding's world-tag relates to. That is, if =p.1/w on PERSONS causes a binding of =p.1/w2034 to 'the-person', then it is necessary to remember that 'w2034' is really just a copy of 'w', with the additional restriction that 'the-person' is bound to 'p.1'. This being the case, 'w2034' labels should be allowed to interact with 'w' ones, even though 'w2034' can not interact with other worlds in general. (In particular, 'w2034' must not be allowed to interact with worlds which represent other bindings of 'the-person'.)

More formally, 'w2034' is a stronger world than 'w' -- 'w2034' disallows the assignment of anything other than 'p.1' to 'the-person'. This being the case, the stronger

world-tag may be validly substituted for the weaker one. This is valid because if an assignment is forced in the weaker world (as indicated by the presence of a label with that world-tag), then the same assignment must be forced in the stronger world (since this world allows fewer arbitrary assignments). Since the assignment is forced in the stronger world, that fact may be indicated via a label with the stronger world as its world-tag. For an example of this, in figure 2-13d the -x/w on 'B' and the -x/w2094 on 'C' can interact (via rule p4) to propagate a -x/w2034 to 'A' if 'w2034' is stronger than 'w'. This happens because the -x/w can have -x/w2034 substituted for it, which allows rule (p4) to propagate the -x/w2034 to 'A'.

Since t-m constraints may be nested (as in an example below), the process of generating a stronger world for each binding can give rise to a multi-layer tree of worlds. It is a tree and not just a linear chain because a single parent world can give rise to many different binding-worlds. Figure 2-13e shows the (one-layer) tree for the example above, in which 'w2034' and 'w2035' are both bindings created from 'w'. The existence of nesting allows such trees to be more than one layer deep. Regardless of depth, a given world in the tree is stronger than all the worlds "above" it in the tree (i.e. its parent, its parent's parent, etc.)

As an example of nesting, if a -p.1/w2094 label occurs on the input-class of some other t-m constraint, then that constraint can propagate a binding of (for example) -p.1/w2037, where 'w2037' is a new world. Figure 2-13f shows the "world tree" extended by this new binding: The new world 'w2037' has 'w2094' as its parent, which in turn has 'w' as its parent. In general, a world such as 'w2037' is allowed to interact with all the worlds above it in the tree (here, 'w2094' and 'w'), because all the worlds above a given one are just

weaker versions of it (they lack one or more bindings).

In order to handle nesting properly, it is necessary to be able to explicitly refer to the world-tags generated by the binding process. Figure 2-13g shows the full version of the t-m constraint. In addition to the input class and the t-m object, it has an "input world-class" ('W-IN') drawn on the input class side, and an "output world-class" ('W-OUT') drawn on the t-m object's side. (The in and out world-classes may be drawn either above or below the centerline of the t-m constraint symbol -- they are distinguished only by which side they are on.) The world-tags generated during the binding process are put on the W-OUT class -- this is what makes it possible to use these generated tags in other constraints. W-IN exists primarily for efficiency -- it acts as a "filter" in that an object in the input class (such as indicated by +p/w) will be considered for binding only if 'w' is in W-IN. That is, the meaning of the t-m constraint is that the t-m object is typical of the input class relative to the worlds in the input world-class. If an input world-class is not specified for a given use of the t-m constraint symbol, then it applies to all worlds (i.e. there is no prior "filtering.")

The two label propagation rules for the t-m constraint are shown in figure 2-14:

(tm1) If an object is in the input class and the object's world-tag is in the input world-class, then a binding can be created (as discussed above), and the binding's newly generated world is propagated to the output world-class (if any). The binding's world is also noted in the world tree. (The diagram shows the world tree both before and after the new binding.) If there is no input world-class, then any object in the input class can be used to create a binding (regardless of the object's world-tag).

(tm2) This is the converse of (tm1): If a binding does exist, then the object which is bound to the t-m object is necessarily in the input class and the world which caused the binding is

necessarily in the input world-class. The essence of (tm2) lies in defining exactly when a binding does exist. Referring to figure 2-14, there are three conditions which must hold. The first is that some object (here 'x') must be bound to the t-m object. The second is that the binding world (here 'w.2') for 'x' must be in the output world-class. The third condition is that the world (here 'w.1') which "believes" that 'w.2' is in the output world-class must be above 'w.2' in the world tree (the dotted line between 'w.1' and 'w.2' in the world-tree diagram indicating that 'w.1' need not be immediately above 'w.2'). That is 'w.2' must be a stronger version of 'w.1' (as created by some binding). Taken together, these three conditions define what a binding is -- it can be seen from the top half of figure 2-14 that rule (tm1) satisfies these conditions when it creates a new binding. Note that (tm1) states that the output world-class contains all the binding-worlds; (tm2) states that it contains only the binding-worlds.

Figure 2-15 shows an inference using both (tm1) and (tm2). The top part of 2-15 states that the class of GRANDPARENTS consists of exactly those persons who have children who have children. To see that this is the case, consider the world-class 'WX'. One one hand, WX is defined to be the output world-class for the t-m constraint involving GRANDPARENTS. From this constraint, WX contains exactly those worlds in which 'the-person' is bound to some grandparent. On the other hand, WX is also defined to be the output world-class for the t-m constraint involving THE-GRANDCHILDREN. From this constraint, WX contains exactly those worlds in which 'the-grandchild' is bound to one of the grandchildren (and hence THE-GRANDCHILDREN is not an empty class). Since the same output world-class WX is used for both these t-m constraints, it means that every world in which 'the-grandchild' is bound to one of THE-GRANDCHILDREN is also a

world in which 'the-person' is bound to one of the GRANDPARENTS (and vice-versa). Now, since THE-GRANDCHILDREN are in fact constrained by the two binary relationship constraints to be the grandchildren of 'the-person', it follows that 'the-person' is bound to some grandparent in exactly those worlds in which 'the-person' is bound to some person who has a non-empty class of grandchildren.

To see how this works in practice, the middle part of figure 2-15 contains the information that Jane Smith is a child of Mary Smith. Now, suppose the lower part of 2-15 (below the dotted line) represents new information to be added to an existing data-base (above the dotted line). This new information states that Billy Jones is a child of Jane Smith. Now, from this new information, it should be possible to infer that Mary Smith is a grandmother (even though that the new information makes no reference at all to Mary Smith, and indeed the user who adds the information need not even know that Mary Smith exists).

The inference is started with a {0} -Billy-Jones/inf label on Billy-Jones and a {0} -Jane-Smith/inf on Jane-Smith. By step {4} there are labels on CHILDREN-OF stating that Jane is a child of Mary and that Billy is a child of Jane. At step {7} Mary is bound to 'the-person' (the typical member of PERSONS). From there, two applications of rule (b1) yield the fact at step {9} that Billy is one of THE-GRANDCHILDREN of Mary (who is still bound to 'the-person'). From this, Billy is bound to 'the-grandchild' at step {10}, and the binding world 'w0002' is put on the output world-class 'WX'. This world interacts with the existing binding of Mary to 'the-person' to propagate (via rule tm2) the +Mary-Smith label to GRANDPARENTS. Note that in order to apply (tm2), the -Mary-Smith/w0001 label on 'the-person' must be treated as though it were

-Mary-Smith/w0002 -- this is legal because 'w0002' is stronger than 'w0001' and thus 'w0002' world-tags can be substituted for 'w0001' ones (Refer back to figure 2-13d for a simpler example of this kind of interaction.)

The conclusion at step {11} is that Mary Smith is a grandmother in world 'w0001'. This means that Mary is a grandmother in 'inf', also. Consider: If Mary were not a grandmother in 'inf', then a label to that effect (i.e. -Mary-Smith/inf) could be placed on GRANDPARENTS without any inconsistency. However, such a label can interact with the existing +Mary-Smith/w0001 (since 'w0001' is stronger than 'inf' and can thus be substituted for it) to cause an inconsistency. Thus Mary is indeed a grandmother (in 'inf'). To put this another way, the world 'w0001' differs from 'inf' only in the presence of some additional bindings. Therefore all labels which do not refer to these bindings are as valid in 'inf' as they are in 'w0001'. After all, 'w0001' is really only a bookkeeping device used to prevent invalid interactions with other possible bindings of 'the-person', and is otherwise completely equivalent to 'inf'.

3 Using the Primitive Constraints

This section describes some of the useful non-primitive constraint expressions that can be constructed from the primitives of section 2. The structure of this section parallels that of section 2, so that (for example) section 3.1 describes some uses of the primitive constraint introduced in section 2.1.

3.1 Using the Partition Constraint

3.1.1 Taxonomies

The simplest combination of partition constraints involves arranging them in a taxonomic hierarchy such as figure 3-1. This allows a great saving of space, since (for example) it is not necessary to have an explicit subclass constraint between REDWOODS and PHYSICAL-OBJECTS -- it is implicit in the structure of the tree. Indeed, if 'x' is known to be a redwood (as indicated by "→x/..." on REDWOODS), then 'x' is known to be a PHYSICAL-OBJECT (by applying rule p1 four times). Hierarchical structures such as figure 3-1 are used in many kinds of "semantic network" representations -- part two of this document compares some of these with CE.

3.1.2 Intersection and Union

Taxonomies such as figure 3-1 have a highly disciplined structure -- each class-point occurs at most once as the superclass in some partition constraint and at most once as a subclass in some other partition. By relaxing this discipline somewhat it is possible to represent all of the Boolean functions (intersection, union, complement, etc.) Intersection and union are treated here -- complement is treated in 3.1.3.

Consider intersection: Given two classes 'A' and 'B', it is desired to constrain the class 'C' to contain exactly those objects which are contained in both 'A' and 'B'. Figure 3-2a is a first attempt at this. In 3-2a, 'C' is a subclass of both 'A' and 'B', so every object in 'C' is necessarily in both 'A' and 'B' and is hence in the intersection. Thus 'C' is no larger then the intersection. However, 'C' may be a good deal smaller than the intersection -- in the worst case, 'C' could be empty (in which case all of 'A' would be in 'A!' and all of 'B' would be in 'B!'). What is necessary is to prevent those objects which belong in the intersection 'C' from mistakenly ending up in 'A!' or 'B!'. Now, if 'A!' were constrained to be mutually exclusive with 'B', then no object in 'A' which is also in 'B' could end up in 'A!'. Hence 'C' would have to contain all the objects in the intersection. Figure 3-2b adds this constraint that 'A!' be exclusive with 'B', as well as the symmetric one that 'B!' be exclusive with 'A'.

Figure 3-2b does indeed constrain 'C' to contain all of the intersection: If some object 'x' is in both 'A' and 'B', then it should be possible to infer that 'x' is in 'C'. That is, starting a +x label from both 'A' and 'B' should result in a +x on 'C'. Referring to 3-2b, such a +x on 'B' would entail a -x on 'A!' by rule (p1). Then the +x on 'A' and the -x on 'A!' would entail a +x on 'C' by rule (p3), and we are done.

Having represented the intersection function, it turns out that the union function comes for free. Consider the classes 'A2' and 'B2' in figure 3-2b. 'A2' contains all of 'B' plus all of 'A' which is not in 'B'. That is to say, 'A2' contains exactly 'A' union 'B'. Similarly, 'B2' contains exactly 'A' union 'B'. Since 'A2' and 'B2' are extensionally the same class, nothing is lost by using the same intensional class for them: Figure 3-2c does this and names the combined class 'D'. Thus in 3-2c the class 'C' represents the intersection of 'A'

and 'B', and the class 'D' represents their union.

Since intersection and union occur fairly frequently, it can become awkward to have to repreatedly write out copies of figure 3-2c. As a more convenient notation, figures 3-2d and 3-2e show the abbreviated network symbols for intersection and union, respectively. One way to view them is as "macros" for the full structure of figure 3-2c -- in an implementation, the union and intersection boxes would be expanded into the network of figure 3-2c, and that would be what is stored in the data-base.

Another way to view them is as "subroutines" -- instead of being expanded, they can be implemented as new primitive constraints, which have a specified input-output behavior (as determined by propagation rules) without any regard for what "fine structure" the boxes might have. Not only does this save space (since the boxes are not expanded), it also saves time during the inference process -- the specialized propagation rules for the intersection box (for example) can operate directly in terms of the classes 'A', 'B', and 'C', without having to worry about the internal state of such points as 'AI' and 'BI'. Of course, adding a new primitive constraint does increase the complexity of the inference process in that it adds more propagation rules. In general, the decision as to which non-primitive constraint expressions should be left as macros and which should be made into primitives is a matter of implementation trade-offs. For the purposes of the rest of section 3, it suffices to note that just because some constraint expression is called "non-primitive" (in that it can be represented in terms of the primitive constraints) does not mean that it must be expanded into a (possibly large) network of primitives in an actual implementation.

As an example, figures 3-3 and 3-4 show the propagation rules for the intersection and union "primitives", respectively. The behavior represented by each of these rules can

be derived from figure 3-2c (for the case of two inputs 'A' and 'B'). For generality, the propagation rules are given in terms of boxes which can have more than two inputs -- these N-way intersections and unions can be modeled in terms of a cascade of two-way ones if it is desired to reduce them to primitive partition constraints. The rules for union are shown in figure 3-3:

(u1) If an object is in one of the inputs, then it must be in the union.

(u2) If an object is not in the union, then it can not be in any of the inputs.

(u3) If an object is in the union and is not in all-but-one of the inputs, then it must be in the remaining input.

(u4) If an object is not in any of the inputs, then it can not be in the union.

(u5) If the union contains exactly one object, and if that object is not in all-but-one of the inputs, then the remaining input must contain exactly that object.

Note that (u1) through (u5) are isomorphic to (p1) through (p5), except that (p1) also makes use of the fact that the subclasses are disjoint. The rules for intersection are duals of those for union (as might be expected). They are shown in figure 3-4:

(i1) If an object is not in one of the inputs, then it can not be in the intersection.

(i2) If an object is in the intersection, then it must be in all the inputs.

(i3) If an object is not in the intersection and is in all-but-one of the inputs, then it can not be in the remaining input.

(i4) If an object is in all of the inputs, then it must be in the intersection.

(i5) If an input contains exactly one object, and if that object is contained in all the other inputs, then the intersection must contain exactly that object.

3.1.3 Complement

The complement of a class is defined as everything in the universe which is not in the class. Thus the way to represent it using a partition constraint is shown in figure 3-5a -- If 'T' is "everything in the universe," then 'B' is the complement of 'A' (and conversely). This behaves properly: If 'x' is in 'A', then it can not be in 'B' by rule (p1); if 'x' is not in 'A' then it must be in 'B' by rule (p3), since 'x' is in 'T'.

To constrain 'T' properly, it is necessary to state that every class is a subclass of 'T'. This can be done (rather wastefully) by having an explicit subclass constraint between each class and 'T'. It can be done less wastefully by having an explicit subclass constraint from the tops of all taxonomies (such as PHYSICAL-OBJECTS in figure 3-1) to 'T' -- all the other classes in the taxonomy are then implicitly subclasses of 'T'. However, the real problem with defining 'T' is that for a given data-base it might be difficult to decide whether or not all classes have indeed been constrained to be (explicit or implicit) subclasses of 'T'. In view of this, it seems preferable to make complement a new primitive instead of a macro. Figure 3-5b gives the network symbol for complement, and figure 3-5c shows the two obvious propagation rules:

(c1) If an object is in one of the complementary classes then it can not be in the other.

(c2) If an object is not in one of the complementary classes then it must be in the other.

Note that given intersection and complement it is possible to define all of the other Boolean functions. Thus (as promised in section 2.1) it has been demonstrated that all Boolean functions can be represented in terms of the partition constraint.

Figure 3-5d shows an example which uses all three of intersection, union, and complement. The network to the right of the dotted line represents the facts that fortunate-

ones are either wise-ones or lucky-ones (or both), that unfortunate-ones are all those that are not fortunate-ones, and that all unfortunate-ones are unhappy-ones. Suppose that these facts already exist in the data-base, and that a user wishes to know if everyone who is both unlucky and unwise is necessarily unhappy. To perform this inference, the user constructs the network fragment to the left of the dotted line: 'A1' is the class of unwise-ones, 'A2' is the class of unlucky-ones, and 'A3' is the intersection of the two (i.e. those that are both unwise and unlucky). To show that all those in 'A3' are necessarily unhappy (i.e. that 'A3' is a subclass of UNHAPPY-ONES), the initial labeling of "+x" on 'A3' and "-x" on UNHAPPY-ONES is used (as discussed in section 1.4.3). If this produces an inconsistency, then the user knows that all unwise and unlucky ones are indeed unhappy.

From the initial +x on 'A3', rule (i2) propagates +x to both 'A1' and 'A2'. From these, rule (c1) propagates -x to WISE-ONES and LUCKY-ONES. Then rule (u1) yields -x on FORTUNATE-ONES. From this, rule (c2) produces +x on UNFORTUNATE-ONES, which rule (p1) finally propagates as +x on UNHAPPY-ONES. This collides with the initial -x on UNHAPPY-ONES, indicating an inconsistency. Note that this inference in effect proves one of DeMorgan's laws (specifically, $[\neg P \cap \neg Q] \Rightarrow \neg[P \cup Q]$), which are not so easy to prove in most other logical systems.

3.2 Using the Object Constraint: Distinct Objects

One issue that was not really resolved in section 2.2 is that of distinct objects. In CE, there is no easy "syntactic" check for object equality (such as comparing print-names) -- deciding whether or not two objects are equal can involve an arbitrarily complex inference. However, in most cases it is possible to structure the data-base so that the object-equality

inferences are very simple.

The basic idea is to use a taxonomic hierarchy such as figure 3-1 to structure the objects known to the data-base. That is, every known distinct object will occur at the end of some branch of the taxonomy tree. For example, SEQUOIA-NAT'L-PARK'S-REDWOODS might be partitioned into object classes which represent the known distinct redwoods. Now, these redwoods are known to be distinct among themselves because they are all subclasses of the same partition (and hence are mutually exclusive). Furthermore, each redwood is known to be distinct from all the non-redwoods because at some level in the tree there is a partition which puts REDWOODS and the class containing the non-redwood object into distinct subclasses. For example, a given redwood and a given bacterium are known to be distinct because of the partitioning of ANIMATE-OBJECTS. In general, every object (at the end of a branch in the taxonomy tree) is known to be distinct from every other such object because at some place above them is a partition constraint which puts them into mutually exclusive classes. Thus when adding a new Sequoia National Park redwood (for example) to the data-base, it is only necessary to make it explicitly distinct from the other known Sequoia National Park redwoods -- the hierarchy of exclusive partitions above it will insure that it is distinct from every other object in the taxonomy.

3.3 Using the Binary Relationship Constraint

3.3.1 Transitive Relations

If, for example, a data-base contains the information that Aristotle is taller then Plato and that Plato is taller than Socrates, then it should be easy to infer that Aristotle is taller than Socrates. This is easy to do if everyone's height is known metrically (e.g. Plato's height is 70 inches), but it is not so easy when such complete information is not available. This section shows how transitive relations such as "taller than" can be efficiently handled within the CE representation even in the absence of complete metrical information.

The key to doing this is that even though Plato's height might not be constrained in terms of a metrical total ordering, it is constrained in terms of a partial ordering with respect to Aristotle's height and Socrates' height. Figure 3-6a shows these constraints. The "stacking" of the binary relationship constraints under HEIGHT-OF is a convenient notation for avoiding crossing lines in the network diagram -- it simply means that all three constraints have HEIGHT-OF as their relation. The arrows labeled "height greater than" are meant to represent the partial ordering -- the remainder of this section is concerned with exactly how such partial orderings can be represented in terms of the existing CE primitives.

Since CE already handles the transitive relations "subclass of" and "superclass of" in an efficient manner, the obvious thing to do is to represent "height greater than" in terms of these. That is, Aristotle's height will be known to be greater than Plato's height because some class associated with the former will be a superclass of (i.e. greater than) some class associated with the latter. Since "superclass of" is reflexive (i.e. 'A' is a superclass of 'A') and "height greater than" is not (Plato's height is not greater than Plato's height), this

section first describes how to represent the reflexive partial ordering "height as great as" (i.e. "height equal or greater than"). Then a technique for representing "height strictly greater than" is described.

To represent "height as great as," a new binary relation HEIGHT-METRIC is needed. This maps from an individual height (e.g. Plato's-height) to the appropriate "metric height." The metric height is a class such that one individual height is as great as another individual height if and only if the former's metric height is a superclass of the latter's. All this is diagrammed in figure 3-6b. ('AMH', 'PMH', and 'SMH' are the metric heights for Aristotle, Plato, and Socrates, respectively.) Now, showing that Aristotle's height is as great as Socrates' reduces to showing that SMH is a subclass of AMH, which is easy (just start a •x at SMH and apply rule (pi) twice). Of course, a well-designed user interface to a CE data-base would contain "macros" to expand constructs such as "A is as tall as B" into the appropriate CE network -- the user need not be aware of the low-level details involving metric heights.

The use of metric heights as in 3-6b does indeed behave properly, but it is also helpful to have some intuitive interpretation of what a class such as PMH "really" contains. One such interpretation is to consider PMH to contain as objects the integers from 0 to some N. This number N then can represent the individual height in some arbitrary units (such as inches). Under this interpretation, PMH would be the class of integers 0,1,...,69,70 (since Plato's height is 70 inches). Note that the data-base need never refer to these number-objects explicitly -- figure 3-6b uses the partial ordering among the classes AMH, PMH, and SMH without ever referring to the objects contained in them. It is clear that this interpretation for metric height classes satisfies the requirement that one such class be a

superclass of another iff the former represents a greater (or equal) individual height -- one sequence 0,1,...,N is a superclass of another one 0,1,...,M iff M≤N.

Given this technique for representing "as great as," it is easy to extend it to represent "strictly greater than." All that is needed is to use a strict version of "superclass." For example, AMH is a strict superclass of PMH iff AMH is a superclass of PMH and PMH is not a superclass of AMH. That is, iff AMH is a superclass of PMH and there is some object which is in AMH but not PMH. Figure 3-6d represents this: AMH is still a superclass of PMH, and AMH is known to contain at least one object ('h1') which is not contained in PMH. Similarly, PMH is a strict superclass of SMH.

Now, even though it is easy enough to state the fact that one class is a strict superclass of another, it is not so easy to infer such a fact. Such a "strict superclass" inference (involving AMH and PMH, for example) must be done in two steps. The first step is to determine that AMH is indeed a superclass (not necessarily strict) of PMH via the usual procedure (i.e. by starting a +x/inf from PMH and a -x/inf from AMH and then watching for an inconsistency). The second step is to find some object which is in AMH and is not in PMH -- such "find" inferences are discussed in appendix A.

It is important to note that this entire discussion of transitive relations has been in terms of the underlying semantics, and not in terms of the user interface. For example, the user interface to a CE data-base might allow statements such as "taller-than(Aristotle,Plato)", with the interface expanding this statement into the network of figure 3-6b. The use of such "macros" is also discussed in section 3.1.2 -- the important point is that this document is concerned with demonstrating CE's functional capabilities, not with specifying a particular syntax for the user interface.

3.3.2 N-ary Relations

Codd's Relational Data-base scheme [Codd 1970] uses N-ary relations as its primitive construct. Section 6.2.1 in part two compares Codd's system with CE in more detail -- this section briefly describes how N-ary relations can be represented in CE in terms of binary relations. An example N-ary relation is the following for persons: A person has a name, a mother, a father, a sex, and a blood-group. Thus the relation tuple for Jane Smith might be ("Jane Smith", "Mary Smith", "Joe Smith", "female", "ab") where the double-quotes indicate that the items in the tuple are character-strings.

One common way of representing N-ary relations in terms of binary ones is to use a separate binary relation for each "slot" of the N-tuple. Figure 3-7a uses this technique to represent the above N-ary relation. In this case, the binary relations used for the individual slots are NAME-OF, NAME-OF-MOTHER-OF, NAME-OF-FATHER-OF, NAME-OF-SEX-OF, and NAME-OF-BLOOD-GROUP-OF. As above, the double-quotes in 3-7a indicate character-strings: For example, the NAME-OF-MOTHER-OF Jane Smith is an object which is the character string "Mary Smith".

Since CE can represent objects (e.g. people) directly (and not just in terms of their names), it seems preferable to use a binary relation MOTHER-OF (which relates a person to the object which is their mother) instead of NAME-OF-MOTHER-OF. Figure 3-7b does this for all the binary relations in 3-7a except for NAME-OF, which is still a character string.

3.4 Using the Inverse Constraint

3.4.1 Total Relations

Section 2.4 shows how the inverse constraint can be used with the binary relationship constraint to state that every object in the domain of the relationship is indeed related to some object in the range. For example, figure 2-9a states that every person has a sex. The construct used in 2-9a occurs sufficiently often to deserve its own symbol. Figure 3-8a shows this symbol for a "total binary relationship" and 3-8b shows what the symbol means. As in section 3.1, this symbol can be either a "macro" for 3-8b, or it can be a new primitive (if implementation circumstances warrant it).

3.4.2 Quantification

Figure 2-15 shows how the class of GRANDPARENTS can be defined by using the typical-member constraint. As can be seen, this is a rather complicated definition, and the t-m constraint itself seems more complex than the other constraints discussed in this document. In view of these complexities, it is fortunate that many instances of such quantification can be accomplished simply in terms of binary relationship constraints and inverse constraints.

For example, figure 3-8c gives a much simpler definition of GRANDPARENTS in terms of PERSONS and CHILDREN-OF: A grandparent is an object which has a child which has a child which is a person. Of course, the class 'B.1' represents the relation PARENTS-OF (the inverse of CHILDREN-OF), so another prose translation of figure 3-8c is that grandparents are the parents of parents of persons. In any case, 3-8c is certainly much simpler than 2-15. Section 5.3.6 in part two discusses more about CE's quantification

mechanisms and how they relate to the corresponding mechanisms of other representations.

3.5 Using the World Constraint

3.5.1 Hierarchical Contexts

Section 2.5 mentioned that worlds can be used to represent a particular "point of view" of some user of the data-base. For example, let W-JANE-SMITH be a world-class which represents all the worlds which are consistent with what Jane Smith (a user) believes. Then figure 3-9a represents the information that Jane Smith believes that all draft evaders are criminals. That is, Jane Smith's "view" of the data-base includes this information. This becomes relevant when Jane Smith initiates some inference involving the class CRIMINALS -- from her point of view that class includes draft evaders, while from some other user's point of view it might not.

When performing such an inference, it is necessary for the information attached to W-JANE-SMITH to be somehow "enabled" for the duration of the inference in a manner which does not conflict with other user's points-of-view. To do this, a +inf label is started from W-JANE-SMITH as part of the initial labeling. This +inf can then propagate to classes such as 'W.23', which enables the appropriate information. This is indeed the desired behavior. As for what the +inf label "means", putting 'inf' in the class W-JANE-SMITH means that 'inf' is one of the worlds consistent with Jane Smith's beliefs. That is, what Jane Smith believes is considered to be true for the duration of current inference.

Now, classes such as W-JANE-SMITH can often be structured within a hierarchy. For instance, Jane Smith might be working on a project within some group in

some agency of some department of an organization. So Jane Smith has her own private view of the data-base, which is a specialization of the view taken by the project as a whole, which in turn is a specialization of the view taken by the group as a whole, etc. Such a hierarchy is shown in figure 3-9b. Note that there may be other branches in the hierarchy (forming a tree) -- the project might have users in addition to Jane who have private views, the group might have other projects, and so forth.

The hierarchy in figure 3-9b is meant to parallel the formal organization chart for Jane's department. Responsibility for changing the data at the various levels can be given to administrators at those levels. This way, policy decisions made at the agency level (for example) can be reflected as information in the agency-level view, which automatically becomes part of every view below it in the hierarchy. In particular, any lower-level view which is inconsistent with the (updated) agency-level view will cause seemingly spurious inconsistencies to appear during inferences using that lower view. In the process of tracing this down (debugging the data, as it were), the inconsistency between the lower-level and agency-level view will hopefully be found. Having found this inconsistency within the organization, it is necessary to either reformulate the agency-level policy so as to not conflict with lower levels, or to reformulate lower-level policy and information to be consistent with the view imposed from above. The point of this paragraph is that using an integrated data-base within an organization might provide a powerful management tool for helping to insure that the organization's higher-level goals are in fact compatible with what is being done at lower levels.

Another uses for hierarchies such as 3-9b is for privacy and protection. Since the only way Jane Smith's private view (for example) can be used is for a '+inf' label to be put

on the class-point W-JANE-SMITH, it follows that someone without access to the class-point is also without access to Jane Smith's view. Thus by controlling access to a limited set of class-points it is possible to control access to all the information attached to them. In this way, the security and privacy structure of the data-base can be made quite independent of the particular pieces of data to be stored within that structure.

3.5.2 Naive Probability

This section shows how a world-class hierarchy can be used to handle qualified information, such as "It is likely that Jane Smith's mother is Mary Smith," and "Most birds can fly." The basic idea is to set up an ordering such as figure 3-9c. The categories such as "very likely" represent a division of the probability continuum into discrete pieces. No attempt will be made here to assign exact numerical values to these categories, and indeed they will not be used in a manner which supports standard probabilistic calculations. For example, it turns out that the probability in this scheme of "P and Q" is the minimum of the two individual probabilities, instead of being their product (under the assumption that P and Q are independent).

In any case, it is necessary to explain exactly what the classes in 3-9c mean and how they are to influence the behavior of the inference process. Like other world-classes, the ones in 3-9c are used to enable various pieces of information. In this case, a class presumably enables information of the appropriate probability. For example, W-LIKELY might enable "Jane Smith's mother is Mary Smith," which would mean that it is likely that Jane Smith's mother is Mary Smith. Now, the classes in 3-9c represent the different levels of "reliability" which a user may demand of the data-base. That is, if the user wants

answers which are "certain," then only the information enabled by the class W-CERTAIN may be used in deriving such answers. If the user is willing to accept "almost certain" answers, then the information enabled by W-ALMOST-CERTAIN can be used in addition to the W-CERTAIN information. In general, if the user demands a particular level of reliability, then only information which is at least that reliable may be used during the inference process.

Thus the general strategy is for the user to pick a reliability level (such as W-ALMOST-CERTAIN) and start a +inf label from that class. This +inf will then enable all of the information attached to that level and will also propagate to all higher levels (such as W-CERTAIN), enabling this information too. Incidentally, this shows why the probability of "P and Q" is the minimum of the individual probabilities: "P and Q" is inferable when both P and Q are enabled, which occurs when the least probable one is enabled.

Now, some users might not want to be required to set some reliability level before initiating an inference -- instead, they might like the data-base to make the inference and then tell them after it is done what its reliability level is. This can be accomplished in several ways. The most brute-force is to first try the inference with a reliability level of "certain." If that produces no answer, then the next lower reliability can be tried. In general, the inference can be repeated using successively lower reliability levels until it produces an answer at some level (assuming that it produces an answer at all). Then this answer has the reliability of that level, which is the highest applicable one since the levels were tried sequentially, in decreasing order.

As with the t-m constraint, such sequential processing can be eliminated by using

more than one world at a time. In this case, the idea is to start a (different) world-object label at every reliability level. Then the one with highest reliability which produces an answer represents the reliability of that answer. A good way to implement such a multiple-label scheme is to use the world-tree mechanism introduced in section 2.6. In this case, the world-objects started from the classes W-CERTAIN, W-ALMOST-CERTAIN, etc., will be called 'w-certain', 'w-almost-certain', etc. Furthermore, these objects will be arranged in the world-tree in the manner of figure 3-9d. The great advantage of doing this (instead of letting the different world-objects act completely independently) is that the different worlds can interact: If for example a label with a world-tag of 'w-almost-certain' interacts with a label with a tag of 'w-likely' at some constraint, then the resulting labels will propagate with tags of 'w-likely' (the "stronger" world -- see 2.6 for a fuller discussion of such interactions). Furthermore, the resulting reliability of the entire inference can be seen immediately by looking at the world-tag of the labels involved in the collision which ends the inference -- the world-tag at the collision will be the reliability-level of the answer. Now, the details of the foregoing are not too important -- the significant aspect of it is that sequential processing ("time") can be rather easily traded for multiple worlds ("space").

Note that the probability mechanism of this section can interact with the context mechanism of the previous one. For example, Jane Smith's data-base view might include "it is almost certain that draft evaders are criminals." Figure 3-9e diagrams this -- Jane Smith's view enables the attachment of "all draft-evaders are criminals" to the "almost certain" reliability level.

Also note that information which is not explicitly enabled by any world-class (via some world constraint) is always "enabled" (because it is not relativized). This means that

information which is not attached to a reliability world-class is assumed to be maximally reliable, and that information not attached to a "view" world-class is part of all views.

3.5.3 Relativizing Other Constraints

Since the world constraint is a relativized version of the subclass constraint, it seems reasonable to ask why relativized versions of the other primitives constraints have not been presented. The reason is that the relativized subclass constraint can be used to "insulate" any of the other primitives such that the primitive is accessible only in the specified worlds. For example, figure 3-9f shows an unrelativized binary relationship constraint (which was copied from figure 2-4e). Figure 3-9g relativizes it to the world class 'W': The twin world constraints serve to "disconnect" one side of 3-9f except for worlds in 'W'. The same technique can be used with all the other primitives.

There is, however, one real difference between the world constraint and relativized versions of other constraints. The world constraint can process "-w" labels on its world-class by propagating labels which deny that the subclass constraint holds in world 'w'. For more complex constraints, however, it is <u>not</u> generally possible to find a pattern of labels which represents the denial of the constraint. This is because these more complex constraints involve a conjunction of conditions -- if the whole thing is to be denied, it is not known which or how many of the conjuncts should be denied. Even so, there is no real problem -- most applications of relativized constraints involve "+" labels (for the purpose of "enabling" the constraint in the specified worlds) and not "-" ones. For example, "-" labels play no necessary part in the hierarchies described above in 3.5.1 and 3.5.2.

3.6 Using the T-M Constraint: Templates

Figure 3-7 shows how an individual Codd-like N-tuple can be represented by N binary relationships, each one specifying one "attribute" of the individual (i.e. one slot in the tuple). Now, it is not meaningful just to assign any arbitrary object as the value of some attribute -- presumably the MOTHER-OF Jane Smith must be a female person, the SEX-OF must one of the SEXES 'male' or 'female', etc. These constraints can be specified for persons in general as shown in figure 3-10a. Many of the pieces of network in 3-10a have appeared before as isolated fragments in previous diagrams -- figure 3-10a begins to hint at the kind of rich interconnection which occurs in any non-trivial CE data-base.

Figure 3-10a is an example of a "template": It uses a t-m constraint to describe the typical person (or whatever) in terms of its attributes. Note that only one t-m constraint is needed in describing all the attributes. This is fortunate because t-m constraints are rather complex computationally and it would be burdensome to have to go through the computations separately for each attribute. Of course in a real data-base a person would have many more attributes, but one t-m constraint would still suffice for all of them.

As shown in 3-10a, the template only specifies the functionality (e.g. one-to-one) and range (e.g. female persons) of the various attributes. This could have been done (rather wastefully) by using a separate t-m constraint for each attribute, such as figure 2-13c does for MOTHER-OF. However, using only one t-m constraint as in 3-10a has the further advantage that it allows the specification of constraints between the attribute values. For example, presumably the-mother and the-father are married (ignoring complications such as illegitimacy and divorce), and furthermore they are married only to each other (ignoring polygamy). Figure 3-10b shows a network fragment which expresses

these constraints -- the class-points 'the-mother' and 'the-father' in 3-10b are meant to be the same ones as in 3-10a. Figure 3-10b also includes the general information that HUSBAND-OF and WIFE-OF are inverses, and defines SPOUSE-OF in terms of them.

In summary, a template for a class provides a general structure of constraints which must be satisfied by the attributes of each object in the class. These constraints can involve either one attribute or more than one. Structurally, a template consists of its t-m constraint, its relevant set of attributes (or "slots"), and a network of constraints which must hold among the values of these attributes.

Part Two -- Epistemology, etc.

Having examined the detailed structure of the trees (in part one), it is now time to look at the forest. Section 4 briefly discusses why representation issues are important in the first place, and divides the CE representation into four more-or-less independent "layers." Section 5 then discusses some of the representational issues appropriate to each layer. Using these issues, section 6 compares CE with other data-base and artificial-intelligence representations.

4 Generalities

4.1 Why are representational issues important?

The representations used in an information-processing system fundamentally affect the kinds of structures which can be built and the kinds of processes which can operate on these structures. This is true even if two representations are in some sense equivalent -- the differences in their basic structures will still affect their macroscopic behavior. For example, consider the difference between Roman and Arabic numerals. They are formally equivalent in that either can be used to represent any positive integer. However, it is extremely difficult to do long division (for instance) using the Roman representation, much more difficult than if the Arabic representation is used. In this case, it is relatively safe to say that Arabic numerals are a "better" representation than Roman numerals -- the procedures for manipulating the Arabic ones are easier for humans to perform.

If a difference of representation can have such a great effect within the simple domain of arithmetic, it is not hard to imagine the correspondingly greater effects such a

difference can have in more complex domains. One such domain is that of computer algorithms, and this domain has literally hundreds of different representations (i.e. the various computer languages), each of which has some partisans who acclaim it to be the "best." It is not the purpose of this document to engage in such a "language debate." However, it often is instructive to compare a new representation with existing ones in order get a better grasp of its strengths and weaknesses. It is with this in mind that section 6 compares CE with other representations.

4.2 Why is the CE representation interesting?

This document makes three claims for the CE representations. The first is that it has a great deal of **expressive power** -- it is sufficiently rich to be able to represent a wide range of information. The second claim is that CE has a firm **formal semantics** -- this allows precise statements to be made about the meaning and behavior of the various CE expressions. The nature of this semantics is discussed in section 5. The third claim is that CE has a high degree of **modularity** (in several senses of the word) -- this allows various "parts" of the representation to be discussed without worrying too much about how they will eventually fit together into "wholes." Also, this modularity is largely responsible for CE's ability to make use of parallelism (discussed below).

4.3 Modularity -- Layered Decomposition

The fact of CE's modularity permeates this entire document: Section 1 discusses label-propagation without regard to how it will be constrained; section 2 discusses the primitive constraints without regard to the macro-structures which will be built out of them;

and then section 3 discusses these macro-structures. Using this decomposition, it is possible to discuss CE in terms of four separate structural "layers." They are:

(1) The idea of label propagation in general;

(2) The particular labels described in section 1.4.1;

(3) The particular primitive constraints described in terms of these labels;

(4) The particular macro-constraints made out of these primitives.

Section 5 discusses the different representational issues which are appropriate to each of these layers. Having a layered decomposition such as this makes a complex system much easier to understand: When studying constructs at one layer, the constructs of the previous layer can be considered to be "atomic" and their fine-structure can be ignored. Indeed, Simon [1969] proposes that humans can understand complex systems only when they are layered in such a manner.

Another advantage of a layered system is that the upper layers can be modified or thrown away without disturbing the lower ones. Since creating each layer of the CE representation involved its own design decisions (some rather arbitrary), it is quite possible that someone else might want to create a similar system incorporating different design decisions. In doing this, the layers below the one to be changed can be carried over intact -- it is not necessary to re-excavate the foundation in order to repaint the roof. For the benefit of those who might be interested in making such changes, appendix D discusses alternatives to some of the design decisions which are embodied in this document.

5 Some Representational Issues

This section presents some general representational issues and discusses CE's position on them. Section 6 below below discusses other representations' positions on these same issues. Section 5 often refers to "other representations" in general without giving any details -- these details can be found in section 6.

5.1 Issues relating to label-propagation in general

5.1.1 Modularity -- Relating "local" to "global"

Section 4.3 discusses one kind of modularity -- the ability to decompose a complex system into more-or-less independent layers. This section considers a different kind of modularity, which allows complex information to be represented in terms of more-or-less independent chunks of "local" information. Now, all of the representations discussed below have this kind of modularity to some extent: They all represent their information (however complex) in terms of some set of locally-meaningful "primitives." However, the representations differ with respect to how a given chunk of local information interacts with another. It is this interaction which allows more "global" information to be built up out of the chunks of "local" information.

In CE, label propagation provides the one and only channel for such interaction. This is an extremely simple channel -- most other representations use far more complex ones. The simplicity of the channel makes it easy to describe the global meaning of a CE expression in terms of the local meanings of the primitive constraints -- one need not worry about possible complex interactions, since there are none. In general, minimizing gratuitous interaction greatly simplifies the task of analyzing and/or synthesizing complex ("global")

expressions in terms of simpler ("local") ones. Many of the benefits of "structured programming," for example, are due to exactly this kind of modularity.

5.1.2 Logical Semantics

The use of a simple interaction channel also allows the precise specification of what a representation's primitives mean in the first place. That is, since all interactions occur through a well-understood channel, the only effect a primitive (or other) expression can have is in terms of how it transmits and receives on that channel. Thus the complete meaning and behavior of a primitive can be specified in terms of its input-output interactions with the channel. On the other hand, if a representation uses a complex or ill-defined channel, then it is usually impossible to completely specify a primitive's input-output interactions. That is, it is not well-defined what the representation's primitives mean locally and how they interact.

Woods [1975] and Hayes [1974] both protest the fact that many representations do not have a precise semantics. This is not to say that representations without a logical semantics are "meaningless" -- they may work very well indeed on certain kinds of examples. However, it is usually impossible to infer from the author's examples anything about how even minor changes in some detail will affect the global structure. Especially, it is difficult to see how far the representation can be extended to cover new examples. Another problem caused by the lack of a logical semantics is that it makes it very difficult to compare two representations -- if it is unclear what one (or both) really mean by some construct, then precise comparison of the two is impossible.

Yet another problem is that lack of a formal semantics can encourage sloppy

thinking. For instance, a primitive without a precise definition may end up being used in examples where its name seems appropriate, even though no real mechanism is given for handling those examples. Woods [1975] and McDermott [1975] point out many cases of such "wishful mnemonics" and other kinds of sloppy formulation. To be sure, having a formal semantics does not make one immune from error, and for some representations the required formalization would take more effort than it would be worth. However, having a logical semantics is definitely an asset when trying to use, study, or extend a representation.

5.1.3 Procedural Semantics

A logical semantics tells what a given expression "means"; a procedural semantics tells how the expression "behaves." Presumably, a representation exists for the purpose of its being used, so one is ultimately interested in how it behaves. In a representation such as CE which is based on label propagations, the logical semantics and the procedural semantics are tightly coupled: An expression's meaning is defined in terms of how it interacts with various kinds of labels, and its behavior is determined by these interactions.

One advantage of having such a tight coupling is that it makes a representation easier to use. Instead of having to keep in mind both the meaning of the "data" (expressed in terms of the representation) and the behavior of some external procedure which accesses it (presumably expressed in some programming language), the data itself specifies both its meaning and the behavior of the accessing procedure. Of course the data might not be directly executable by some given piece of hardware, in which case a simulator is needed. Appendix B discusses three different possible implementations of CE -- one made of parallel hardware which executes the primitives directly, and two which simulate this.

Another advantage of the tight coupling is that it makes a system easier to debug. Since the accessing procedure operates in a manner which parallels the semantics of the data, the state of a procedure (in case of a crash, for instance) will be in semantically meaningful terms. In the case of CE, the inference procedure's state can be defined as the states of all the class-points (in terms of what labels are on them). Thus if there is a crash, the inference procedure's state will be easy to express in a semantically meaningful manner (e.g. "the object 'x' is known to be a person"), and things can be debugged on this level. Furthermore, it is the case that labels are never erased during an inference -- new ones may be put on a class-point, but none may be removed. Thus not only the current state but the complete history of previous states is available during debugging. Note that "error analysis" can be viewed as a high-level form of such debugging, wherein the "bug" is some kind of inconsistency caused by bad data. In this case, the history of the relevant inference can be used to determine the path of data-accesses that led to the inconsistency, which is necessary (although certainly not sufficient) for determining exactly where along that path the erroneous datum lies.

5.2 Issues relating to CE's particular labels

The remarks in section 5.1 apply to any system which operates via label propagation. The remarks in this section apply only to systems which use labels similar to the "+obj", "-obj", and "±obj" labels of CE. Appendix D discusses a similar labeling scheme using slightly different labels -- most of the remarks in this section apply to that scheme also.

5.2.1 Fregean Systems

CE (and all the other representations discussed in this document) are "Fregean" in that their universe consists of discrete objects and relationships among objects. In CE's case, this is a consequence of the fact that its labels are defined as relating discrete objects to classes of objects. The reason for mentioning all this here is that there seem to be certain limitations on what Fregean systems can represent. Hayes [1974] discusses this in more detail. One of his examples is "substance": A substance (such as water) is usually not thought of as being composed of discrete objects. A pail (or drop, or ocean) of water seems to be a single distinguishable object, but what of the water itself? It is the existence of such issues which indicate that the final answers to the representation problem are by no means available (especially for sophisticated Artificial Intelligence applications). However, the state of the art is such that it does seem profitable to apply AI technology (such as CE) to the problems of data-bases.

5.2.2 Consistency Checking

CE's labels are designed to facilitate consistency checking -- an inconsistency is detected iff a "-obj" label collides with a "+obj" (or "-obj") at some class-point. Section 1.4 shows that redundancy checking and the answering of yes/no questions can be subsumed under consistency checking; appendix A shows how this can be extended to "find" questions. An important real-world data-base application of such consistency checking is to validate incoming data in terms of what is already known. If an inconsistency is detected, then something is wrong and the appropriate actions should be taken (such as rejecting the bad data, rejecting it and logging a record of the inconsistency, asking a human to correct it, or attempting some form of automatic error analysis and correction).

5.2.3 Additions / Deletions / Updates

Using CE, checking the consistency of new data with respect to the existing data-base is the way to validate it before adding it to the data-base. Some representations which do not have a consistency-checking procedure do, however, have other means for handling the addition of new data. Similarly, some representations provide special means for handling deletions of existing data. An update, of course, can be considered to be a deletion followed by an addition, so those representations which handle additions and deletions can also handle updates. In addition, some representations have a separate (more efficient) means for update processing.

CE does not provide any additional mechanisms for handling deletions (and hence updates). In general, deleting a piece of network from a CE data-base can not possibly cause the remaining data to become inconsistent: Inconsistency is a state resulting

when "too much" is known (in particular, when both an assertion and its negation occur), so deleting some data from an existing data-base (which, anthropomorphically, makes the data-base know less) is a safe operation. Note that deleting the datum "α" means that the data-base after the deletion does not know "α" -- this does not mean that the data-base knows "not α."

The one problem which can arise during a deletion is that some data might be stored redundantly. For example, the data-base might contain both "Jane-Smith is in the class FEMALES" and "the SEX-OF Jane-Smith is 'female'" (along with the general information that FEMALES is exactly the class of objects which has a sex of 'female'). In this case, deleting "Jane Smith is in the class FEMALES" will not cause the data-base to forget the fact that Jane Smith is female -- the undeleted redundant data will still imply it. A degenerate case of this is when the same datum is represented twice -- deleting one of the instances of it obviously has no effect on the other. So, what is needed is some manner of telling whether a deleted datum is still implied by the data-base. This is simply ordinary redundancy checking: Temporarily add the negation of the deleted datum and see if there is an inconsistency. If there is, then the data-base without the deleted datum still implies it. Having detected this anomaly, one is in the same situation that occurs when an inconsistency is detected during the addition of data -- some clever program (or person) must be called to determine how to resolve things.

5.3 Issues relating to CE's primitives

5.3.1 Logical Consistency

The discussion of "consistency checking" throughout this document has been based on the supposition that whenever a representation's inference procedure signals an inconsistency, then there is in fact inconsistent data in the data-base (such as having both "Jane Smith is in the class FEMALES" and "the SEX-OF Jane Smith is 'male'"). However, it might be the case that the inference procedure occasionally signals inconsistencies when there are in fact none. If the inference procedure can not be relied on, then the task of consistency checking is made that much more difficult -- in such cases, it is not clear whether a signaled inconsistency is really due to inconsistent data or is just an artifact of the inference procedure. Thus it is useful to be able to show that an inference procedure is "logically consistent" -- that it never signals spurious inconsistencies.

For representations with complex ad hoc inference procedures, it is very difficult (if not impossible) to show that they are logically consistent. For CE, it is easy to demonstrate logical consistency because the behavior of the inference procedure is tightly coupled to the meanings of the CE primitives. Consider: All the inference procedure does is put labels on classes. These labels represent assertions (by the inference procedure) about which extensional object are (and are not) in the various classes. Now, a spurious inconsistency could be caused only by putting a "wrong" label on a class (e.g. putting a +x label on a class C when the data-base does not anywhere imply that 'x' is in 'C'). However, section 2 shows that the inference procedure only propagates the "right" labels: For each primitive in section 2, its label-propagating behavior is directly derived from its meaning. Thus there is no place where a spurious inconsistency can be introduced.

Note that a demonstration of logical consistency requires three things: A logical semantics which describes what the representation means; a procedural semantics which describes how the inference procedure behaves; and some form of connection between the two to show that the behavior is in fact compatible with the meaning. Thus it is impossible to demonstrate the logical consistency of a representation which does not have both a logical and a procedural semantics.

5.3.2 Logical Completeness

A representation can be said to be logically complete iff every possible inconsistency in the data can be found by the inference procedure (given enough time). It turns out that CE is not complete -- a simple example is shown in figure 5-1. Here, it is clear that 'A' and 'C' contain the same objects (since both 'A' and 'C' must contain exactly the same objects as 'B'). However, it is not possible to derive an inconsistency starting with the labeling that some object 'x' is in 'A' but is not in 'C' -- none of the propagation rules can be applied.

Now, all known complete inference procedures for sufficiently rich representations (e.g. those containing at least the Boolean connectives) end up taking time proportional to an exponential function of the size of the data-base being used. Indeed, Karp [1972] presents mathematical evidence to the effect that any complete inference procedure for any such representation must take exponential time. Thus for a moderately large data-base, having a complete inference procedure is of absolutely no benefit unless one is prepared to wait a very very long time while it runs.

5.3.3 Practical Completeness

Since logical completeness is so impractical, the best that can be hoped for is that a data representation be reasonably complete and efficient with respect to the kinds of structures that are most commonly encountered in the data-base. Lacking any large-scale empirical evidence as to how CE and other representations perform in practice, the issue of practical completeness can not be resolved; thus no mention is made of it in section 6. However, it is reasonable to say that CE processes certain "simple" constructs (for humans) such as "all A are B" in a computationally simple manner, and that the degree of computational complexity involved in processing a CE expression correlates reasonably well with the expression's intuitive complexity. Whether or not this is of any importance remains to be seen.

5.3.4 "General" vs. "Specific" Information

A significant feature of CE is that the same set of primitives is used to represent both "general" and "specific" information. An example of general information is: "Every person has a unique sex, which is either female or male" (see figure 3-10a). An example of specific information is: "Jane Smith's sex is female" (figure 2-4a). Now, since the CE data representation and inference procedure make no built-in distinctions between "general" and "specific," the single inference procedure can detect all of the following kinds of inconsistencies:

(1) Specific vs. General (or vice versa): Some piece of specific data is inconsistent with the general information. For example, the specific information that "Jane Smith's sex is Mary Smith" is inconsistent with the general information that "every person's sex is either male or

female." (Presumably the general information contains a taxonomy such as figure 3-1 which has SEXES be mutually exclusive with PHYSICAL-OBJECTS. Thus it is known that the object 'Mary-Smith' is not one of the objects 'male' and 'female'.)

(2) Specific vs. specific: Two pieces of specific data are mutually inconsistent. For example, "Jane Smith's sex is female" and "Jane Smith's sex is male." Each is consistent with the general information, but they are inconsistent with each other.

(3) General vs. general: Two aspects of the general information are mutually inconsistent. For example, the general information might contain "all draft evaders are criminals" along with "some draft evaders are heroes" and "no heroes are criminals." Such an inconsistency could arise (for instance) if the general information came from different sources.

It is significant that the same procedure which detects inconsistencies involving specific information can also detect inconsistencies in the general information. For a given data consistency-checking application, this feature makes it significantly easier to "debug" the general information which is to be used to check the incoming specific information.

5.3.5 Incomplete Information

In addition to being useful for consistency checking, using the same primitives for general and specific information facilitates the representing of incomplete information. For example, one might have the information that "all draft evaders are criminals" without having an exhaustive list of all the draft evaders. In some representations, the only way to state that "all draft evaders are criminals" is to take such a list of all draft evaders and to state individually for each one that he is a criminal. Also, the only way to answer the query "Are all draft evaders criminals?" is to examine every individual draft evader in the data-

base and check if he is a criminal. It so happens that of all the representations listed in section 6, the only ones that can both make statements and answer queries about incomplete information are those which use the same primitives for general and specific.

5.3.6 Quantification -- Explicit, Implicit, and Sloppy

For those representations which do represent both general and specific information in a similar manner, some means is needed for distinguishing the two. For example, "(HAS PERSON SEX)" might mean (in some hypothetical representation) that some person has a sex, or that every person has a sex, or that all persons have the same sex, etc., etc. There seem to be three different techniques for handling the distinction between "general" and "specific."

The first is to explicitly differentiate the two by associating different quantifiers with each (or by associating a quantifier with one and leaving the other as unmarked). In CE, explicit "general" quantification of objects is provided via the typical-member constraint (with all unquantified objects being "specific"). In contrast to this, mathematical logic provides explicit quantifiers for both "general" and "specific" (i.e. "$\forall$" and "$\exists$", respectively).

The second technique is to use primitives which involve implicit quantification. In CE, most primitives are defined in such a manner that they can be applied to general classes as well as specific objects. In some sense, CE only deals with general classes -- a specific individual object is represented as a class which happens to be constrained to contain exactly one object. To see the quantification implicit in the primitives, consider figure 2-8a, which states that PARENTS-OF and CHILDREN-OF are inverses of each

other using one primitive constraint (and no explicit quantification). In mathematical logic (which makes all quantification explicit), the same thing would be written as: VxVy[PARENTS-OF(x,y) ≡ CHILDREN-OF(y,x)].

The third technique for handling the distinction between "general" and "specific" is to ignore it. This can lead to ambiguities such as the above "(HAS PERSON SEX)." This technique will be called "sloppy quantification." Note that only those representations which do not have a logical semantics fall prey to sloppy quantification -- having a semantics prevents one from ambiguously using such words as "has." Woods [1975] and Hayes [1974] point out several kinds of sloppy quantification. Of course, those representations which do not allow both general and specific information have no need for any sort of quantification in the first place.

5.3.7 Worlds and States

One major difference between CE and other representations lies in CE's use of the "world object" and "world class" constructs. A world-object represents a (partially-specified) state of the universe, and a world-class represents a collection of these. A significant feature of CE is that it treats worlds as entities which can be manipulated in the same manner as simpler objects. That is, world-objects may be quantified, may participate in binary relationships, and may in general be used in all the ways that other objects can. Thus it is possible to reason <u>about</u> worlds in addition to reasoning "within" them. Appendix C shows how the uniformity of this approach makes it relatively easy to reason using "knowledge about knowledge" (such as "Billy knows who Jane Smith's real father is, and she doesn't know that he knows.") This may someday have applications for intelligence

data-bases.

5.4 Issues relating to CE's non-primitive expressions

5.4.1 Representational Completeness

The sections on "logical completeness" and "practical completeness" discuss the completeness of the inference procedure (in terms of how thorough it is in finding inconsistencies). This section discusses a different sort of completeness: A representation is "representationally complete" for a given application if all of the data required for the application can be encoded as structures in the representation. This, like practical completeness, is difficult to judge in the absence of a significant amount of empirical evidence, and in any case it is relative to the particular application. One purpose of section 3 is to show that various useful macro-structures can indeed be built out of the CE primitives. To recapitulate, the structures mentioned in section 3 are: taxonomies, Boolean connectives, distinct objects, transitive relations, N-ary relations, total relations, inverse relations, hierarchical contexts, naive probability, and templates. It would take too much space in section 6 to comment on how adequately each of the mentioned representations handles all of these constructs -- only a few will be mentioned for each.

5.4.2 Procedural Attachment

The constructs from section 3 which are listed above are ones which CE handles fairly well. In addition, there are other constructs which CE does not currently handle but which are important in some of the other representations discussed in section 6. One of these is "procedural attachment": This allows executable procedures to be attached to

various pieces of data in such a manner that accessing the data causes the appropriate procedures to be invoked.

For example, some systems handle additions to the data-base by running procedures which are appropriate to the specific datum being added, and similarly for deletions and updates. This will be called "antecedent" processing, in that having the data (to be added or whatever) triggers the procedure. Another example is that procedures can be used to derive certain kinds of data during an inference -- the appropriate data is computed by some procedure (using perhaps other data in the data-base), instead of actually having the data be explicitly present. This will be called "consequent" processing, in that needing the data triggers the procedure.

5.4.3 Events

Another important representational construct which CE does not currently handle is the notion of "events". Simply put, an event corresponds to some change in the world (which might have to be reflected as a change to the data-base). Now, CE does have provisions for accomodating changes in the data-base (see section 5.2.3 on additions, deletions, and updates). However, CE currently has no explicit representation for the meaning of an event. For example, a representation of the event "getting married" should presumably say something about what must be true before the event can take place (e.g. in the USA the beings getting married must be of different sexes, be of marriageable age, and not already be married). In addition, an event's representation should specify what changes as a consequence of the event -- for "getting married," it is presumably necessary to change the beings' marital status and to indicate that they are now spouses.

One way of representing all this is to represent an event as the difference between a "before" state and an "after" state. So the before-state of "getting married" would specify the above preconditions (different sexes, etc.), and the after-state would specify the postconditions (marital status is 'married', etc.) Let this be called the "static" approach for representing events, in that the event (which consists of two states and a transition between them) is specified in terms of the two static states, and the nature of the transition is derived from this.

Another approach will be called the "dynamic" approach: In this one, the before-state and the transition are specified, and the after-state just follows as a consequence of "doing" the transition to the before-state. The standard way of doing this is to have the transition be some procedure which is executed to transform the before-state into the after state. Since the static approach can also be said to specify a procedure (implicitly, in terms its effects on the before-state), the defining characteristic of the dynamic approach will be considered to be that the procedure which specifies the state-transition is a black box: The structure of such a procedure is unimportant, since we are only interested in the effects it has in terms of transforming the before-state.

This is hardly the place to enter a discussion of the philosophical nature of events and the "best" way to represent them. However, it is reasonable to include some discussion of the technical advantages and disadvantages of the static and dynamic approaches. A major disadvantage of the static approach has been termed the "frame problem" [McCarthy & Hayes 1969]: It is not sufficient to just specify the differences between the before and after states -- it is also necessary to somehow specify that nothing else changes (unless perhaps it is a necessary consequence of the specified changes). For

example, getting married presumably does not change a person's sex (or parents, or blood group -- the list of what does not change is clearly too huge to explicitly enumerate).

The dynamic approach does not have this difficulty since the transition procedure presumably knows exactly what aspects of the before-state to change, and anything it does not touch is ipso facto unchanged. However, the black-box nature of the procedure makes it much more difficult to reason about events (as opposed to just performing them). For example, using the dynamic approach it is impossible to decide if a given after-state could have resulted from a given event -- it is not possible to run the event's black-box procedure "backwards" in an attempt to derive a before-state which could have produced the given after-state.

Given that the two approaches are good for two different things, an obvious solution is to have both. The problem with this is that it is not generally possible (currently) to show that a given transition procedure correctly implements a given static description. That is, it is quite possible that the dynamic description and the static description of purportedly the same event are not in fact equivalent. To insure this equivalence, one either needs a powerful procedure-analysis technique (to see if the dynamic description does indeed satisfy the static one) or an equally powerful procedure-synthesis technique (to derive the dynamic description given the static one). Both of these are quite beyond the current state of the art.

5.4.4 Arithmetic

One concept which the current formulation of CE has a great deal of trouble with is that of "number." It is possible (but quite unwieldy) to express numbers in terms of

cardinalities of classes, such as is done in certain axiomatic formulations of arithmetic. It is also possible to formalize arithmetic using the notion of "successor" (as is done in Peano's axioms). Either of these formal approaches has the disadvantage of being quite unnatural for most humans (and most existing data-bases).

In addition to the "formal" approach to arithmetic, there is the "procedural" approach. Many representations handle arithmetic by using specialized procedures for the arithmetic operations. Due to the general opacity of "black box" procedures, these representations find it difficult (if not impossible) to reason about arithmetic. But of course such systems do not have to -- for their applications they only need to reason using arithmetic.

6 Some Representations

The section discusses several representations in terms of the issues presented in section 5. No attempt is made to explain the different representations in great detail -- anyone desiring such details should consult the bibliography. The representations considered here are chosen from those concerned with data-bases (DBTG, Codd's relational model), mathematical logic (first-order predicate calculus), cognitive simulation (Quillian's semantic memory), and artificial intelligence (Planner-like languages, semantic networks). This is a reasonably representative sample, but it ignores some of the newer ideas, especially those which currently lack sufficiently concrete documentation (e.g. MERLIN [Moore & Newell 1974], and "frames" [Winograd 1975]).

6.1 An Aside: "Assertions" vs. "Networks"

Before proceeding with individual discussions of each of the above representations, it is instructive to group them into two broad classes: those which represent data in terms of "assertions" and those which use "networks." Syntactically, the difference between the two is obvious: Network representations (such as CE) encode their data in some kind of graphical network, while assertional representations prefer a linear notation. For example, figure 2-4a is a network representation for "the sex of Jane Smith is female." A corresponding assertional representation might be "(SEX-OF Jane-Smith female)". In general, tokens which appear in assertional notations correspond to points (often called "nodes") in network ones. In addition, expressions nested within assertions can correspond to network points. For example, an alternative assertional representation for figure 2-4a is "female = (SEX-OF Jane-Smith)": Here, SEX-OF is a <u>function</u> and the result of the

function is represented by the whole expression "(SEX-OF Jane-Smith)".

Given this kind of rather direct syntactic correspondence between assertional and network notations, it is perhaps tempting to say that the only difference between them is the syntax used, and that there is no reason other than personal taste for preferring one to the other. Indeed, when "network" information is entered into a computer the network is usually first encoded into some linear notation which the computer can easily read. For example, various parts of CE have been implemented in LISP, which requires that everything be encoded as parenthesized expressions. Then again, a common notation for LISP itself involves drawing the parenthesized expressions as networks! -- see figure 6-1. Thus it is clear that any network can be recoded as a set of assertions, and vice-versa.

However, there is more than a syntactic difference between networks and assertions when it comes to processing them. Network notations use visual connectivity to emphasize the local connections between things, and thus are good for representations such as CE which operate on the basis of tracing through such connections. (In the case of CE, the local connections provide the paths along which labels propagate, and there is no other kind of processing). Assertional notations, on the other hand, emphasize the syntactic patterns of the expressions -- they are usually processed via some kind of pattern matching which compares two whole expressions at one time (instead of having to do it in terms of local connections). Of course, at some level the pattern matcher must use "local connections" (e.g. the fact that two tokens are equal), but the user is not concerned with this level of detail.

Thus in the discussion which follows, systems will be called "network-based" if they process information in terms of local connections, and "assertion-based" if they use the

rather more "global" connections provided by pattern-matching.

6.2 Assertion-based systems

6.2.1 Codd's Relational Data Model [Codd 1970]

The primitive construct in Codd's representation is the flat N-tuple, and a relation is a class of such N-tuples (just as in CE a binary relation is extensionally a class of 2-tuples). The "slots" in each tuple contain atomic values (such as character strings or numbers) -- they do not point to other tuples. Tuples are accessed via pattern matching -- the standard accessing operation is to create a new relation consisting of all tuples in an existing relation (or cross-product of relations) which match a given pattern. The user interface to a relational data-base consists of a high-level query language, which gets compiled (or interpreted) into a series of pattern-match requests.

The logical semantics for this system is the "relational algebra," which describes how relations may be meaningfully subsetted, projected, etc. The procedural semantics is embodied in the pattern-matcher which implements these operations. Thus there is a reasonably close coupling between the logical and procedural semantics.

As for consistency checking, this an area of current research. Much of this research is devoted to developing additional representations which can be used alongside the tuples. One reason that some other representation is needed is that the tuples themselves deal only with "value" objects such a numbers and strings -- there is no direct way to refer to real-word objects (such as persons).

When performing additions, deletions, or updates, it is necessary to do special processing to insure that the assumptions of the relational algebra are not violated. For

example, during additions it is necessary to check that the tuple to be added does not duplicate one that is already there. During deletions, it is necessary to also delete those tuples which "depend on" the one being deleted. (The notion of "depend on" actually turns out to be quite complex, and is a topic of current research interest.)

It is easy to show that Codd's scheme is logically consistent and complete. It is consistent because the procedural semantics (as implemented in the pattern matcher) is a direct reflection of the logical semantics (the relational algebra). Of course, it may be very difficult to show that a given implementation of a pattern matcher correctly implements the relational algebra (especially when complex optimization is done), but the general idea is simple. As for completeness, since the relational data-base's universe is finite (consisting of a finite set of relations, each being a finite class of finite-length tuples), any exhaustive enumeration procedure will be a complete one.

Now, the major limitation of Codd's scheme is that it has absolutely no facilities for expressing general information. Since the rest of the issues discussed in section 5 depend in some way on the use of general information, it is pointless to discuss them in the context of this representation. With regard to the issue of consistency checking mentioned above, another reason for needing a separate representation for expressing consistency constraints is that they usually involve general information. For example, "all persons have a unique sex, one of 'male' or 'female'" is one such general constraint. In summary, the ability to handle general information is the major functional difference between CE and Codd's scheme.

6.2.2 Planner-like Languages

The Planner-like languages are the result of one approach for adding general information to a Codd-like data-base. These languages were developed for artificial intelligence applications, and include Planner [Hewitt 1972], Conniver [McDermott & Sussman 1973], GOL [Pople 1972], and QA4 [Rulifson et al 1972]. For purposes of this brief discussion, no distinction will be made among them (even though significant differences do exist) -- the discussion is in terms of the general approach, not in terms of some particular incarnation of this approach.

There are two components of these representations. The first is an assertional data-base which is essentially like Codd's. The differences are minor: In Codd's scheme, the tuples are "in" the appropriate relation, while in assertional data-bases the appropriate relation is "in" each tuple (by having the first slot in the tuple be the relation's name). Also, assertional data-base tuples may be nested, such as
"(COLOR-OF BLOCK1 (DARK RED))".

The second component of a Planner-like representation handles the general information. This is done using procedural attachment as discussed in section 5.4.2. The procedures are attached to patterns, such as "(COLOR-OF ?X ?Y)". When such a pattern is successfully matched against an assertion in the data-base, the pattern's variables (here, X and Y) get bound to the appropriate pieces of the assertion (e.g. X = BLOCK1 and Y = (DARK RED)). This binding process is the way a procedure receives its arguments -- the procedure has access to the bindings of its pattern's variables.

Both "antecedent" and "consequent" processing are done using attached procedures. For antecedent processing, there is one set of procedures for additions, and a

separate set for deletions. When a datum is about to be added to the assertional data-base, all "addition" procedures attached to patterns which match the datum are executed. These procedures may in turn access the data-base, possibly causing other prccedures to be run. Similarly, appropriate "deletion" procedures are executed when a datum is deleted. For consequent processing, there is another set of procedures for generating assertions which match a given pattern. For example, a consequent procedure attached to the pattern "(PRIME ?N)" might generate the prime numbers (it of course being infeasible to store them all directly as assertions of the form (PRIME 2), (PRIME 3), (PRIME 5), etc.) In more complex cases, the generating procedures can themselves access the data-base, possibly invoking other procedures.

Now, since all processing within a system based on Planner-like languages is controlled by the attached procedures, the "procedural semantics" (i.e. behavior) of the system is determined by the user who codes these procedures. Thus little can be said about a Planner-like system's behavior "in general," because little can be said "in general" about the behavior of any programming language. As a consequence of this, there is no built-in logical semantics for the meanings of the assertions. For example, the assertion "(NOT (COLOR-OF BLOCK1 GREEN))" might mean that block1 is not green, if the relevant procedures have been coded to treat "NOT" according to its customary meaning. Thus it can be very difficult to determine the global meaning of a given assertion, since it depends on the whole structure of procedures installed in the system (which may be very complex).

Since there is no general logical semantics, there can be no general way of doing consistency checking. Also, notions of "logical consistency" and "logical completeness" are

inapplicable without a logical semantics. Consistency checking can be implemented for a particular application by having the "addition" procedures do whatever checking is necessary before a datum is added, but this requires that all the information about what to check be coded directly in the procedures. Thus when new checks are needed it is necessary to change all the relevant procedures, which can be very difficult. Also, it is impossible to do "general vs. general" consistency checking, since the general information is implicit in the structure of the procedures and is not directly manipulable.

Some Planner-like languages (e.g. Micro-Planner [Sussman et al 1970]) handle universal quantification by explicitly iterating through the set of relevant pattern-variable bindings. For example, the notion of "every dark red object" is represented in a procedure as a loop which iterates through all of the bindings of X for assertions which match "(COLOR-OF ?X (DARK RED))". This of course means that the class being quantified over must be reasonably small -- "every person" would take too long, and "every prime number" would take infinite time. As a concrete example, the query "Are all draft evaders criminals?" is answered by enumerating all of the known draft evaders and then checking each one for criminality. Not only will this take quite a while if there are many draft evaders, but it requires complete information concerning exactly who all the draft evaders are. Robert Moore [1975] is currently researching the problem of handling incomplete information within a Planner-like system.

The languages QA4 and Conniver do provide a mechanism for handling multiple worlds. Each world-class (called a "context") is implemented as a list of "layers." Each layer describes the differences between itself and the context represented by the following layers in the list. This implementation makes it easy to create a hierarchy of

contexts without unnecessary copying -- only those assertions which are different need be recorded. Using these contexts and the appropriate procedures, it is easy to represent events dynamically -- an event procedure takes a before-context and returns an after-context which is the before-context with an additional layer representing the changes due to the event.

However, unlike CE world-classes, contexts are not themselves manipulable. That is, it is not possible to reason about contexts. For example, it is usually not possible to determine whether one context is a stronger version of another one, while in CE this is easily done (by showing that the first world-class is a subclass of the second).

With respect to the topics listed under "representational completeness" in section 5, the only ones which Planner-like languages handle well are N-ary relations and exceptions (such as "All birds can fly, except a few such as penguins and ostriches," which CE can handle using probability). N-ary relations (i.e. assertions) are primitives in the system. Exceptions can be handled using the context mechanism: Creating a new context from an old one by adding an additional layer implies that the new one is exactly like the old one except where explicit differences are noted in the new layer. Note that the notion of "subcontext" (i.e. a context grown from another one by adding a layer) is quite different from the CE notion of one world-class being a subclass of another. In CE, the subclass may be different from the superclass by being stronger (i.e. by "knowing more"), but it must be consistent with the superclass (i.e. it may not "know different"). On the other hand, in QA4 and Conniver a subcontext may be arbitrarily different from its supercontext.

In summary, the major functional differences between CE and Planner-like languages are (1) that CE facilitates consistency checking, and (2) that Planner-like languages facilitate procedural attachment.

6.2.3 First-order Logic and Resolution

In the format usually used by humans, the primitives of first-order logic include variables, constants, Boolean connectives, N-ary functions (e.g. SUM-OF(x,y)), N-ary predicates (e.g. GREATER-THAN(x,y)), and explicit quantifiers (∀ and ∃). "Resolution" [Robinson 1965] is the machine-oriented inference procedure commonly used with first-order logic. It requires that expressions be converted to "Skolem conjunctive normal form," which basically involves transforming them to remove the constants, the Boolean connectives, and the explicit quantifiers. Given a set of expressions in this format, the resolution procedure uses "unification" (a pattern-matcher) to combine two existing expressions and thus generate a new one. This new expression is then added to the set of expressions, and the unification cycle repeats. Usually, the cycle is repeated until an inconsistent expression is generated -- as with CE, this implies that the original set of expressions was inconsistent.

Thus resolution (like CE) is oriented towards consistency checking. This requires first-order logic to have a logical semantics, and requires resolution to have a corresponding procedural semantics -- there are in fact formal arguments which demonstrate that both these conditions do hold. Furthermore, resolution is known to be both logically consistent and logically complete.

Like Planner, first-order logic does handle general information. Unlike Planner, the general information is expressed in the same manner as the specific information. Furthermore, first-order logic can handle incomplete information -- it is not necessary to have information about all the members of a class in order to make inferences concerning that class.

So far, first-order logic and CE seem quite similar -- it is now time to look at the

differences. For one, first-order logic has certain formal difficulties with expressing the notion of equality (i.e. identical objects). These difficulties lead to various attempts to extend resolution to handle object identity in a more natural manner. In CE, equality is simply a degenerate case of subclass, which is a primitive. Another difference is that first-order logic can handle arithmetic (and indeed most of mathematics) -- it uses the "formal" approach discussed in section 5.4.4.

However, the major functional difference between first-order logic and CE lies in CE's use of worlds -- first-order logic has no analogous construct. This makes it very difficult (if not impossible) for first-order logic to handle hierarchical contexts, knowledge about knowledge, etc. Some AI research has been done on the issue of adding worlds to first-order logic, notably by McCarthy (e.g. McCarthy & Hayes [1969]) -- this is still a wide-open area.

6.3 Network-based systems

6.3.1 DBTG and COBOL

The local connections in a DBTG network [Codasyl 1971] are the access paths along which a COBOL program may trace in order to get access to the various records in the data-base. Thus the interaction channel (i.e. the manner in which the "local" data structures interact to make more "global" ones) consists of the particular COBOL procedures which access the network.

As discussed above with respect to Planner-like languages, using arbitrary procedures as part of the interaction channel means that there can be no general logical semantics for the representation. That is, what a particular piece of data-structure "means"

is totally dependent on the detailed behavior of the procedures which access that data. As above, if a representation lacks a logical semantics then it is impossible to have a general consistency-checking procedure for it (since such a procedure obviously needs to know what the data-structures mean in order to tell if they are consistent). In DBTG, almost all consistency checking (done at the time of an addition, deletion, or update) must be explicitly coded into the particular programs which do the additions, etc. Unlike Planner-like languages, DBTG has little provision for procedural attachment. This means that the COBOL programs tend to become rather monolithic as new features are added to a DBTG system -- there is no way to modularly attach new procedures to the relevant data (as opposed to combining them all in one monolithic program).

There are of course further differences between DBTG and other representations (such as Codd's), but these are irrelevant to comparing DBTG with CE. The major difference between DBTG and CE is that virtually all of the interesting information in a DBTG system ("general" information, information about what the data-structures mean, etc.) is buried deep within the particular COBOL procedures instead of being more directly accessible (for purposes of consistency checking, changes, etc.) It is of course true that DBTG can do anything that CE (or any other data-base scheme) can do, but only because COBOL is a Turing-universal programming language -- DBTG does very little towards making such universal power more tractable to use.

6.3.2 Quillian's Semantic Memory

One of the first network-based representations was Quillian's [1967] model for human associative memory. The "semantic memory" consists of a set of nodes representing

"concepts" (such as "dog", "meat", "eat", etc.) connected by links representing "associations" (e.g. there might be links connecting "eat" to both "dog" and "meat," presumably indicating that dogs eat meat). The memory is accessed by taking two concepts and finding the shortest path of associations connecting them. Thus given "dog" and "meat," the shortest path might be the one through "eat."

This is very much in the spirit of psychological word-association tests, and is not at all meant to be a model of more-structured "logical" thinking. For example, the above example could just as well mean that meats eat dogs -- the links have no meaning other than that of pure association. Now, being a psychological model, Quillian's system finds the shortest path in a psychologically plausible manner. The system propagates markers along the links breadth first (in parallel), starting at the two given concepts ("dog" and "meat"). Thus the place where these two "wave fronts" intersect is guaranteed to lie along the shortest path between the two given concepts. The psychological plausibility of this lies in the fact that it can be accomplished by neuron-like cells working in parallel.

Clearly Quillian's scheme is too unstructured to be useful in a data-base. It is included here because it exemplifies label propagation and some other aspects of CE. For one, it does have a kind of logical semantics -- the notion of "shortest path" can be rigorously defined in terms of graph theory. The procedural semantics is straightforward (parallel marker propagation), and the connection between the logical and procedural semantics lies in showing that parallel propagation indeed results in finding the shortest path.

Unlike CE, the critical aspect of Quillian's scheme is the <u>timing</u> of the propagations -- if they are not done strictly breadth first then the first connecting path

found might not be the shortest. In CE, the order in which things are done is irrelevant -- a different order may cause a label collision to occur at a different point in a CE net, but it is only the occurrence which matters, not the particular location.

6.3.3 Semantic Networks

Having introduced the idea of an explicit network of links and nodes to encode meaning, Quillian and others tried to apply such networks to harder problems. Quillian's [1969] TLC system was an attempt to do natural-language comprehension using an extended version of his semantic memory. TLC's network consists of different kinds of links, with different rules for propagating markers along them. This prevents confusions such as the above "meats eat dogs."

However, all existing semantic memory schemes (e.g. TORUS [Mylopoulos et al 1975], OWL [Martin 1974], and Fahlman's [1975]) lack a well-defined logical semantics. This leads to confusions such as the "sloppy quantification" discussed in section 5. The "procedural semantics" for a semantic network system tends to be embodied in some complicated procedure for traversing the network, rather like DBTG. Quillian's original idea of well-defined parallel marker propagation seems to have been rejected as being too tied up with a very naive view of neurophysiology, having no bearing on how things should be represented in a computer. From one viewpoint, CE attempts to show that parallel propagation is interesting computationally as well as psychologically.

7 Some History

7.1 CE's Past

As can be seen from section 6, research on CE has been influenced by work on mathematical logic and by work on semantic networks. From mathematical logic comes the emphasis on having a well-defined semantics for all constructs. Also, a CE "world" is quite similar to a logical "model," the major difference being that in logic the models are not themselves manipulable objects in the representation, while in CE the worlds are manipulable. From semantic network research comes the idea of parallel marker propagation and the idea that everything should be specified in terms of local connections.

In addition, the work which initially interested me in the idea of doing "semantic" computations using networks is Lamb's linguistic research into Stratificational Grammar [Lamb 1966, 1969]. Much of the philosophical perspective which underlies CE is derived from Lamb, and so are some of the notational conventions (e.g. the symbol for CE's partition constraint is the same as Lamb's "ordered OR.") It is clear to me that without Lamb's influence the research leading to this document would never have occurred.

7.2 CE's Future

As mentioned in section 5, CE can not currently handle events, procedures, or arithmetic -- one obvious possibility for future research is to extend CE so that it does handle these. Representing events is currently one of the hard problems in AI research -- the clean semantics of CE's notions of "world" and "world class" may prove useful here.

The appendices deal with several topics which are not as well worked-out as the body of this document -- fleshing out the details of these topics ("finding,"

"implementations," and "knowledge about knowledge") is another task for future research. "Knowledge about knowledge" is especially promising because this topic concerns itself with the relationships among different worlds (the "real world," the world which represents some person's beliefs, etc.), and CE is a representation in which it is easy to state facts about worlds. "Finding" is a topic which must be worked out in more detail in order for CE to be practical for real data-base applications, and issues of "implementation" are of course always important when one is proposing a new representation. I believe that CE is a system which can in fact be built upon by myself and others, and that it is not just a pretty toy.

Appendices

A "Finding"

The body of this document is oriented towards consistency checking, which entails the ability to answer "yes/no" queries. This appendix briefly discusses three techniques for handling "find" queries, which are queries of the form "find all objects 'x' such that ...." Within the CE framework, the starting point for handling such a query is to construct (intensionally) the class which contains all the desired objects, and then to determine which objects are in fact in that class. For example, "Find all the children of Jane Smith" would be answered by constructing the class 'Z' in figure A-1 and then finding all the objects which are constrained to be in 'Z'. Clearly, an object is in 'Z' if and only if it is known (by the data-base) to be a child of Jane Smith.

The three techniques presented below are different ways of finding all the objects in a class such as 'Z'. Of course, if the query is "Find one of ..." then the "Find all" procedure can be run until the first object is found (after which the procedure can be halted).

A.1 "Find the ..." using Object Identification

If it is known that the "Z" class contains exactly one object, then finding is quite simple. Figure 2-6b shows an inference for "Find the sex of Jane Smith." Here, the "Z" class is the object-class 'x' (which is constrained to contain the single sex of Jane Smith). Starting an "=x" label from this class, the goal is to have 'x' identified with some object in the data-base (in this case, 'female'). As described in section 2.3, this identification occurs when a "+x" label collides with an "=female" label -- this means that 'x' and 'female' are the

same object. Thus the label propagation procedure has found the single object in the "Z" class.

A.2 "Find all ..." using Suction

When the "Z" class contains more than a single object, there is still a simple technique which can be used to do the finding. Consider figure 3-1 and the query "Find all redwoods." The basic idea behind the "suction" technique is to start a generated "-g0096" label from the "Z" class (in this case, REDWOODS), and then note all the objects to which the "-g0096" propagates. In 3-1, it is clear that the "-g0096" will propagate down the taxonomy using rule (p2) until it reaches all the objects at the bottom (for example, the objects which are the known Sequoia National Park redwoods) -- these objects are not shown in the figure.

It remains to be demonstrated that all the objects reached by the "-g0096" are indeed in the "Z" class. Suppose such an object (call it 'obj') were not in "Z". Then it would be consistent to label "Z" with "-obj". This "-obj" label could then propagate in the same manner as the "-g0096" did, and thus reach 'obj'. Since an object-class such as 'obj' can broadcast an "-obj" label, the "-obj" and the "-obj" would collide at 'obj', indicating an inconsistency. Thus the original assumption that 'obj' was not in "Z" is false, so in fact every object reached by the "-g0096" must be in the "Z" class.

This technique is called "suction" because the "Z" class sends out "-" labels in an effort to "pull objects into it." Having done a suction inference, the user is left with the problem of determining which data-base objects have been reached by the "-g0096". From the user's point of view, the data-base consists of a black box with some classes being

accessible as "terminals" -- the terminals consist of all the classes which the user knows something about. In particular, the object-classes in which the user is interested will be among the terminals. So, what the user needs to do is to look at all the object-class terminals and see which ones have the "-g0096" on them. Of course for a large data-base the user will need some sort of automatic monitor to watch the terminals and signal the user when an interesting label (such as the "-g0096") arrives. It is not difficult to see how such a monitor can be constructed (either out of hardware, or as part of a CE system simulator).

A.3 "Find all ..." using Reflection

The one problem with suction is that it is very incomplete -- there are many simple cases in which the generated "-g0096" becomes blocked and can not propagate far enough to reach the relevant objects. Figure A-2 shows a simple case of this. The network above the dotted line states that Jane Smith is a child of Mary Smith, and that Billy Jones is a child of Jane Smith. Now, the query is "Find all grandchildren of Mary Smith." The network below the dotted line constructs the class 'Z' to be the children of the children of Mary Smith, and the task is to find all such objects in 'Z'. Clearly, 'Billy-Jones' is an object in 'Z'. To see this in terms of label propagations, just start an "-Billy-Jones" label from Billy-Jones. By applying rule (b2) twice (along with rule p1) and then rule (b1) twice, a "+Billy-Jones" will propagate to 'Z', indicating that Billy-Jones is indeed in 'Z'.

However, suction fails to propagate a "-" label from 'Z' to 'Billy-Jones'. Starting from a "-g0097" on 'Z', there are no propagation rules which can be applied. What is needed is to start from the other end -- some means is needed to have 'Billy-Jones' start an "-Billy-Jones" from itself. As described in section 2.2, it is infeasible to have <u>every</u> object-

class in the data-base broadcast an "=" label, since this would swamp the system. Thus the goal of the "reflection" technique is to start a special kind of label from 'Z' which will reach the relevant objects (such as 'Billy-Jones') and probably some of the irrelevant ones too. All objects reached by this label will then reflect back their "=" labels, and then only the ones which are truly in 'Z' will eventually have a "+" label propagate to 'Z'. That is, the initial label sent out from 'Z' is meant to select a small number of objects (relative to the size of the data-base), and then these selected objects are tested for membership in 'Z' by having them broadcast their "=" labels.

The second stage of this process is already well defined -- as described in section 2.3, an object-class broadcasts an "=" label whenever it is reached by any other label (including the one sent out from 'Z'). It remains to describe the first stage -- the label to be sent out from 'Z'. This must be a new type of label, since "+", "=", and "-" labels can be easily blocked. Call this new label the "?" label. Unlike the "+", "=", and "-" labels, the "?" label has no real semantics -- a "?" label on a class simply means that the class is somehow "associated" with the "Z" class.

One possible propagation rule for "?" is to say that when any constraint detects a "?" on any attached class-point, it should propagate a "?" to all of its other attached class-points. This rule results in a Quillian-like "wave front" of "?" labels, propagating out from the initial "Z" to every class which is connected (in the graph-theoretic sense) to "Z" by some path of constraints and points. Since every relevant object (i.e. ones in "Z") must be connected to "Z" by <u>some</u> such path, this propagation technique is guaranteed to have "?" reach every relevant object (which will then reflect back its "=" label). Unfortunately, this particular propagation rule will cause the "?" to reach <u>every</u> point in the network, unless

the network is really two or more totally disjoint ones. That is, everything is likely to be connected (via some possibly long path) to everything else, so this propagation rule is not selective enough.

The details concerning a more adequate set of propagation rules for "?" have not yet been worked out -- that is why this material appears here in an appendix instead in the body of the document.

B Implementations of CE

This appendix describes three possible implementations of CE. The first uses unconventional cellular hardware to do the label propagations in parallel. The second is a modification of the first which uses an array of microprocessors (and is more feasible than the first with current technology). The third implementation is one that actually exists -- the label propagations are performed using an ordinary general-purpose computer.

B.1 Using Cellular Parallel Hardware

The basic idea is to have each constraint in the data-base be an active processor which continuously looks at its attached class-point for patterns of labels which match the constraint's propagation rules. When such a match is found, the processor propagates the appropriate labels to other class-points. Each class-point is a register which indicates what labels (if any) are currently on that point. Thus each constraint is a processor which reads and writes the registers corresponding to the class-points attached to that constraint.

The main limitation of such a CE machine lies in the number of labels which might pile up on a single class-point. There are basically two ways to approach this. The first approach is to endow each point with a fixed number (N) of slots, each slot containing a pointer to a label. Here, the number of labels on each point is limited to N, but the total number of labels in the whole network may be much greater (since each point may have up to N different labels). The disadvantage of this scheme is that each of the label slots will be several bits wide (the log of the maximum number of labels allowed in the network at one time). If this number is M, then each point requires N times M bits, and the constraint processor needs to be able to copy and compare these M-bit labels. This requires either M

wires connecting each constraint to each of its points, or some multiplexed scheme using fewer wires. Either of these alternatives is undesirable because it is important to minimize both the number of wires and the complexity of each processor.

The second approach allows both for fewer wires and simpler processors. It relies on limiting the number of different labels in the entire network (not just on each point). Assuming that N labels are allowed, then each point need only have 2N bits worth of storage: Each label is represented by two bits, whose four states indicate "+", "-", "=", and "none." With this scheme, copying a label involves changing only 2 bits (the bits being indicated either by a pointer of log N bits, or by multiplexing). Through all points runs the "point bus" which sends reset signals to all points and handles the signaling of inconsistencies. A point will signal an inconsistency if it is told to set a label's state to "-" and its current state is "+" or "=", and vice versa.

The partition constraints and the object constraints do not need to know what is "inside" a label (being interested only in the +/-/= states), but the other constraints do have to be able to decompose a complex label into its simpler components. Hence the system must contain one N-slot "label memory" which gives the structure for each label -- either an object/world pair, or an ordered-pair/world triple. The "world" part of each label is a log-N pointer to the label in the label memory which is the original label's world-tag. To allow access to this memory, there must be a "label bus" which presents the contents of the label memory for inspection by the constraint processors. This can either be done on a request basis or in some synchronized manner (such as giving the contents of 1 thru N in order with the proper synch).

An additional complexity is that is that t-m constraints, world constraints, and

binary relationship constraints can generate new labels. This must be handled by the label bus either on a request basis or by having all unused label memory slots be assignable during their presentation-time on the bus. Also, the label memory must contain information about object identifications and about the world tree.

Finally, there must be a procedure for "growing" new wires when a new datum is added to the data-base, otherwise it would be necessary to change pieces of hardware every time new data is added. Consider the CE machine to consist of a tessellation of constraint-processor cells and point-registers. Each processor cell is of a fixed type (corresponding to the particular kind of primitive constraint which the cell implements), and has fixed wires attached to 1 through 4 point registers which it "owns" (the number of points being determined by the type of the constraint). Now, a new datum is added to the data-base by adding some new constraints. A new constraint is "added" by selecting a currently-unused processor cell of the correct type and "linking" its owned points to the appropriate other points in the network. When two point are "linked" it means that they represent the same class and hence their registers must be kept in the same states -- they must be "wired together." Now, figures 2-1g and 2-1h show such a "wire" -- it is a partition constraint with only one subclass. Thus the task of "growing a wire" between two point-registers translates to the task of activating enough of the currently unused partition constraints to form a chain between the points. The chain will actively propagate all the labels from each of the original point-registers to the other. To grow the chain, a Quillian-like "wave front" of special labels which propagate only through unused partition constraints can be started from each of the original point-registers. The place where the two wave fronts first intersect is then known to be part of the shortest path connecting the original points. The

unused partition constraint which is at the place of intersection can activate itself, and propagate a wave of "activate yourself" labels back along both wave fronts. Those partition constraints in the original wave fronts which do not receive "activate" labels are not part of the chain being grown, and thus remain unused.

Of course, many details still remain to be worked out before a cellular CE machine could be built, even if current LSI technology is capable of the task.

B.2 Using Microprocessors ("active pages")

Since the cellular machines proposed above are not likely to exist for a while yet, it would be convenient to be able to use current technology to implement the CE parallelism. One way to do this is to segment the network into local "pages", each with its own microprocessor for propagating labels within the page. The network within a page would be implemented as a linked-list structure, so the problem of growing wires in cellular hardware does not occur. Each processor has access to the "label bus" as above, in addition to a common "mail bus" which is used to export (and import) labels which cross page boundaries.

It would not be unreasonable with present technology to have a 1-page chip which contains the processor and the linked-list memory for a single page. These could then be stacked up to make the data-base, with more chips being added to the stack as the data-base grows.

B.3 Using a Digital Computer

A CE system has been implemented to provide the data-base and the low-level inference capability for the "MACSYMA Advisor" [Genesereth 1975]. MACSYMA is a very complex system for doing symbolic mathematics, and the advisor is a proposed subsystem to aid users when they need help. The user will interact with the advisor in more-or-less natural English. The advisor uses the English input, the history of the user's interactions with MACSYMA, and the advisor's own knowledge about MACSYMA in general and this user in particular in order to formulate its advice. The advisor consists of an English parser, a high-level problem solver, and a low-level data-base and inference capability (for which CE is used). The data-base contains essentially all of the advisor's information about MACSYMA and about the user. Genesereth estimates that the CE data-base will contain about 20000 constraints.

The CE system used for the advisor is implemented in LISP without any multiprocessing. Parallelism is simulated by having a priority queue of propagations to be done. With a sequential system, it is very important to have good heuristics for deciding which propagation to do next -- doing them purely breadth-first (as parallel hardware would) is quite wasteful. Two of the heuristics used are:

(1) Propagate "+" labels in preference to "-" ones. For example, it is much less expensive to propagate "+" labels upwards in a taxonomy (such as figure 5-1) using rule (p1) than it is to propagate "-" labels downwards using rule (p2). Basically, this heuristic says that is usually more informative to know what something <u>is</u> as opposed to what it is <u>not</u>.

(2) Propagate existing labels in preference to generating new ones. This is a useful heuristic since the implementation is limited in terms of the number of different labels that

can exist at one time. In addition, it prevents inferences from wandering off into long nestings of relations (caused by rule b2), such as "my father's brother's political party's candidate."

There are other heuristics which will be described in Genesereth's report on the advisor.

C Knowledge about Knowledge

The problems involved with representing "knowledge about knowledge" are interesting both technically and philosophically; they are also quite difficult. This appendix shows how some of these problems can be resolved by using the CE "world" construct, which allows explicit statements to be made about various worlds (both physical and metaphysical). This appendix is divided into five sections: The first two deal with "belief"; the third and fourth deal with "knowledge" (i.e. "true" beliefs); and the fifth briefly discusses modal logic. The example used throughout this appendix is the following: "Billy knows who Jane's real father is, and she doesn't know that he knows."

C.1 Belief

To introduce the idea of "belief," this section uses a simplified version of the above example -- the full version is used later. The simplified version is: "Billy believes that Jane's real father is John, and Jane doesn't believe that Billy believes it." Figure C-1a represents this using CE.

Region (a) of C-1a states that Jane's father is the object 'jf' (named acronymically). Without having any other information about 'jf' (which region (a) does not), all this says is that Jane has a unique father.

Region (b) states that W-JF=J is the class of all worlds in which Jane's father equals John. What it literally says is that W-JF=J is the class of all worlds in which object-class 'jf' is a subclass of the object class 'John'. (As has been mentioned many times, one object-class is a subclass of another if and only if the two objects are the same.)

Region (c) defines W-BILLY to be the class of all worlds which are consistent

with Billy's beliefs. Of course, the relation class 'BELIEVES' has no a priori meaning to CE (just as 'FATHER-OF' does not), but it is assumed that the user always uses 'BELIEVES' to mean the relation between an individual and all the world-objects which are consistent with that individual's beliefs. The name "BELIEVES" is of course arbitrary and is not part of the data-base in any case -- the important structural feature of 'BELIEVES' is that all references to an individual's beliefs are made via this class. I belabor this point only to emphasize that nothing new has been introduced -- 'BELIEVES' is just an ordinary binary relation.

Region (d) states that Billy believes that Jane's father is John. That is, every world in W-BILLY (i.e. every world consistent with Billy's beliefs) is also a world in W-JF=J (i.e. is a world in which Jane's father equals John). As in section 2.5, the use of the subclass constraint means that W-BILLY is stronger than W-JF=J -- Billy believes at least that Jane's father is John, and he may believe other things.

Region (e) adds the constraints that Jane does not believe that Billy believes that her father is John. As with W-BILLY, the class W-JANE contains all worlds which are consistent with Jane's beliefs. The class W-BBJF=J is all worlds in which Billy believes Jane's father is John (i.e. all worlds in which W-BILLY is stronger than W-JF=J). The partition constraint then means that there is no world in W-JANE which is also in W-BBJF=J (i.e. the two classes are mutually exclusive). That is, none of Jane's possible world-views allows for the possibility that "Billy believes ...". More literally, in every one of Jane's worlds it is the case that W-BILLY is not a subclass of W-JF=J. That is, in all of Jane's worlds there is some world in W-BILLY which is not in W-JF=J, so Billy has at least one possible world in which Jane's father is not John.

C.2 "I believe ..."

There is one remaining problem with figure C-1a. Region (e) states that in all of Jane's worlds it is not the case that W-BILLY is a subclass of W-JF-J, while region (d) states that in all worlds it is the case that the W-BILLY is such a subclass -- since the subclass constraint in region (d) is not relativized, it is "enabled" for all worlds. Thus it is necessary to relativize region (d). Well, W-BBJF-J is already defined as being exactly those worlds in which the subclass constraint holds. Thus the subclass constraint in region (d) can be deleted, and something new should be connected to W-BBJF-J. The question is, who is the one who believes that "Billy believes ..."? The answer is that the data-base believes it. Therefore a world-class is needed to represent the data-base's "point of view" -- call it 'I'. Then figure C-1b shows what should be added when region (d) is deleted -- it states that I (the data-base) believe that Billy believes.... Behaviorally, the class 'I' is used by putting a "+inf" label on it as part of the initial labeling. This enables whatever is attached to 'I', such as the W-BBJF-J.

Now in one sense everything in the data-base is qualified by "I (the data-base) believe such-and-such" -- after all, the data-base itself can be viewed as being an entity with a point of view, much as Billy and Jane are. The reason for needing an explicit representation for "I (the data-base)" is that it may be necessary to represent other points of view which conflict with the data-base's. In the above example, Jane's point of view regarding Billy's beliefs is different from the data-base's, and this conflict is what motivated the introduction of 'I' in the first place. By having an explicit 'I', the data-base can keep track of the difference between facts it believes to be true in all worlds (including Jane's, for example), and facts which it believes to be true only in its own worlds. The next

section deals with how the data-base can represent the fact that some of these many worlds are "true" and the some (like Jane's) are not true.

C.3 Knowledge, God, and Wisdom

The simplified version of the example, represented in figure C-1, deals only with "belief." However, the original example deals with "knowledge" -- not only does Billy believe that Jane's father is John, he knows it. One solution is to say that whatever the data-base believes is necessarily "true" (at least insofar as the data-base is concerned). This is quite reasonable -- after all, how could the data-base ever accomplish anything if it were in continual doubt about the validity of what it believed? Of course, there may be cases in which it is desirable to represent the fact that the data-base considers itself to be an unreliable source of information regarding certain topics -- this can be handled by the "probability" mechanism described in section 3.5.2. So, in the context of the above example, the fact that it is true that Jane's father is John can be represented by adding the fragment shown in figure C-1c: Now, both John and the data-base believe that Jane's father is John, which makes John's belief "true" (insofar as the data-base is concerned).

The problem with this scheme is that "truth" is defined to be exactly that which the data-base believes. It is reasonable to say that the data-base's beliefs contain only true statements, but it is unreasonable to say that the data-base's beliefs contain all true statements -- the data-base certainly does not have complete information about everything that is true in the universe. If the data-base does not have this complete information, who does? The solution is to create a world-class W-GOD, which is meant to contain all the worlds which are consistent with "reality."

Now, it is desirable to retain the above notion that everything the data-base believes is true, even though the data-base's beliefs do not encompass all truths. This is represented in figure C-2a. As usual, the direction of the subclass arrow is from stronger to weaker: God's beliefs (i.e. reality) are consistent with the data-base's beliefs, but God believes many other things in addition. Note that W-GOD might be so strong as to be a single object: Representing W-GOD as an object-class would mean that God allows only one possible universe. Although the issue of whether the universe is "one" or "many" might be of philosophical interest to some, it appears to have no technical importance here -- I have not yet found any cases for which it makes a difference whether or not W-GOD is an object-class.

Using W-GOD makes it possible to abstractly describe two different aspects of an entity's "wisdom." The first aspect is that everything the entity believes is in fact true -- this is shown in figure C-2a. The second aspect is that the entity knows all there is to know (i.e. all truths) -- this is shown in C-2b. The combination of the two of course means that the entity knows exactly what God does. Now, this notion of "absolute wisdom" is clearly not very useful -- often someone is considered to be wise only with respect to a given subject area. One possible solution is to divide God's knowledge into several domains -- figure C-2c divides knowledge into the domains of "accounting," "mathematics," and "other." Then, to say that Billy knows everything there is to know about accounting, the network in figure C-2d can be used. Of course, domains such as "accounting" are much too large for most purposes -- they can be further divided until an appropriate size is reached (such as "Billy knows all there is to know about accounting for mergers using the pooling-of-interests technique").

Of course, just using the name "W-ACCOUNTING" does not mean that the data-base thereby knows what the domain of accounting is. It may be necessary for some applications to constrain W-ACCOUNTING appropriately -- this is a topic for future research.

C.4 Interworld Objects

The above discussion of W-GOD does not seem to have much direct relevance to the example of Jane's father -- everything works correctly by just using the construction in figure C-1c (without needing to introduce W-GOD). As detailed above, this construction involves the assumption that the data-base knows everything that is true, but this assumption causes no difficulty in the example because in fact the data-base does know all the relevant facts. However, figure C-1 does not quite handle the original example, which is "Billy knows who Jane's father is...", not "Billy knows Jane's father is John." The problem is that Billy may know that Jane's father is John (or whoever), but the data-base does not know it. That is, there is an entity (Billy) who knows more than the data-base, which means that the data-base can not be used as the arbiter of truth. This is why W-GOD is needed for the example.

A rephrasing of the relevant part of the example is "Billy believes that Jane's father is α, and Jane's father is in fact α." That is, both Billy and God believe that Jane's father is identical to the object 'α', but the data-base does not know anything else about 'α'. As a first attempt at representing this, consider figure C-3a, which is meant to replace the relevant parts of C-1a.

The problem with C-3a is that the object-class 'α' can be a different extensional

object in different worlds, just as in figure 2-3a 'the-president-of-the-US' can be different people at different times. In particular, 'α' can be different for God and for Billy, which goes against the idea that God and Billy should have the same 'α'. The solution is to introduce a new primitive constraint, the "interworld object" constraint, the symbol for which is shown in figure C-3b. This constraint acts the same as the ordinary object constraint, except it always represents the same extensional object in all worlds. Its label-propagating behavior is the same as the normal object constraint's rule (ol) -- it broadcasts an "-obj" label. The difference is that collisions between such a label and a "+obj2" label can occur regardless of whether or not the two labels have the same world-tag. Thus objects in different worlds can become identified with each other. This behavior implements the fact that the interworld object (an intensional construction) represents the same extensional object in all worlds. By making the object-class "α" an interworld object, the example is completed: The data-base knows that Billy knows the identity of Jane's father, without the data-base itself knowing that identity.

C.5 Modal Logic

Modal logic deals with (among other things) the distinction between "necessary" truths and "contingent" truths. A necessary truth is one that is true from the definitions of the terms used -- for example, it is necessarily true that all crows are birds, if we use "crow" and "bird" with their normal meanings. However, it is only contingently true that all crows are black -- no logical laws would be violated if a pink crow appeared tomorrow. Modal logics are systems in which the distinction between "necessary" and "contingent" can be explicitly specified. In this sense, CE can be used as a modal logic. Figure C-4a states that

all crows are necessarily birds -- the subclass constraint holds in all possible worlds. Figure C-4b states that all crows are contingently black -- the subclass constraint holds in this reality, but it might not hold in some other. That is, the data-base allows for the possibility that in some worlds it might not be the case that all crows are black.

Another aspect of modal logic deals with notions such as "want," as in "John wants Jane to be with him." A rough translation of this is that John "desires" a world in which Jane is with him. This can be represented directly by introducing a binary relation DESIRES with the same form as BELIEVES -- it relates an individual to a class of worlds. Working out the details of this is a topic for future research.

D Design Decisions

As with any research project, certain more-or-less arbitrary design decisions had to be made during the formulation of CE in order that the work might proceed. This appendix briefly discusses some of these decisions and some alternatives to them. The main reason for including this appendix is that for certain applications of CE, some of these alternative designs might be preferable to the ones described in the body of this document.

D.1 Other Labels

There are alternatives to the use of "+", "-", and "=" labels. One such alternative is to eliminate the "=" label. The only interesting propagation rule which this change would eliminate is rule (b5), and for some applications this rule might not be necessary. Rules (b6) and (b7) were included only for completeness -- they refer to relations which contain only a single ordered pair, and this notion has not yet proved to be useful.

If "=" is eliminated, it is necessary to reformulate the process of object identification. With "=", identification occurs when an "=" collides with a "+" (or another "="). Without "=", identifications occur when a "+" reaches an object-class. Thus the "+" will have to come to the object, instead of the "+" and the object's "=" being able to meet "half way." This may reduce the number of object identifications -- whether or not this is important depends on the particular application.

Another alternative is to add the "-ϕ" label. If this label is on a class, it means that the class is empty. Here, "ϕ" is a notation for "all objects," so "-ϕ" means that all objects are known to be not in the class. This label fits in well with the existing propagation rules

which involve "=" -- after all, "=" means that the class is almost empty. For instance, rules (p5) and (u5) can be augmented to put "+" labels on the classes currently labeled "-" since they are necessarily empty. This additional information may be of use in some applications, particularly where there are many empty classes.

A further alternative is to scrap all of the existing labels in favor of a different scheme. The existing labels all refer to objects -- a different scheme can be used in which the labels refer to classes. In one such scheme, there are three labels:

"←A" on a class 'C' means that 'A' is a subclass of 'C';

"→A" on 'C' means that 'C' is a subclass of 'A';

"•A" on 'C' means that 'A' and 'C' are mutually exclusive.

In this scheme, "←" corresponds roughly to "+"; "•" corresponds to "-"; and a combination of "←A" and "→A" on the same point 'C' corresponds to "=" (since each class is known to be a subclass of the other). The propagation rules for the primitive constraints can be modified to handle these labels appropriately. The problems with this scheme stem from the fact that it does not refer to objects. For one thing, it is unclear how the idea of "object" (e.g. single-member class) can be formulated at all -- some other kind of label is probably needed. For another thing, it is not knowable whether the classes being referred to are empty or not -- all the classes could be empty and the labels would still propagate. This actually might be an advantage for applications where it is required that all classes in the network be non-empty.

D.2 Other Objects

Currently, CE has two kinds of objects which can appear in labels: simple objects such as "x", and ordered pairs such as "<x,y>". It may be useful to introduce other kinds of objects for certain applications. For example, consider integer arithmetic. Special integer-objects "3", "-17", etc. can be defined, and primitives such as "sum" and "difference" can be defined which use them. For handling inequalities, objects can be defined to represent integer intervals, and these can be manipulated by constraints which express the various inequality relations.

Another kind of object is the nested ordered pair, such as "<a,<<b,c>,d>>". Nothing in this document has required them, yet nothing explicitly prohibits them either. It clearly complicates things to allow labels which are arbitrarily deep nestings of pairs, but this complexity might be worthwhile in some cases. For one thing, LISP-like structures can be built by defining a primitive constraint for CONS which can be used to put these pairs together (and take them apart). For another thing, quantification could be performed without using the t-m constraint. The basic idea is to introduce a primitive for relation composition, and let the bound (quantified) objects be explicitly carried along in nested tuples. This technique is equivalent to Skolemization in first-order logic. It is not hard to work out the details of such a scheme for relation composition -- such details are not given here primarily because the resulting expressions seem to be very unnatural and awkward to use.

D.3 Other Primitive Constraints

Since CE is highly modular, it is possible to introduce a new primitive without having to worry about how it will interact with all the existing primitives. Indeed, throughout this document new primitives have been repeatedly introduced or proposed (e.g. in the immediately preceding section). Since CE is built entirely out of label-objects and primitive constraints, any addition to CE will be either in terms of new objects, new constraints, or both.

One such possible addition involves "procedural attachment" as described in section 5.4.2. Within the CE framework, this involves defining a new constraint which behaves normally -- it looks for appropriate patterns of labels on its attached class-points, and propagates new labels when such patterns occur. However, inside this constraint might be an arbitrary procedure for accessing the outside world either to receive information or to produce effects. For example, the class 'PERSONS' might be tied via such a procedure to an external file which lists all the persons. A "+obj/inf" label reaching this class causes the procedure to add 'obj' to the list of persons, and a "-obj/inf" causes an inconsistency if in fact 'obj' is a person on the list.

The major limitation on the power of such a procedure is that is must have a well-defined semantics in terms of label propagations. In the 'PERSONS' example, the semantics is easy to express since the file in the outside world corresponds quite directly to a CE class, but more complex procedural interaction with the outside world will certainly be more difficult to express in CE's terms. This is an area for future research.

Bibliography

1. Bobrow, Daniel G. and Allan M. Collins (eds.), Representation and Understanding, New York: Academic Press, 1975.

2. Boyce, R. F., D. D. Chamberlin, W. F. King III, and M. M. Hammer, "Specifying Queries as Relational Expressions: SQUARE", Proceedings of ACM SIGPLAN-SIGIR Interface Meeting, Gaithersburg, Maryland, Nov. 1973.

3. CODASYL Data Base Task Group, April 1971 report, ACM, 1971.

4. Codd, E. F., "A Relational Model for Large Shared Data Banks", CACM, June 1970.

5. Fahlman, Scott E., "Thesis Progress Report: A System for Representing and Using Real-World Knowledge", Cambridge: M.I.T. AI Lab Memo 331, May 1975.

6. Genesereth, Michael R., "A MACSYMA Advisor", Cambridge: M.I.T. Mathlab memo 10, December 1975.

7. Hawkinson, Lowell, "The Representation of Concepts in OWL", Cambridge: M.I.T. Project MAC Automatic Programming Group Internal Memo 17, 1975.

8. Hayes, Patrick J., "Some Problems and Non-problems in Representation Theory", Proc. AISB Summer Conference, U. of Sussex, July 1974.

9. Hewitt, Carl E., "Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot", Cambridge: M.I.T. AI Lab TR-258, 1972.

10. Karp, Richard M., "Reducibility among Combinatorial Problems", in Complexity of Computer Computations, R. E. Miller & J. W. Thatcher, ed., New York: Plenum Press, 1972.

11. Lamb, Sydney M., "Linguistic and Cognitive Networks", New Haven: Yale U. Linguistic Automation Project memo, June 1969.

12. Lamb, Sydney M., Outline of Stratificational Grammar, Washington, D.C.: Georgetown U. Press, 1966.

13. Levin, Michael I., John McCarthy, Paul W. Abrahams, Daniel J. Edwards, and Timothy P. Hart, LISP 1.5 Programmer's Manual (Second edition), Cambridge: The M.I.T. Press, 1965.

14. Martin, William, "OWL", Cambridge: M.I.T. Project MAC internal notes, 1974.

15. McCarthy, John, and P. J. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence", in Meltzer & Michie (eds.), Machine Intelligence 4, New York: American Elsevier, 1969.

16. McDermott, Drew V., and Gerald J. Sussman, "The CONNIVER Reference Manual" (version 2), Cambridge: M.I.T. AI Lab memo 259, 1973.

17. McDermott, Drew V., "Artificial Intelligence Meets Natural Stupidity", Cambridge: M.I.T. AI Lab mimeograph, October 1975.

18. Minsky, Marvin (ed.), Semantic Information Processing, Cambridge: The M.I.T. Press, 1968.

19. Moore, J. and Allen Newell, "How Can Merlin Understand?" in Gregg, L. (ed.), Knowledge and Cognition, Potomac, Maryland: Lawrence Erlbaum Associates, 1974.

20. Moore, Robert, "Reasoning from Incomplete Knowledge in a Procedural Deduction System", Cambridge: M.I.T. SM thesis proposal, April 1975.

21. Mylopoulos, J., A. Borgida, P. Cohen, N. Roussopoulos, J. Tsotsos, and H. Wong, "TORUS -- A Natural Language Understanding System for Data Management", Proc. 4th IJCAI, 1975.

22. Pople, Harry E., Jr., "A Goal Oriented Language for the Computer", in Newell, Representation and Meaning, Prentice-Hall, 1972.

23. Quillian, M. Ross (1967), "Semantic Memory", in Minsky (1968).

24. Quillian, M. Ross, "The Teachable Language Comprehender", CACM, August 1969.

25. Raphael, Bertram (1964), "SIR: A Computer Program for Semantic Information Retrieval", in Minsky (1968).

26. Reich, Peter A., "Symbols, Relations, and Structural Complexity", New Haven: Yale U. Linguistic Automation Project memo, June 1968.

27. Robinson, J. A., "A Machine-Oriented Logic Based on the Resolution Principle", JACM, October 1965.

28. Robinson, J. A., "The Generalized Resolution Principle", in Michie (ed.), Machine Intelligence 3, New York: American Elsevier, 1968.

29. Rulifson, J. F., J. A. Derksen, and R. J. Waldinger, "QA4: A Procedural Calculus for Intuitive Reasoning", Palo Alto: Stanford U. PhD thesis, November 1972.

30. Schank, Roger C., "Conceptual Dependency: A Theory of Natural Language Understanding", Cognitive Psychology, 1972, v. 5, pp. 552-631.

31. Simon, Herbert A., The Sciences of the Artificial, Cambridge: The M.I.T. Press, 1969.

32. Sussman, Gerald, Terry Winograd, and Eugene Charniak, "Micro-Planner Reference Manual", Cambridge: M.I.T. AI Lab memo 203, July 1970.

33. Winograd, Terry, "Frame Representations and the Declarative/Procedural Controversy," in Bobrow and Collins (1975).

34. Woods, William A., "What's in a Link: Foundations for Semantic Networks," in Bobrow and Collins (1975).

35. Zloof, Moshe M., "Query by Example", Proc. National Computer Conference, Anaheim, California, May 1975.

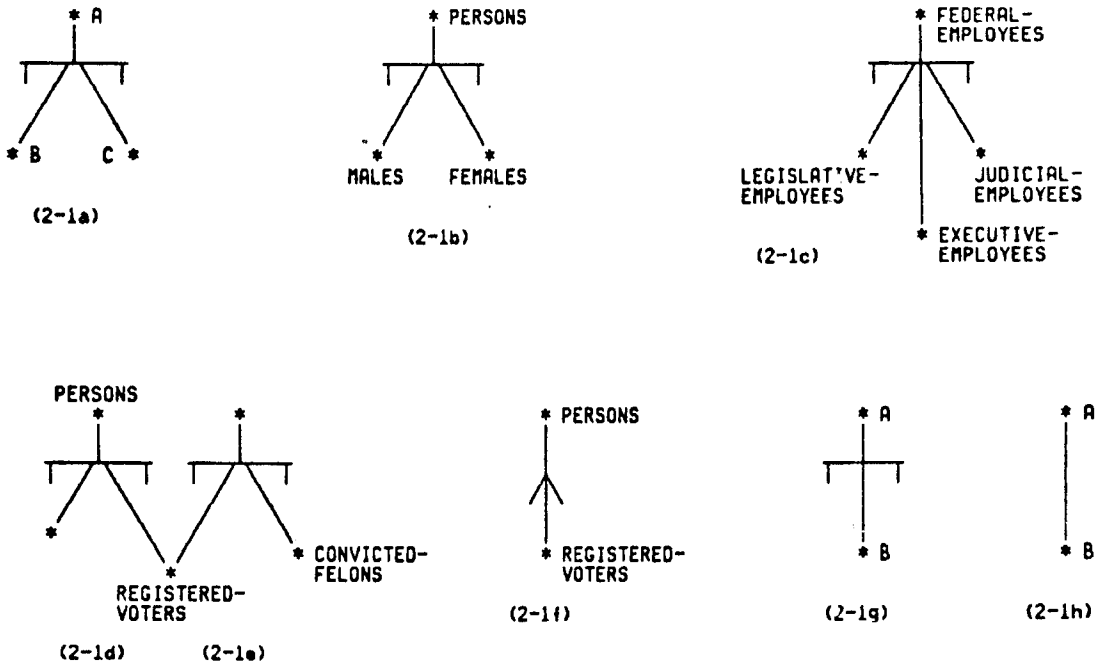


Figure 2-1 — The Partition Constraint

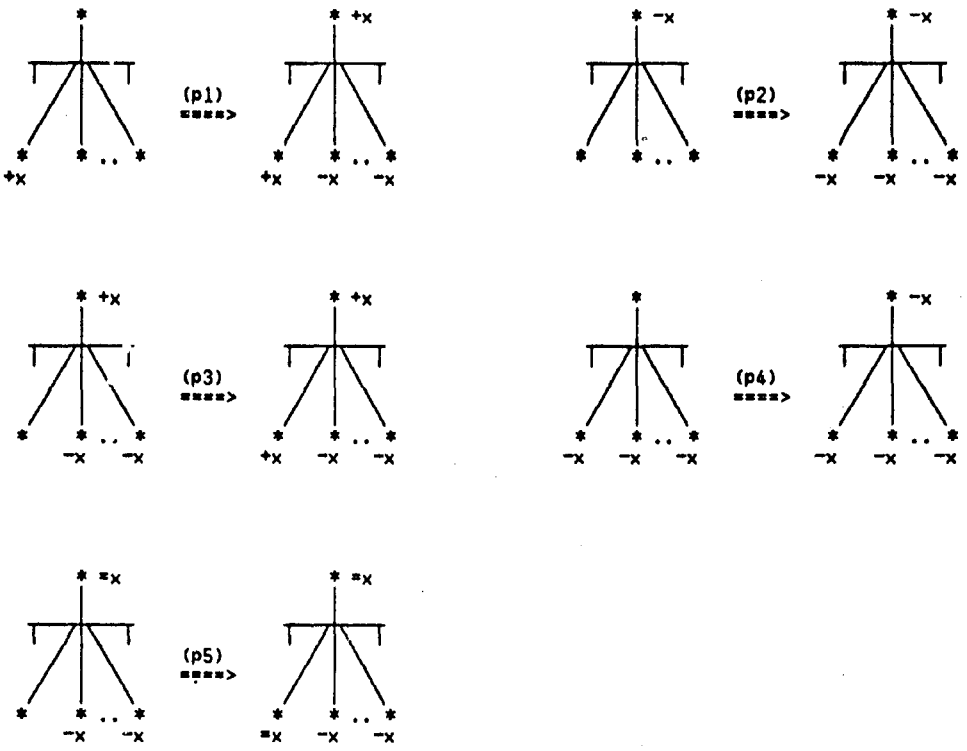


Figure 2-2 — Rules for the Partition Constraint

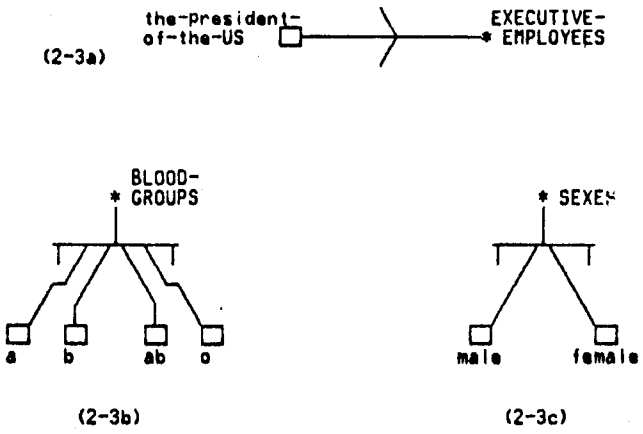


Figure 2-3 — The Object Constraint

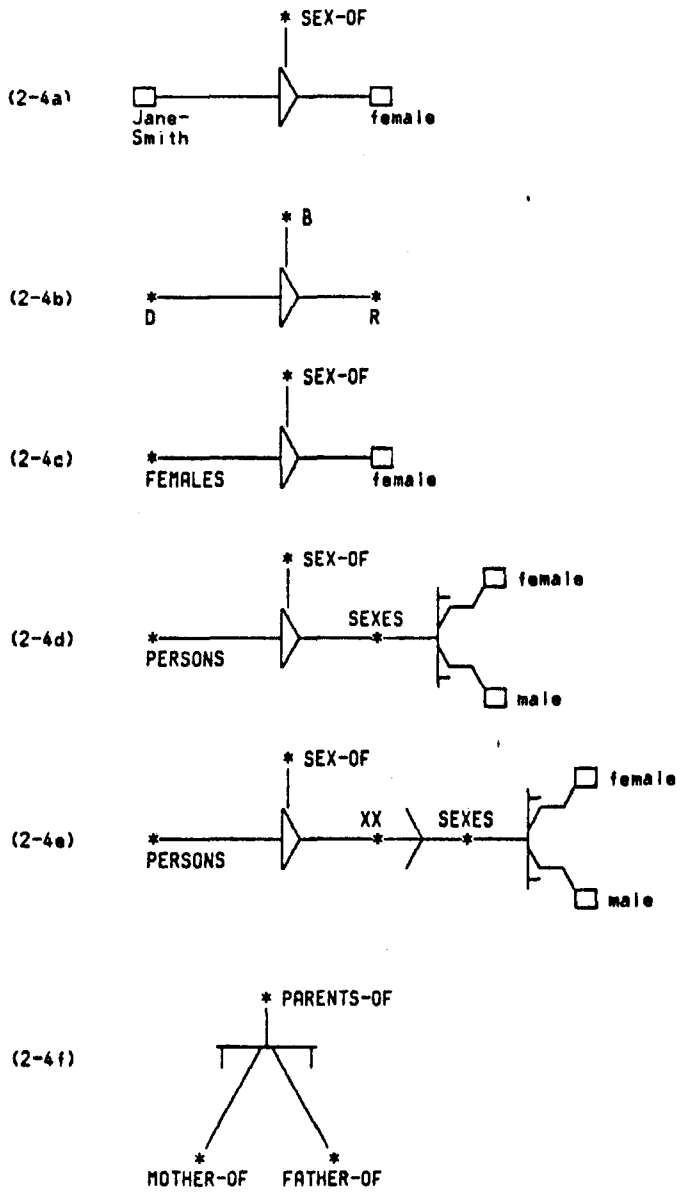


Figure 2-4 — The Binary Relationship Constraint

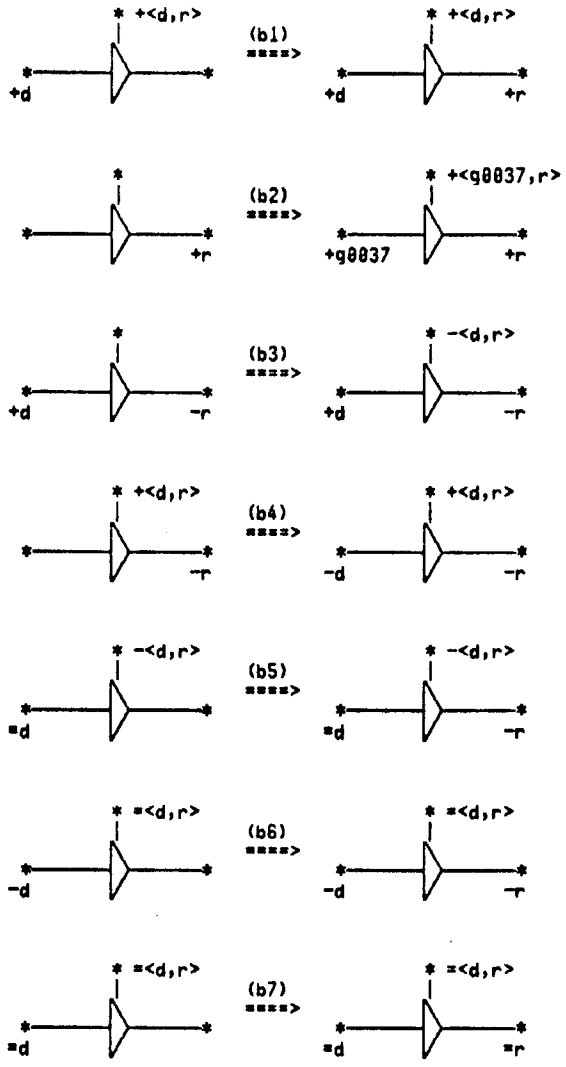


Figure 2-5 — Rules for the Binary Relationship Constraint

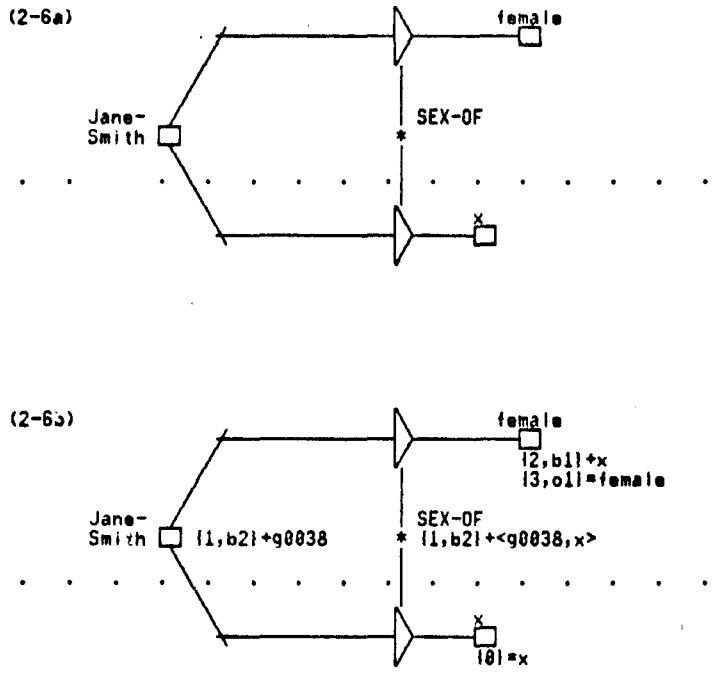


Figure 2-6 — A simple inference

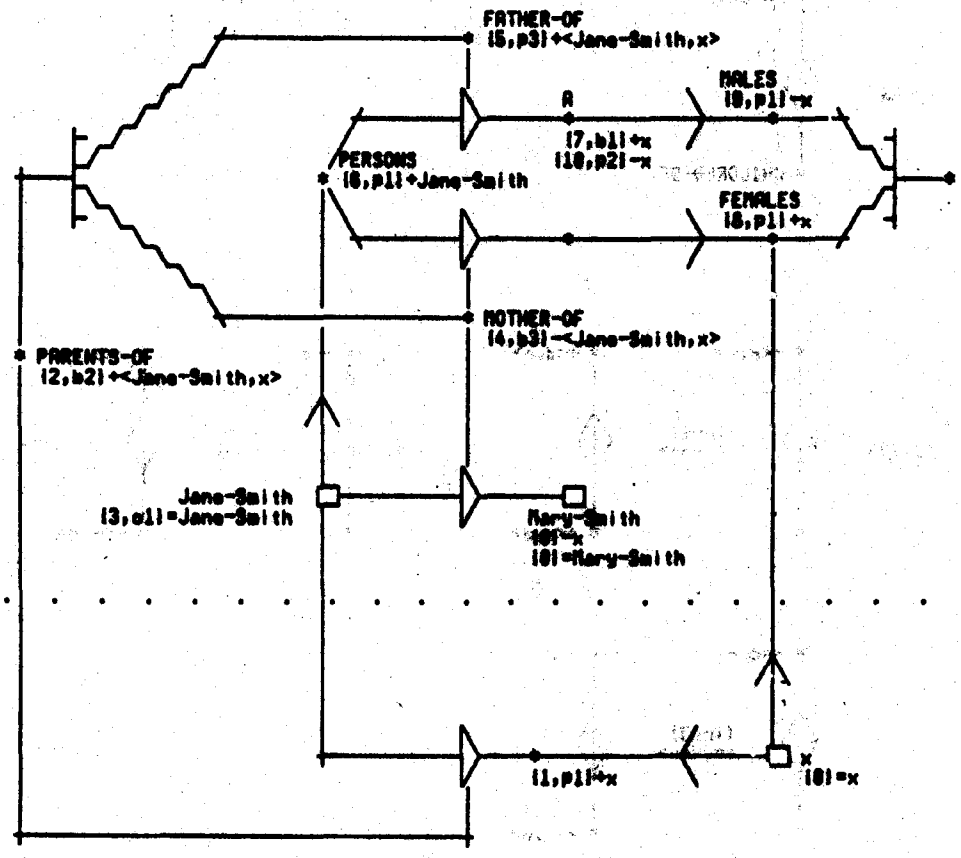


Figure 2-7 — A more complicated inference

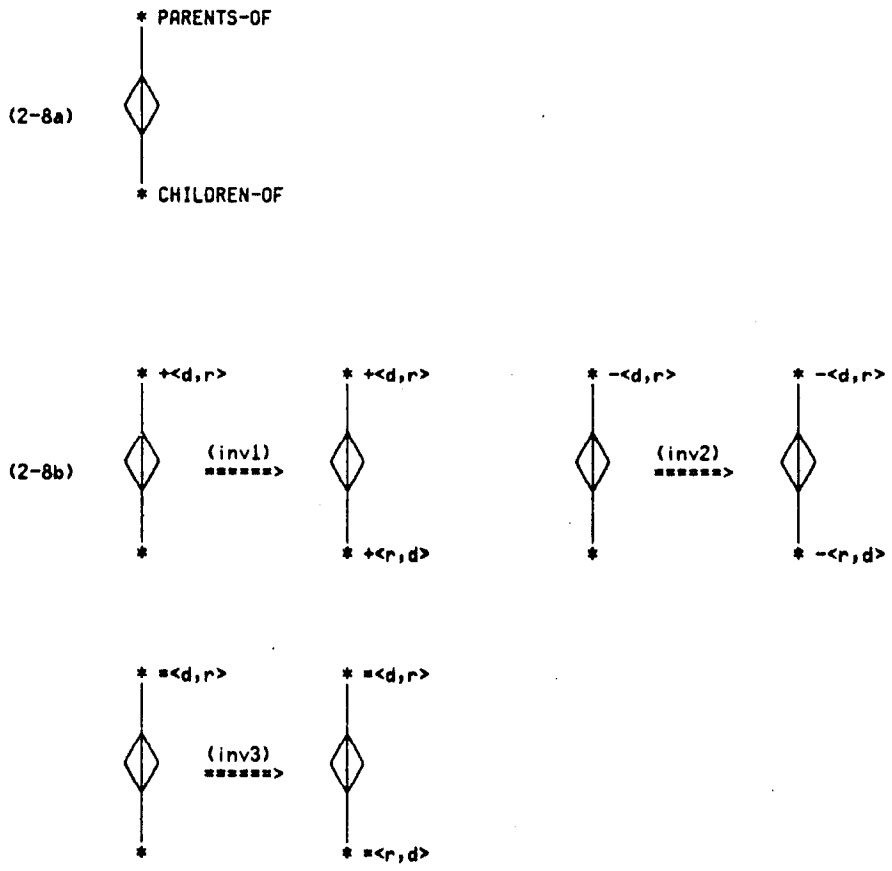


Figure 2-8 — The Inverse Constraint

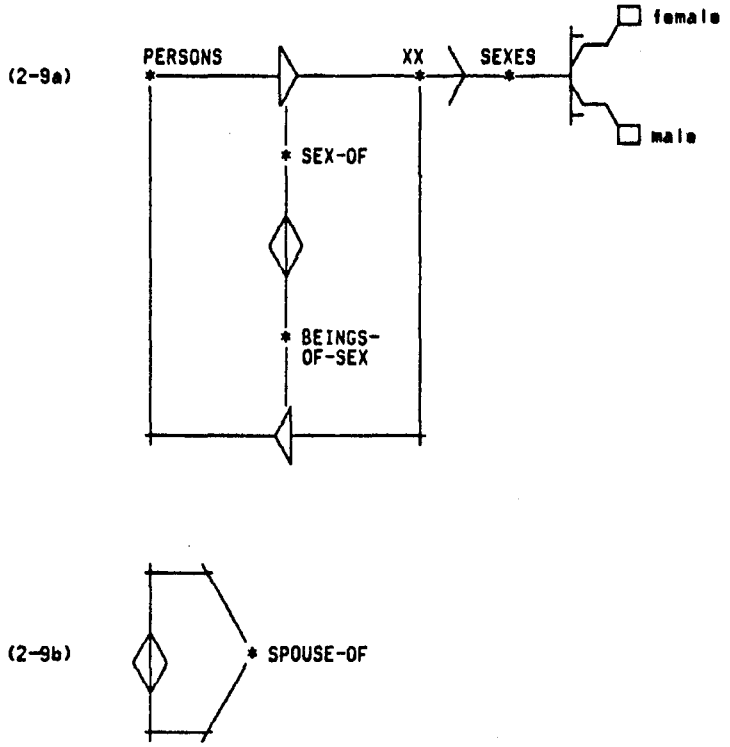
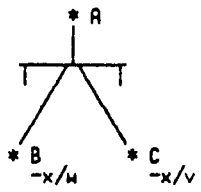
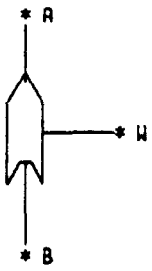


Figure 2-9 — Examples using the Inverse Constraint

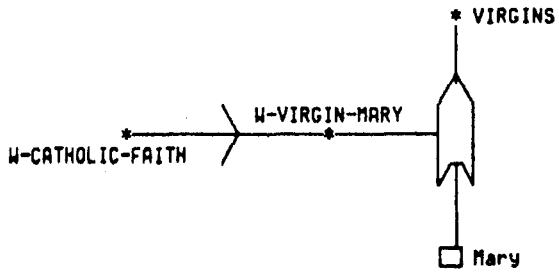


(2-18a)

(2-18b)



(2-18c)



(2-18d)

Figure 2-18 — The World Constraint

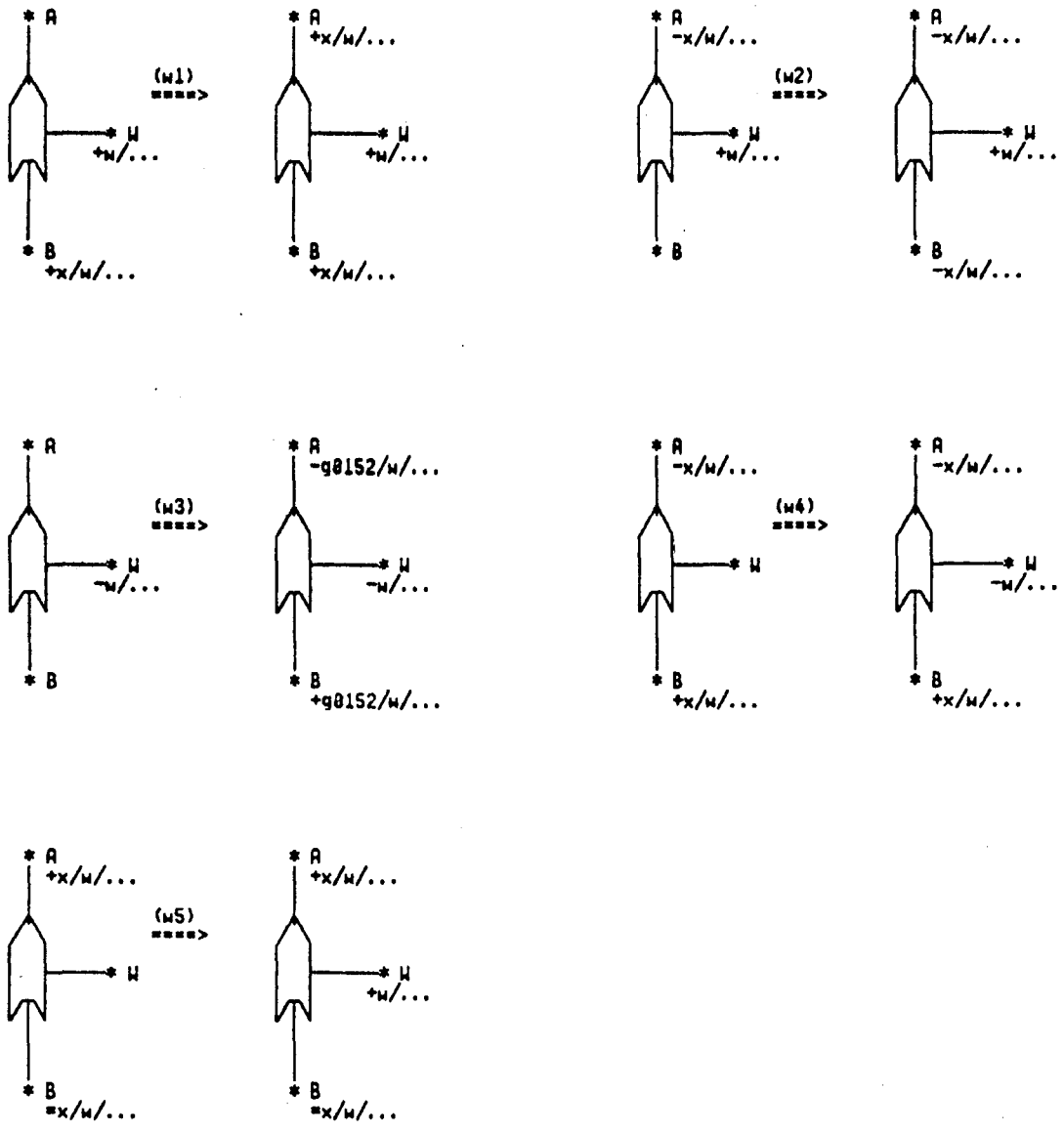


Figure 2-11 — Rules for the World Constraint

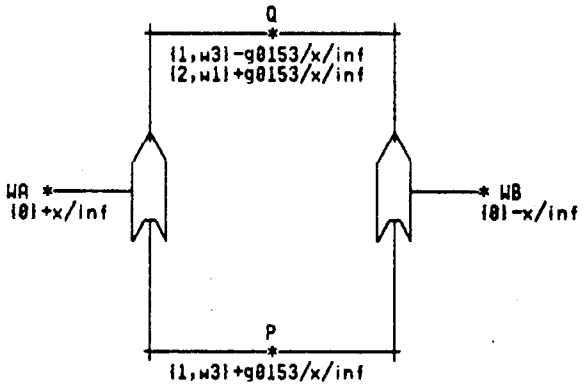
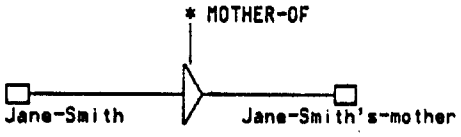
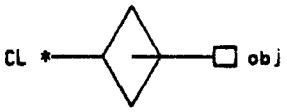


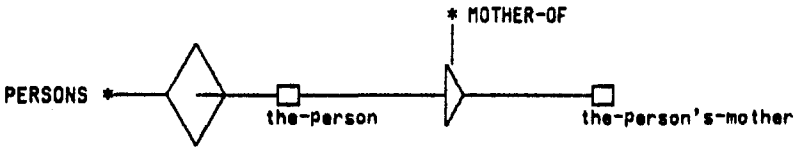
Figure 2-12 — An inference using the World Constraint



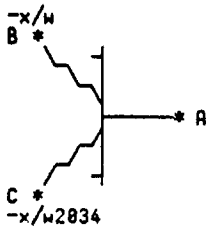
(2-13a)



(2-13b)

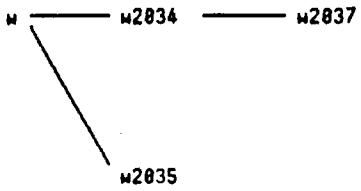


(2-13c)

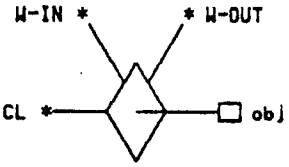


(2-13d)

(2-13e)



(2-13f)



(2-13g)

Figure 2-13 — The Typical-Member Constraint

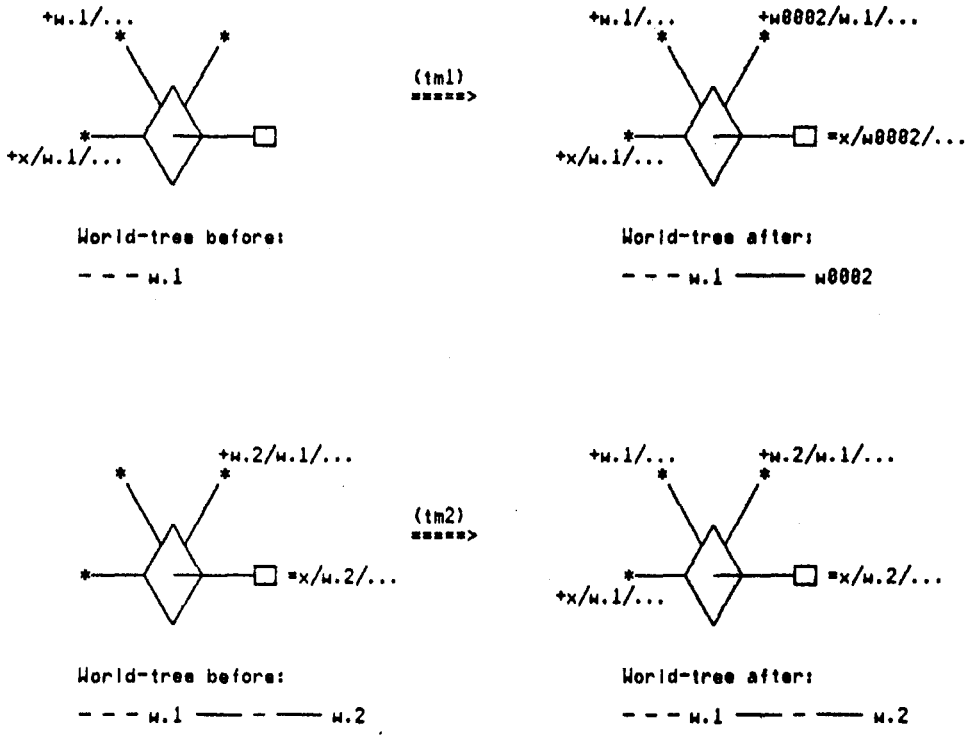


Figure 2-14 — Rules for the Typical-Member Constraint

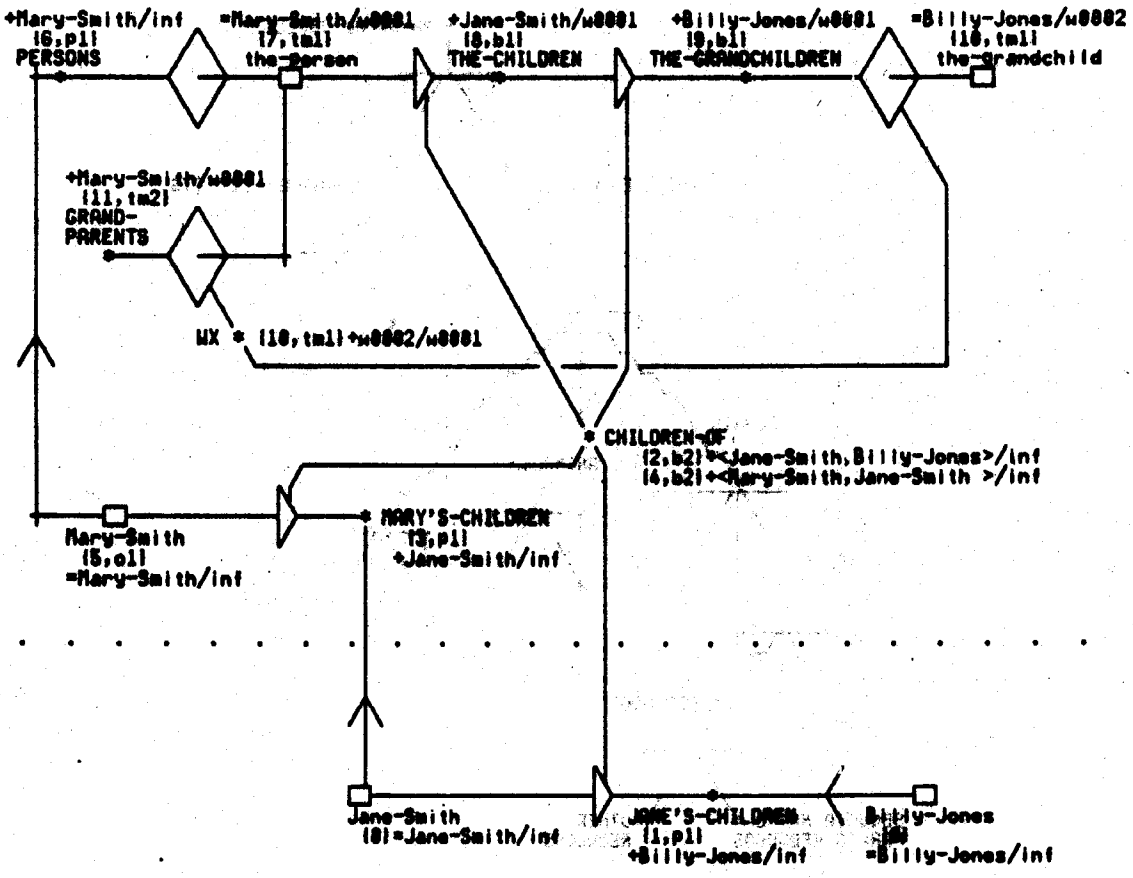


Figure 2-15 — An inference using the Typical-Member Constraint

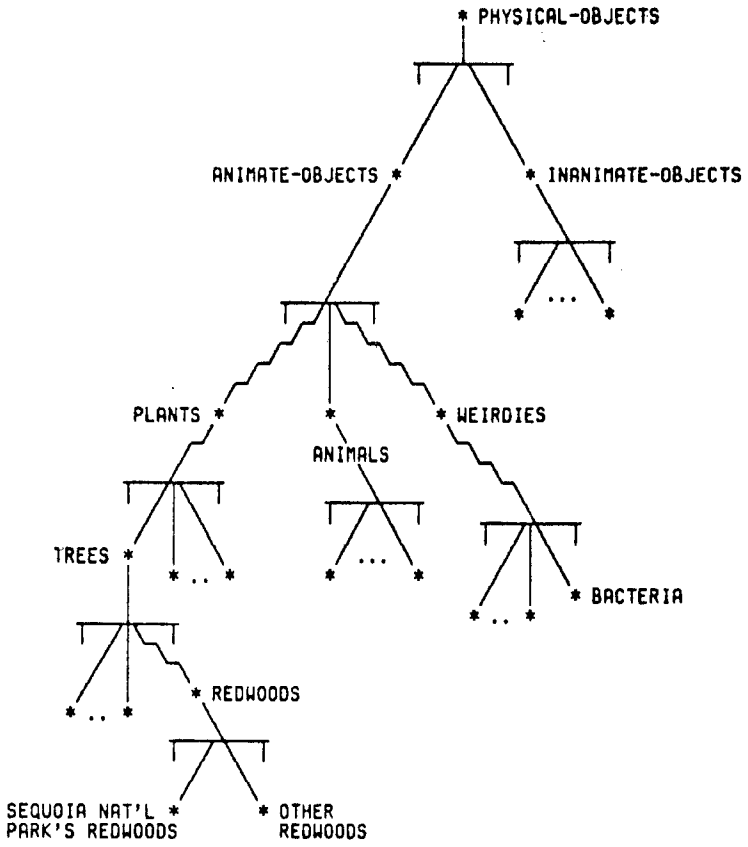
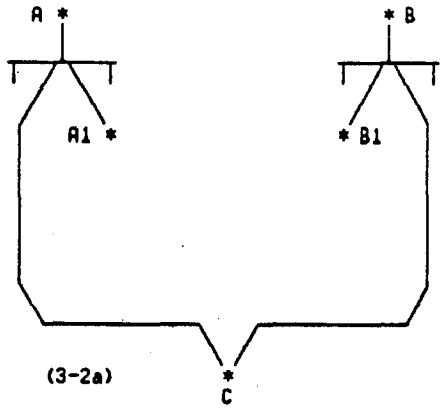
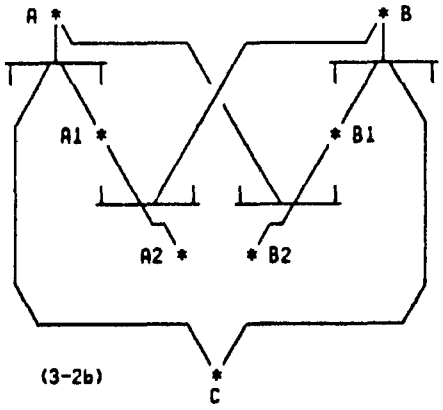


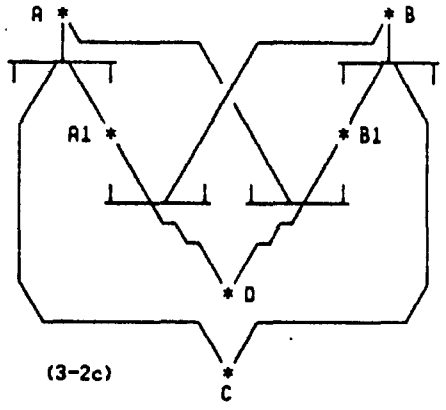
Figure 3-1 — A sample taxonomy



(3-2a)

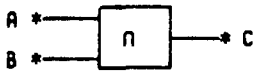


(3-2b)



(3-2c)

(3-2d)



(3-2e)

Figure 3-2 — Intersection and Union

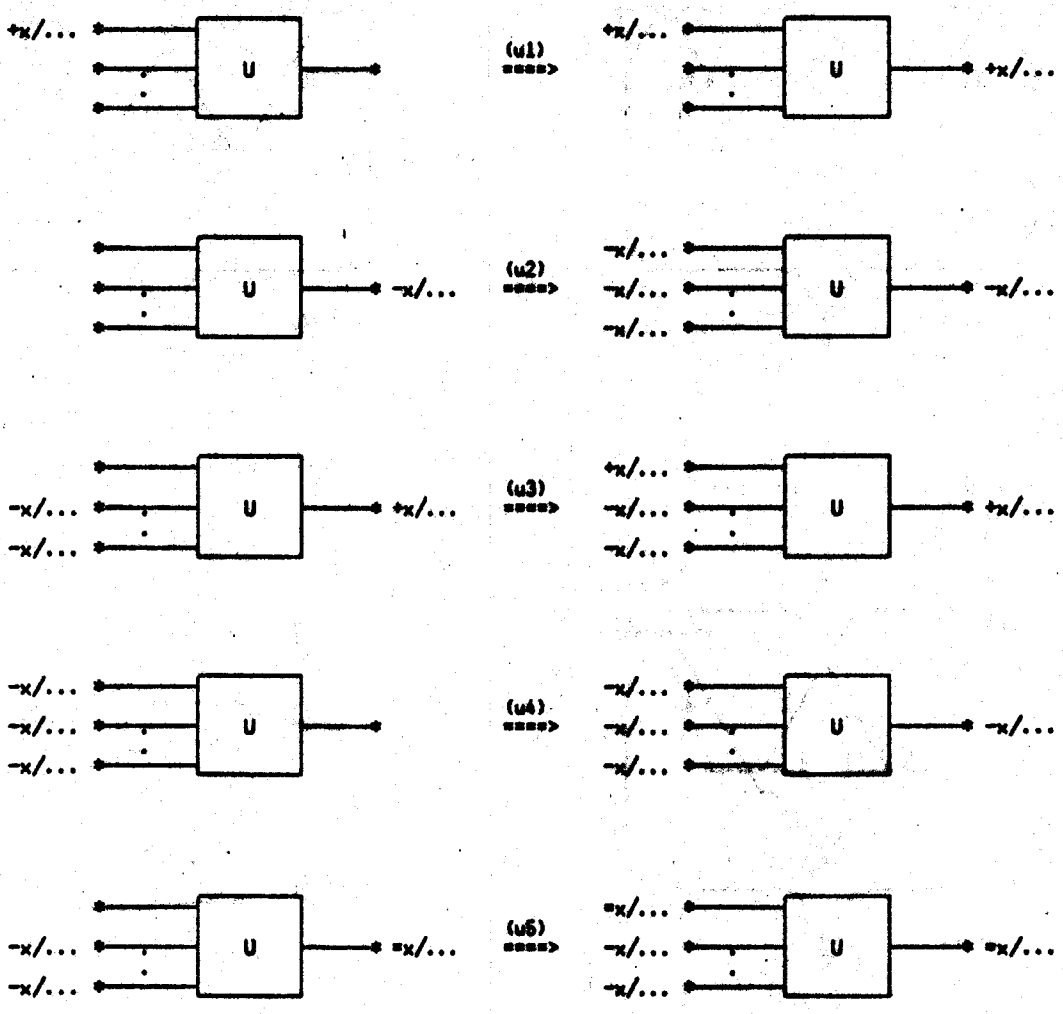
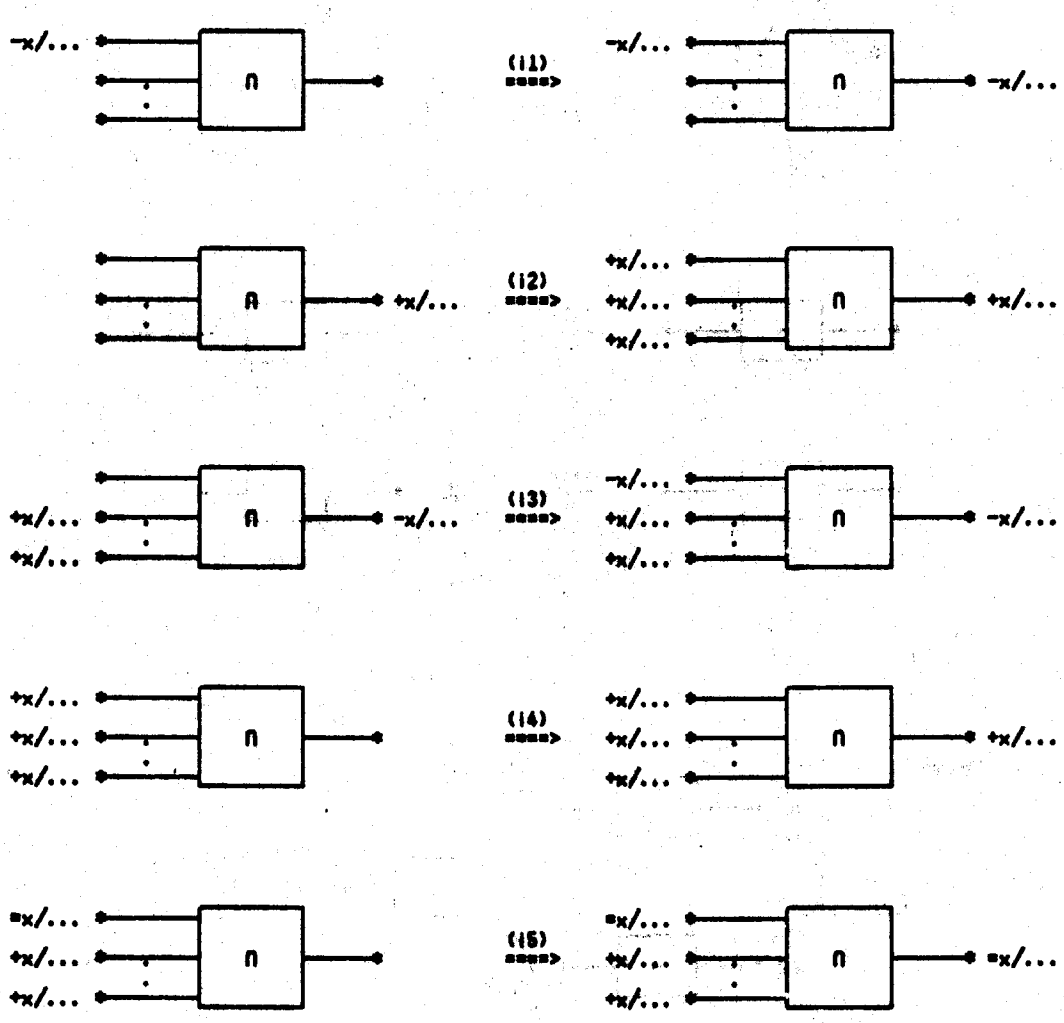


Figure 3-3 — Rules for the Union Constraint



Figire 3-4 — Rules for the Intersection Constraint

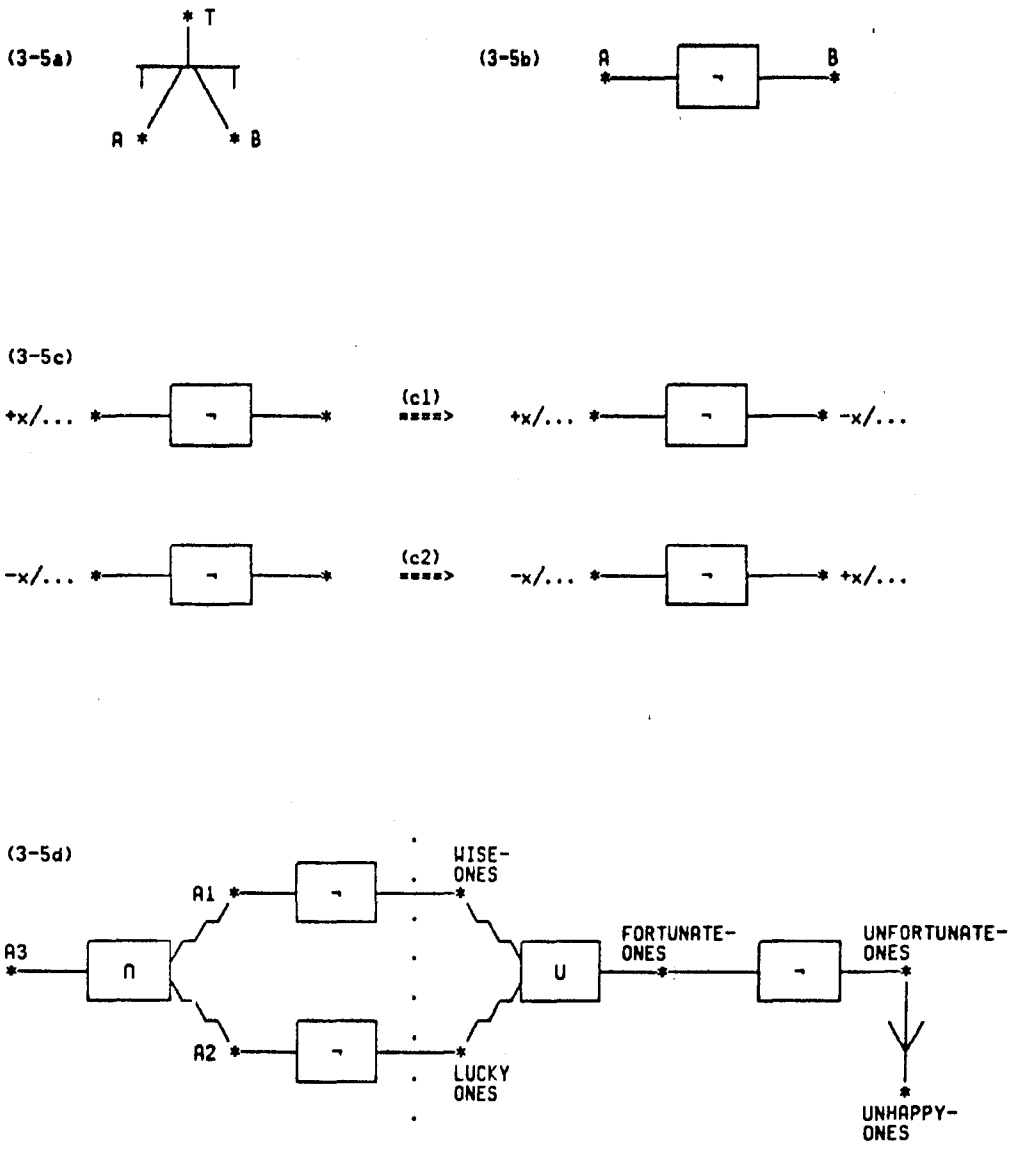
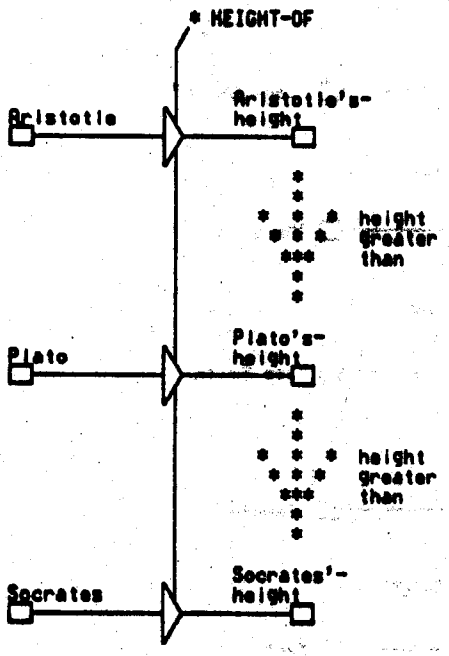
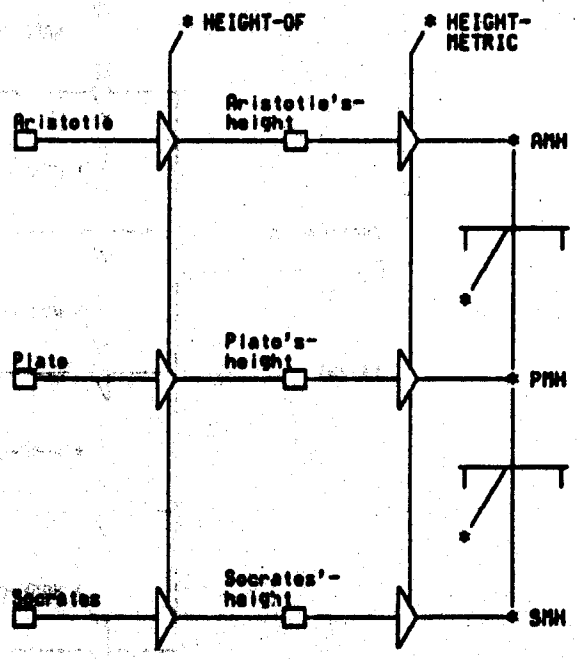


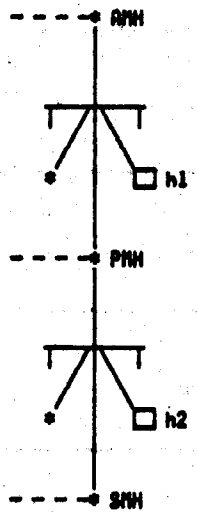
Figure 3-5 — The Complement Constraint



(3-6a)

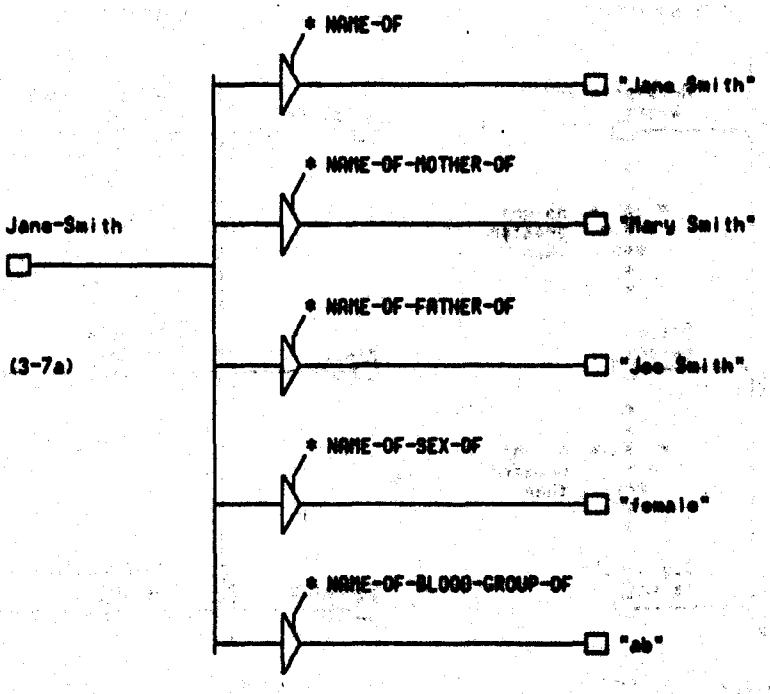


(3-6b)



(3-6c)

Figure 3-6 — A Transitive Relation



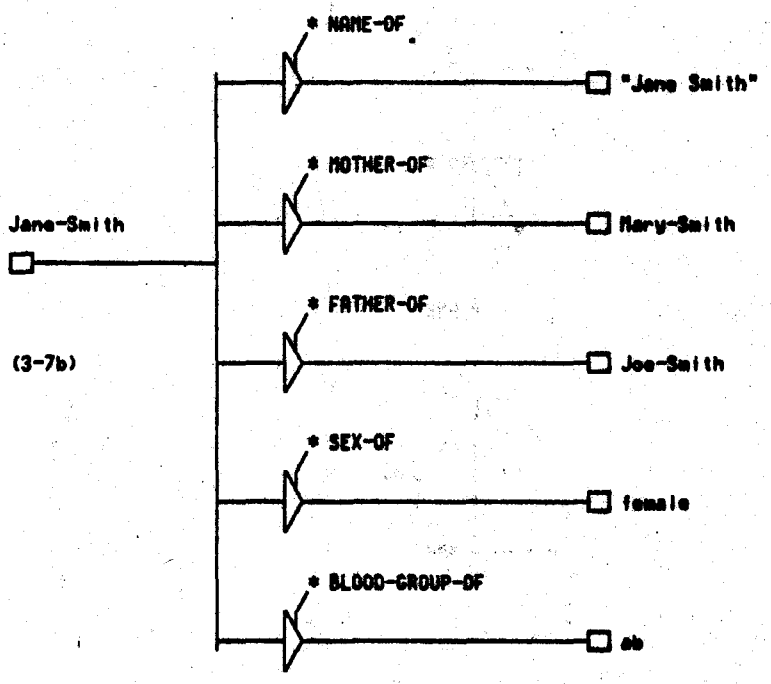
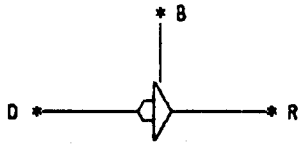
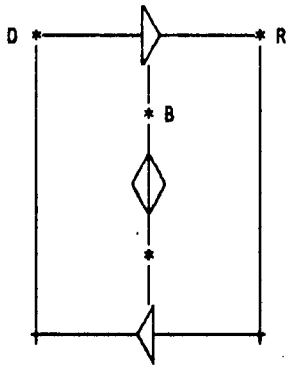


Figure 3-7 — An N-ary Relation

(3-8a)



(3-8b)



(3-8c)

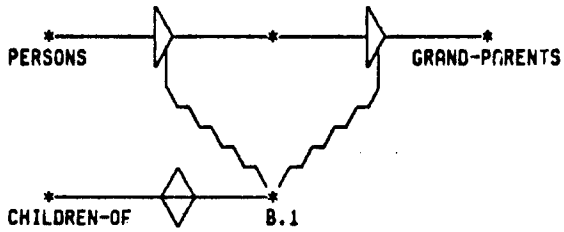
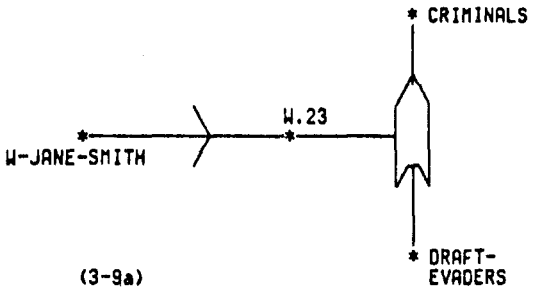
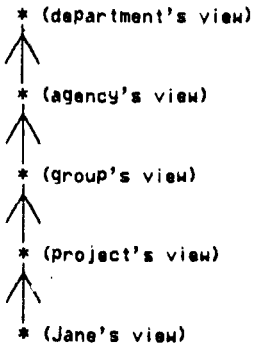


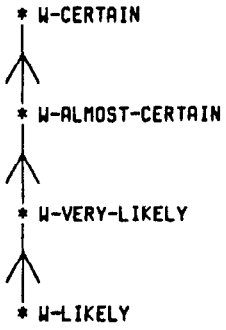
Figure 3-8 — Using the Inverse Constraint



(3-9a)



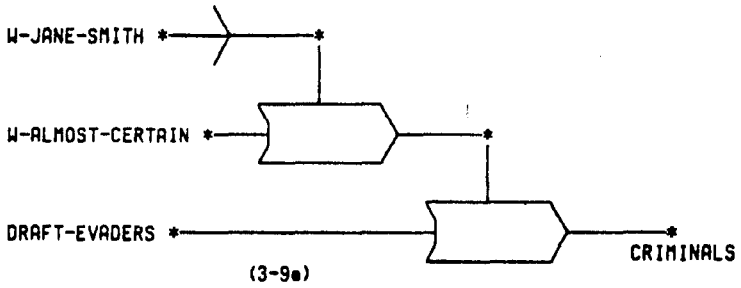
(3-9b)



(3-9c)

inf —— w-certain —— w-almost-certain —— w-very-likely —— w-likely

(3-9d)



(3-9e)

Figure 3-9 (3-9a - 3-9e) — Using the World Constraint

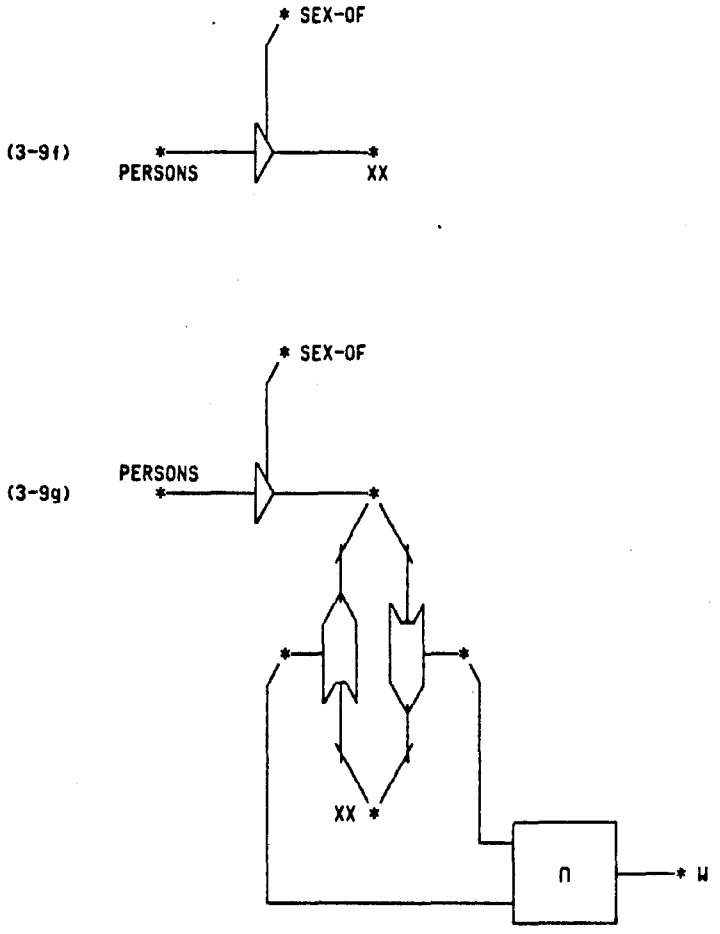
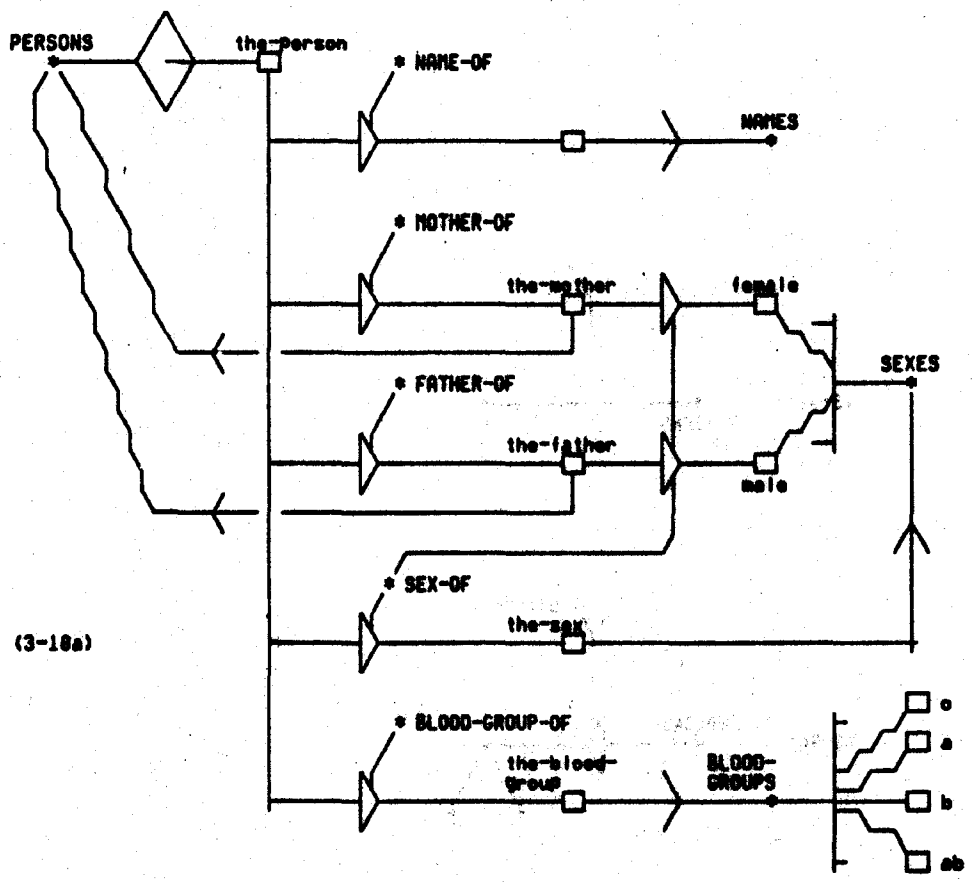


Figure 3-9 (3-9f - 3-9g) — Using the World Constraint



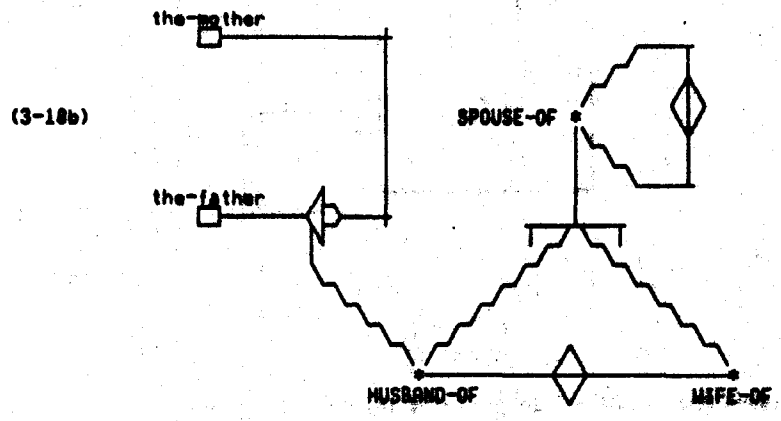


Figure 3-18 — A Template

Figure 5-1 — An example of Incompleteness

Linear notation: (A (B C) D)

Network notation:

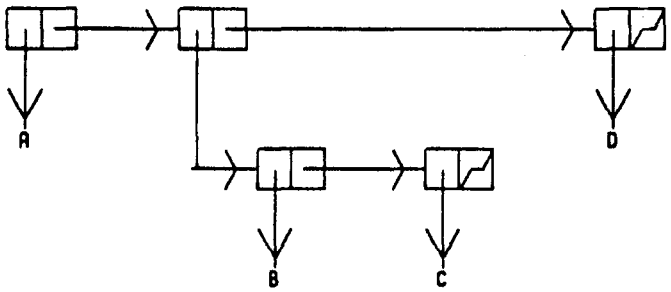


Figure 6-1 — Linear and Network notations for LISP

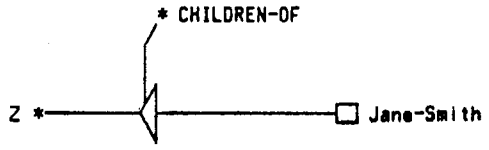


Figure A-1 — Constructing the "Z" class

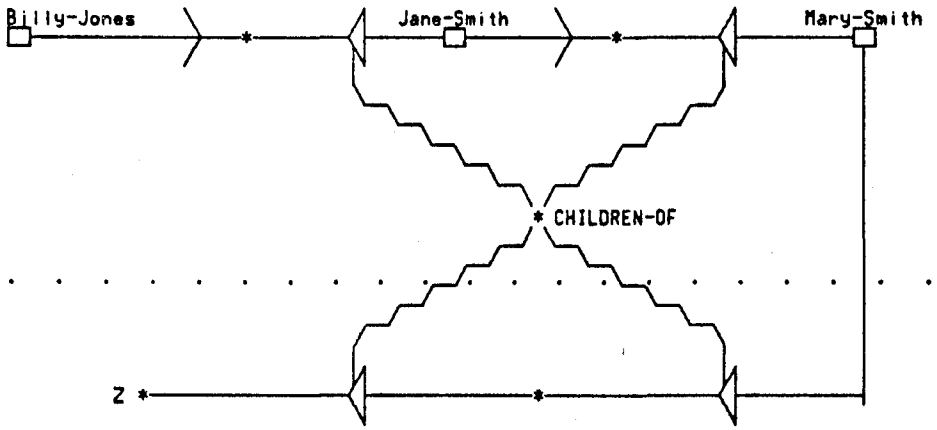
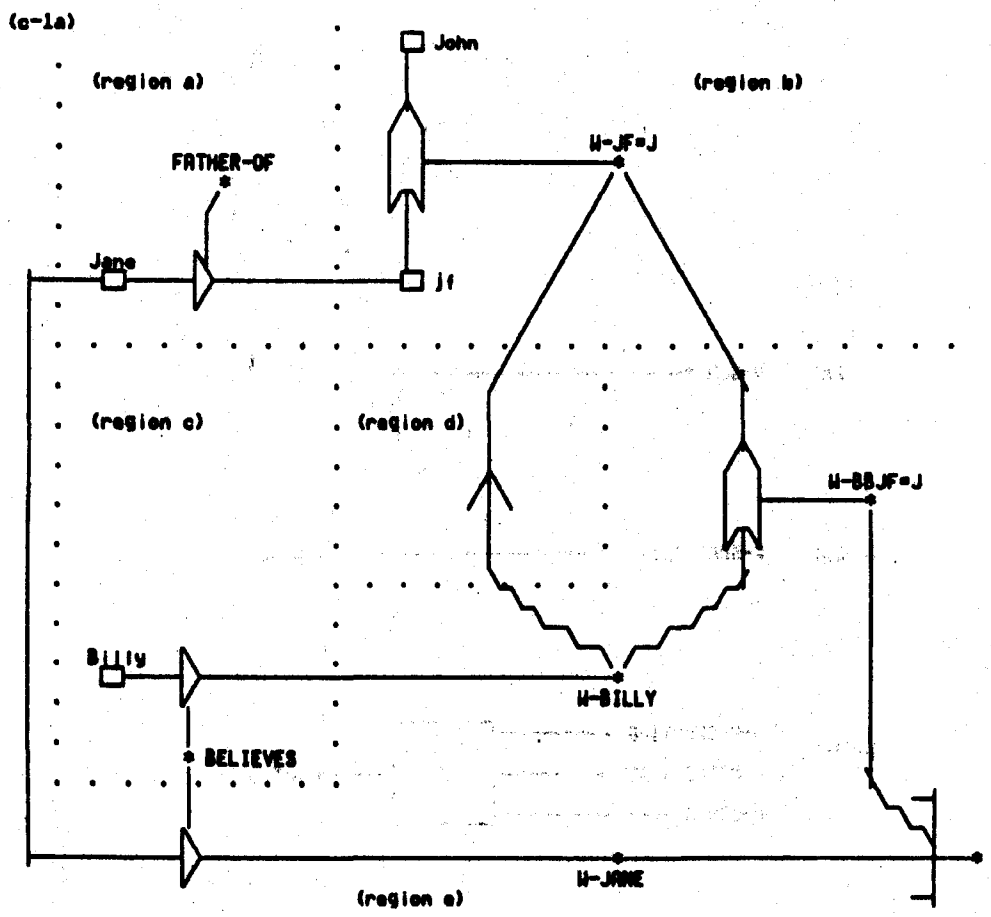
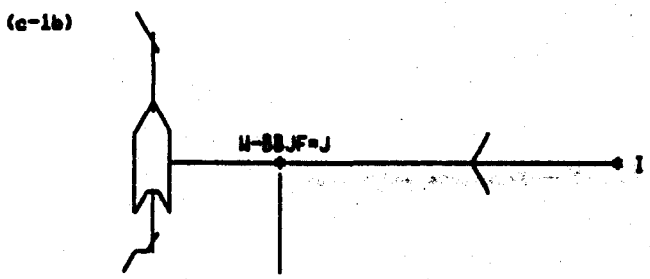


Figure A-2 — Example for Reflection-finding





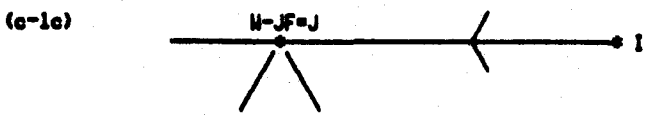


Figure C-1 — Beliefs

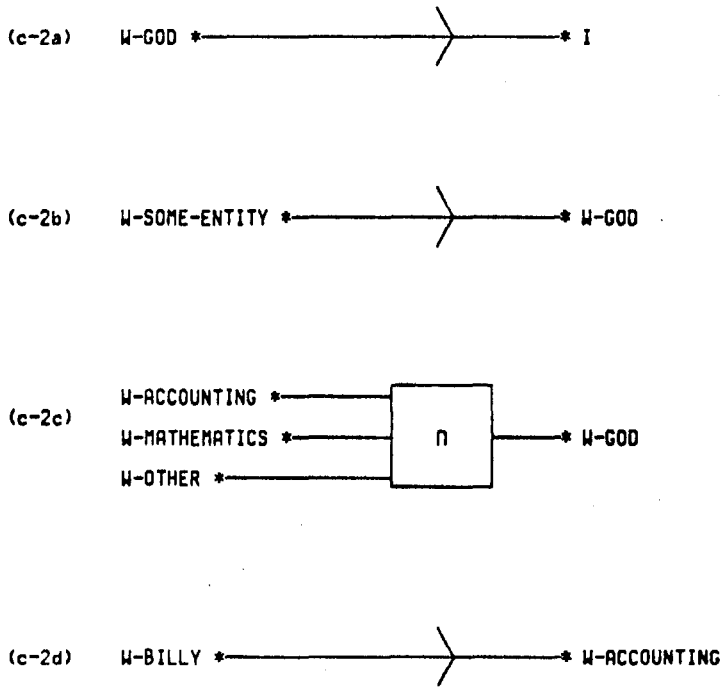


Figure C-2 — Knowledge and Wisdom

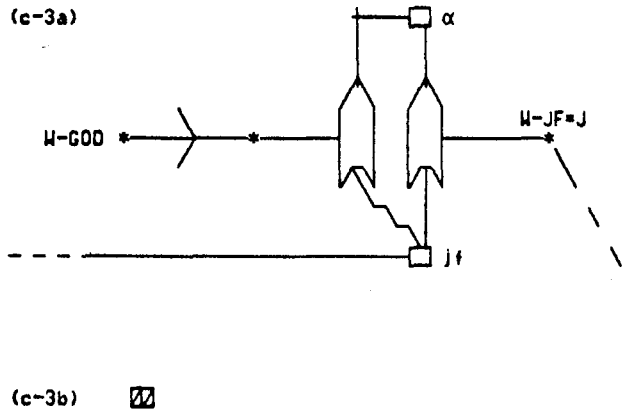


Figure C-3 — The Interworld Object

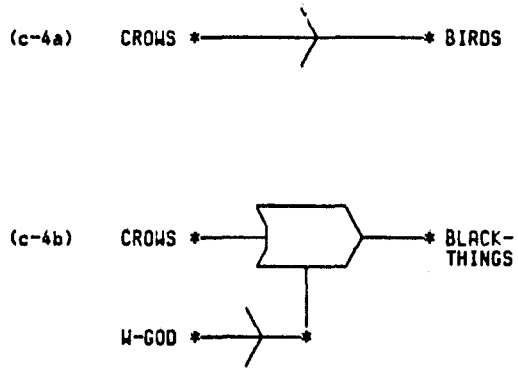


Figure C-4 — "Necessary" and "Contingent" Truths

Indices

This index notes where some of the technical terms are defined.

This index notes where the propagation rules are defined.

CS-TR Scanning Project
Document Control Form

Date : 12/11/95

Report # LCS-TR-158

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 156 (163-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form (2) ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☒ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:
IMAGE MAP: (1-156) UNNUMBERED TITLE PAGE, 2-156
(157-163) SCANCONTROL, PRINTER'S NOTES, DOD(2), TRGT'S (3)

Scanning Agent Signoff:
Date Received: 12/11/95 Date Scanned: 1/17/96 Date Returned: 1/18/96

Scanning Agent Signature: Michael W. Cook

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: TR-158	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: Some Data-base Applications of Constraint Expressions	5. TYPE OF REPORT & PERIOD COVERED: S.M. Thesis, January 1976
6. PERFORMING ORG. REPORT NUMBER: TR-158	
7. AUTHOR(s): Richard W. Grossman	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0661 N00014-75-C-0643
9. PERFORMING ORGANIZATION NAME AND ADDRESS: MASSACHUSETTS INSTITUTE OF TECHNOLOGY LABORATORY FOR COMPUTER SCIENCE (formerly Project MAC) 545 Technology Square Cambridge, Massachusetts 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research Department of the Navy Information Systems Program Arlington, Virginia 22217	12. REPORT DATE: February 1976 13. NUMBER OF PAGES: 157
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*: Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report presents a novel network-like representation for information, called "constraint expressions" (CE). CE makes use of some of the knowledge-representation techniques developed by Artificial Intelligence research. A CE network consists of points (which represent classes of objects) interconnected by constraints (which represent the relationship which are known to hold among the classes). All constraints are defined in terms of six primitive ones. The data in a CE network is accessed by propagating various

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

20. kinds of labels through it. Each constraint can be viewed as an active process which looks to other points when such patterns occur.

The CE representation provides several significant features which are not found in most current data models. First, the same mechanism is used to represent "general" as well as "specific" information. For example, "The sex of Jane Smith is female" is specific, while "Every person has a unique sex which is either 'male' or 'female' is general.

Second, CE's label-propagation procedure implements logical consistency checking: Data-base integrity can be maintained by checking all new data for consistency with the existing information. Since the data-base can contain general information (representing a "semantic model" of the data-base's application domain), new specific data can be rejected if it is inconsistent with either other specific data or with the general information. Also, the general information can itself be checked for internal consistency.

Third, the CE representation is sufficiently modular and well-defined so that it has a precise formal semantics, which insures that CE's definition contains no hidden ambiguities or contradictions.

Fourth, CE's modularity allows the label propagations to be done in parallel, so that parallel hardware can be used to full advantage.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**