

**A Specialized ATMS for Equivalence Relations**

By

Caroline Nan Koff

A THESIS

submitted to

**Oregon State University**

in partial fulfillment of  
the requirements for the degree of

**Master of Science**

Completed May 12, 1988

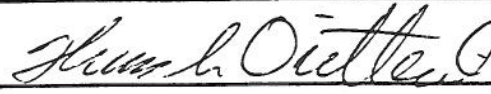
Commencement June 1989

## AN ABSTRACT OF THE THESIS OF

Caroline N. Koff for the degree of Master of Science in Computer Science  
presented on May 12, 1988.

Title: A Specialized ATMS for Equivalence Relations

Abstract approved:



Thomas G. Dietterich.

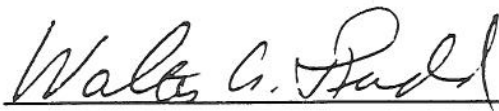
A specialized ATMS for efficiently computing equivalence relations in multiple contexts is introduced. This specialized ATMS overcomes the problems with existing solutions to reasoning with equivalence relations. The most direct implementation of an equivalence relation in the ATMS—encoding the reflexive, transitive, and symmetric rules in the consumer architecture—produces redundant equality derivations and requires  $\Theta(n^3)$  label update attempts (where  $n$  is the number of terms in the equivalence class). An alternative implementation is one that employs simple equivalence classes. However, this solution is unacceptable, since the number of classes grows exponentially with the number of distinct assumptions. The specialized ATMS presented here produces no redundant equality derivations, requires only  $\Theta(n^2)$  label update attempts, and is most efficient when there are many distinct assumptions and few nogoods. This is achieved by exploiting a special relationship that holds among the labels of the equality assertions because of transitivity. The standard dependency structure construction and traversal is replaced by a single pass over each label in a particular kind of equivalence class. The specialized ATMS has been implemented as part of the logic programming language FORLOG.

APPROVED:



---

Professor of Computer Science in charge of major



---

Head of Department of Computer Science

---

Dean of Graduate School

Date thesis presented May 12, 1988

Typed by Caroline N. Koff for Caroline N. Koff

## ACKNOWLEDGEMENTS

I would like to thank my advisor, Tom Dietterich, for his excellent guidance throughout this research, and for lending me many of his insights in AI. Working for and with him has given me an invaluable research experience.

Nick Flann, a fellow student and a friend, has also guided this research closely along with my advisor. I thank him for his patience, especially in teaching me FORLOG, and for his support and feedback.

The bulk of the ideas of this thesis was first presented in a technical report (written by Nick Flann, Tom Dietterich and myself), and therefore, both Nick Flann and Tom Dietterich have made significant contributions to the overall development of this thesis. Nick Flann is largely responsible for the final construction of the proof of the label-update algorithm (presented in Section 2.3.5). Tom Dietterich is responsible for the overall outline of this thesis, and for formulating the ideas on introducing the problem, the existing approaches to the problem, the computational cost of de Kleer's label-update algorithm (all presented in Chapter 1), and on the explanation of my label-update algorithm (Section 2.3.3).

I would like to thank Jim Holloway, Giuseppe Cerbone, and Marion Hakan-son, for their helpful comments on the earlier drafts of this thesis.

A special thanks to Ritchey Ruff for his comments on my thesis and for helping me with the use of equipment.

This research was supported by the National Science Foundation under grants DMC-85-14949 and IRI-86-57316 and by Tektronix, Inc. under contract No. 530097.

I am also thankful to my family for their emotional support and encouragement, and especially to Pravin P. Kamdar, my husband and best friend. He has helped me to state my ideas in writing and has patiently proof read numerous drafts

of this thesis. Discussions with him have allowed me better understand my label-update algorithm. He has constantly supported and encouraged me throughout my undergraduate and graduate years, and I wish to dedicate this thesis to him.

## Table of Contents

|       |                                                      |    |
|-------|------------------------------------------------------|----|
| 1     | Introduction                                         | 1  |
| 1.1   | Simple Equality Reasoning . . . . .                  | 1  |
| 1.2   | Equality Reasoning Under Multiple Contexts . . . . . | 2  |
| 1.3   | Existing Approaches . . . . .                        | 4  |
| 1.3.1 | Extending UNION-FIND . . . . .                       | 4  |
| 1.3.2 | Employing de Kleer's ATMS . . . . .                  | 5  |
| 2     | A Specialized ATMS for Equivalence Relations         | 11 |
| 2.1   | The Equality Database . . . . .                      | 11 |
| 2.2   | The Problem Solver . . . . .                         | 12 |
| 2.3   | The Label-Update Algorithm . . . . .                 | 15 |
| 2.3.1 | The Algorithm . . . . .                              | 15 |
| 2.3.2 | An Example . . . . .                                 | 15 |
| 2.3.3 | An Explanation . . . . .                             | 17 |
| 2.3.4 | Computational Costs . . . . .                        | 18 |
| 2.3.5 | Proof of Correctness . . . . .                       | 19 |
| 3     | Extending the Method                                 | 22 |
| 3.1   | Equality Tokens . . . . .                            | 22 |
| 3.2   | Optimization . . . . .                               | 23 |
| 3.3   | The Revised Label-Update Algorithm . . . . .         | 24 |
| 3.4   | Removing Inconsistencies from the Labels . . . . .   | 24 |
| 3.4.1 | Removing Primitive Nogood Environments . . . . .     | 26 |

|       |                                                               |    |
|-------|---------------------------------------------------------------|----|
| 3.4.2 | Removing Derived Nogood Environments . . . . .                | 27 |
| 3.4.3 | A Concluding Remark on Removing Inconsistencies . . .         | 30 |
| 4     | Implementation Issues                                         | 31 |
| 4.1   | The Equality Database Data Structure . . . . .                | 31 |
| 4.1.1 | The Equality Node Data Structure . . . . .                    | 33 |
| 4.1.2 | The Equivalence Class Node Data Structure . . . . .           | 34 |
| 4.2   | Interfacing the Specialized ATMS with a de Kleer-style ATMS . | 34 |
| 5     | Summary                                                       | 35 |
| 5.1   | Advantages and Disadvantages . . . . .                        | 35 |
| 5.2   | Future Research Issues . . . . .                              | 36 |
|       | Bibliography                                                  | 37 |
|       | Appendix                                                      | 38 |
| A     | Glossary of Terms                                             | 38 |

## List of Figures

|     |                                                                                          |    |
|-----|------------------------------------------------------------------------------------------|----|
| 1.1 | The dependency structure for the three equality assertions . . .                         | 7  |
| 2.1 | The equivalence class of Figure 1.1 . . . . .                                            | 12 |
| 2.2 | An equivalence class with a contradictory equality . . . . .                             | 14 |
| 2.3 | Before the label updates . . . . .                                                       | 16 |
| 2.4 | After the label updates . . . . .                                                        | 16 |
| 2.5 | A partial equivalence class with a shared support . . . . .                              | 18 |
| 2.6 | Incremental extension of an equivalence class . . . . .                                  | 20 |
| 3.1 | The revised label-update algorithm . . . . .                                             | 25 |
| 3.2 | An equivalence class with the nogood environments encircled . .                          | 27 |
| 3.3 | An equivalence class with the nogood environment encircled . .                           | 28 |
| 3.4 | An equivalence class of Figure 3.3 with the derived nogood environment removed . . . . . | 29 |
| 3.5 | An equivalence class of Figure 3.4 with a new support . . . . .                          | 29 |
| 4.1 | A partial data structure for the equivalence class in Figure 2.3 .                       | 32 |
| 4.2 | A complete data structure for the equivalence class in Figure 2.3                        | 33 |



## List of Tables

|     |                                               |    |
|-----|-----------------------------------------------|----|
| 2.1 | Summary of the label-update process . . . . . | 16 |
|-----|-----------------------------------------------|----|

# A Specialized ATMS for Equivalence Relations

## Chapter 1

### Introduction

#### 1.1 Simple Equality Reasoning

Consider the following reasoning problem. Given equality assertions of the form  $x = y$ , where  $x$  and  $y$  are either Skolem constants<sup>1</sup> or ordinary constants, compute the symmetric and transitive closure of the equality relation, detect contradictions among the assertions, and answer queries of the form “Is the equality  $x = y$  true?”. This problem has a long history in computer science, beginning with the need to implement the EQUIVALENCE and COMMON declarations for a FORTRAN compiler [Arden, Galler, & Graham, 1961]. An EQUIVALENCE or a COMMON declaration in a FORTRAN program requires the compiler to form an equivalence class among the variables given in the declaration. The best known solution to this reasoning problem involves representing equivalence classes (sets of constants known to be equal to one another) as trees spanning from a chosen variable (the class representative) to the other members of the class by applying the UNION-FIND algorithm [Galler & Fisher, 1964]. Then, to reduce the cost of the FIND operation, the path compression algorithm is applied [Aho, Hopcroft, & Ullman, 1974].

---

<sup>1</sup>See Appendix A for definitions of all technical terms appearing in this thesis.

## 1.2 Equality Reasoning Under Multiple Contexts

In this thesis, we are interested in solving the more general problem of reasoning with equality under multiple contexts. Informally, this problem is analogous to compiling several FORTRAN programs simultaneously when these programs share some EQUIVALENCE statements but not others. Each program corresponds to a context, and the goal is to record all of the EQUIVALENCE statements in a single database while keeping track of which statements belong together.

To define the problem formally, we must first introduce the concepts of assumptions, environments, and contexts. In any problem solving system, there are two kinds of facts that must be recorded: premises and derived facts. Premises are given to the system from some external source, while derived facts are computed by the system from the premises. For example, suppose an equality reasoning system is given the premises  $x = 1$  and  $y = x$ . By applying the axioms of equality, the system might then infer that  $y = 1$ .

In the multiple-context reasoning problem, some of the premises can be declared to be *assumptions*. Each assumption is assigned a unique atomic symbol (usually an upper-case letter). For example, the premise  $x = 1$  could be declared to be an assumption and assigned the symbol  $A$ . Similarly, the premise  $y = x$  could be declared to be an assumption and assigned the symbol  $B$ . The problem of reasoning under multiple contexts is to maintain, for each derived fact, information concerning the assumptions that it depends upon. In this case, the derived fact  $y = 1$  depends on assumptions  $A$  and  $B$ . In general, when a new fact is derived, the assumptions that it depends upon ( $\{A, B\}$ ) can be computed simply by taking the set union of the assumptions of the facts from which it was derived ( $\{A\} \cup \{B\}$ ).

Assumptions can be used to investigate alternative sets of beliefs. For example, we can introduce a premise such as  $x = 2$  and declare it as assumption  $C$ . Now the reasoning system will infer that  $y = 2$  under assumptions  $B$  and  $C$ . The reasoning system will also notice that assumptions  $A$  and  $C$  are mutually contradictory, because  $x$  cannot simultaneously be equal to 1 and to 2.

An *environment* is a set of assumption symbols, such as  $\{A, B\}$ . Another way of describing the state of the reasoning system at this point would be to say that the fact  $x = 1$  is true in environment  $\{A\}$ ,  $y = x$  is true in environment  $\{B\}$ ,  $y = 1$  is true in environment  $\{A, B\}$ , and so on. The environment  $\{A, C\}$  is said to be a *nogood* environment, because it is contradictory.

A *context* is the set of all facts that are believed in a given environment. In the environment  $\{A, B\}$ , for example, the facts believed include  $x = 1$ ,  $y = x$ , and  $y = 1$ . In the environment  $\{B, C\}$ , the facts believed include  $x = 2$ ,  $y = x$ , and  $y = 2$ . Notice that these contexts include all facts believed in any subset of the environment.

In general, it is possible for a fact to be believed in more than one environment. To continue our example, suppose that the reasoning system is given the assumptions  $y = z$  (assumption  $D$ ) and  $z = 1$  (assumption  $E$ ). Then the fact  $y = 1$  is believed in two separate environments,  $\{A, B\}$  and  $\{D, E\}$ . To store this information, each fact in the reasoning system is assigned a label. The label is a *set* of environments (i.e., a set of sets of assumption symbols). Hence, the fact  $y = 1$  would be assigned the label  $\{\{A, B\}, \{D, E\}\}$ .

With this introduction, the multiple-context reasoning problem can now be defined formally. Given equality assertions of the form  $x = y$ , where  $x$  and  $y$  are either Skolem constants or ordinary constants and where each assertion has an attached label, compute the symmetric and transitive closure of the equality relation, detect contradictions among the assertions, and answer queries of the form "Is  $x = y$  true in environment  $\{A_1, A_2, \dots, A_k\}$ ?"

Notice that if all equality assertions are assigned the same label, for example,  $\{\{A\}\}$ , then all derived facts will also have this label, and the multiple-context reasoning problem reduces to the single-context reasoning problem that we described at the start of this chapter. We will see below that the UNION-FIND algorithm implicitly assumes that all assertions exist in a single context, and therefore, the algorithm will not provide a solution to the multiple-context reasoning problem.

The multiple-context reasoning problem arises in any situation where equal-

ity assertions are present and there is a need to investigate multiple contexts simultaneously. In particular, it arises in the FORLOG logic programming system [Flann, Dietterich & Corpron, 1987]. FORLOG is a forward-chaining logic programming language that employs Skolem constants in place of Prolog's "logical variables" and performs equality reasoning instead of unification. It is implemented using an extended version of de Kleer's [1986c] consumer architecture. We expect that the same problem will arise in any parallel logic programming system. Let us explore two forward chaining approaches to solving this equality reasoning problem.

### 1.3 Existing Approaches

There are two obvious methods for reasoning with equality in multiple contexts: (a) employing a multiple-context version of the UNION-FIND algorithm and (b) employing de Kleer's ATMS.

#### 1.3.1 Extending UNION-FIND

The first approach is to employ some kind of an equivalence class tree data structure by applying the UNION-FIND algorithm, where an equivalence class is a set of constants and Skolem constants that are all equal to one another in a single context. In single-context systems (like Prolog and RUP [McAllester, 1982]), the context in question is implicit, and this is very efficient. However, when we move to multiple contexts, the number of equivalence classes explodes. For example, suppose we have the following three equality assertions constructed from four Skolem constants,  $a, u, v$ , and  $w$ :

$$\langle a = u, \{A\} \rangle, \langle a = v, \{B\} \rangle, \langle a = w, \{C\} \rangle.$$

In this case, seven equivalence classes must be constructed:

- $\{a, u\} \{A\}$ ,
- $\{a, v\} \{B\}$ ,

- $\{a, w\} \{C\}$ ,
- $\{a, u, v\} \{A, B\}$ ,
- $\{a, u, w\} \{A, C\}$ ,
- $\{a, v, w\} \{B, C\}$ ,
- $\{a, u, v, w\} \{A, B, C\}$ .

If only the last class were constructed, it would not be possible to answer the query “Is  $u = v$  true in  $\{A, B\}$ ?” correctly. So, every distinct context must have its own equivalence class. Since there are  $2^k - 1$  contexts for  $k$  assumptions, this results in an exponential explosion, and hence it is an unacceptable solution.

### 1.3.2 Employing de Kleer’s ATMS

The exponential explosion is not unique to the problem of equality reasoning under multiple contexts. It exists in general when reasoning under multiple contexts.

De Kleer has designed an efficient solution to this problem by constructing an architecture called the ATMS (assumption based truth maintenance system) [de Kleer, 1986a]. In the ATMS, each fact is stored in a data structure called an *ATMS node*. This data structure contains the fact itself (e.g.,  $x = 1$ ), which is called the *datum*. It also contains the label for the fact (e.g.,  $\{\{A\}\}$ ), which is computed as described above. Finally, it contains a set of *justifications* that record how that fact was computed. We will employ a tuple notation of the form  $nodeid : \langle datum, label, justifications \rangle$  for each ATMS node.

For example, given the premises  $x = 1$  (assumption  $A$ ) and  $y = x$  (assumption  $B$ ), the ATMS would create three ATMS nodes:  $node1 : \langle x = 1, \{\{A\}\}, \{\} \rangle$ ,  $node2 : \langle y = x, \{\{B\}\}, \{\} \rangle$ , and  $node3 : \langle y = 1, \{\{A, B\}\}, \{(node1, node2)\} \rangle$ . Notice that each justification is itself a list of ATMS nodes that, taken together, are sufficient to derive this node.

The solution (to the problem of equality reasoning under multiple contexts) employing de Kleer’s ATMS will consist of two parts: (a) the ATMS-based problem

solver, which will be responsible for computing the symmetric and transitive closure of the equality assertions (plus detecting contradictions), and (b) the ATMS database, which will efficiently represent the equality facts and update their labels by applying de Kleer's label-update algorithm.

### Encoding the Equality Axioms

Computing the symmetric and transitive closure of the equalities can be easily accomplished by directly encoding the equality axioms to the ATMS-based problem solver. Among the three equality axioms (reflexive, symmetric, and transitive), only the transitive axiom need be represented directly. The reflexive axiom,  $x = x$ , can be handled by the query routines, and the symmetry axiom,  $x = y \supset y = x$ , can be handled by establishing a canonical ordering over the terms, and doing some clever pattern matching on the left-hand-side of the transitivity axiom:

$$\forall x, y, z \quad x = y \wedge y = z \supset x = z. \quad (1.1)$$

Here  $x$ ,  $y$ , and  $z$  are either Skolem constants or ordinary constants. Whenever the antecedent pattern of this axiom is satisfied by a set of facts in the ATMS database during problem solving, a new equality assertion is derived and added to the ATMS database as an ATMS node. For example, consider the following two equality assertions, where  $sk1$ ,  $sk2$ , and  $sk3$  are Skolem constants:

$$node1 : \langle sk1 = sk2, \{\{A\}\}, \{\} \rangle, \quad (1.2)$$

$$node2 : \langle sk2 = sk3, \{\{B\}\}, \{\} \rangle. \quad (1.3)$$

Here, the equality fact  $sk1 = sk2$  of (1.2) is true in environment  $\{A\}$ , and has no justifications because it is an assumption. Similarly for (1.3). The ATMS database consisting of these two facts will allow the problem solver to satisfy the antecedents of (1.1) and will produce the following *derived* ATMS node:

$$node3 : \langle sk1 = sk3, \{\{A, B\}\}, \{(node1, node2)\} \rangle. \quad (1.4)$$

Here, the fact  $sk1 = sk3$  of (1.4) is true in environment  $\{A, B\}$ , and is justified by the two ATMS nodes from which it was derived.

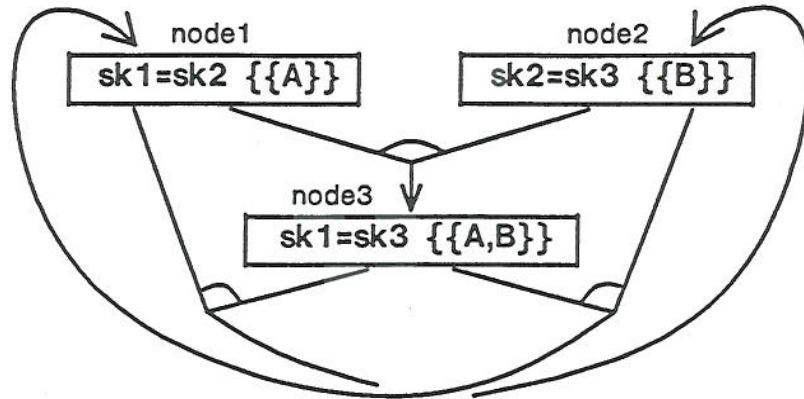


Figure 1.1: The dependency structure for the three equality assertions

Obviously, the equality fact in *node3* is the only new equality information derivable from the equality facts in *node1* and *node2*. But by applying the symmetry axiom, the newly derived equality in *node3* will twice satisfy the antecedent of (1.1) in conjunction with the equalities in *node1* and *node2* respectively, and rederive the following two equalities:

$$node1 : \langle sk1 = sk2, \{\{A, B\}\}, \{(node2, node3)\} \rangle, \quad (1.5)$$

$$node2 : \langle sk2 = sk3, \{\{A, B\}\}, \{(node1, node3)\} \rangle. \quad (1.6)$$

The equality assertions of (1.5) and (1.6) will result in attempting to add the newly derived environment,  $\{A, B\}$ , to the existing labels of *node1* and *node2*. However, one of the requirements imposed on all labels by the ATMS is that they be kept in minimal form (i.e., in most general form) by removing all subsumed environments from a label. Therefore, since the newly derived environment,  $\{A, B\}$ , is subsumed by the existing environments  $\{A\}$  of *node1* and  $\{B\}$  of *node2*, it is not added to the labels of the nodes *node1* and *node2*. The final resulting dependency structure constructed from these three ATMS nodes are shown in Figure 1.1. (Justifications for each equality assertion are shown as two links merging to support that assertion.) Since the newly derived environment,  $\{A, B\}$ , was not added to the labels of the nodes *node1* and *node2*, the equality derivations of (1.5) and (1.6) were redundant. However these two redundant derivations allowed the problem solver to



generate additional justifications: the justification that *node2* and *node3* together can derive *node1*, and the justification that *node1* and *node3* together can derive *node2*. We will see next how these justifications become necessary for the ATMS to correctly update the labels when an existing node label is given a new supporting environment.

### De Kleer's Label-Update Algorithm

When an ATMS node label is given a new supporting environment, a label update process is performed to propagate this new environment to the rest of the ATMS nodes' labels. De Kleer's label-update algorithm [de Kleer, 1986a] performs this process by recursively updating the consequent node labels (the nodes justified by the node which received the new supporting environment) by traversing the justification links. Although the algorithm guarantees that the labels will be consistent and complete upon termination of the algorithm, each node's label may have been updated more than once. By applying this algorithm to a collection of mutually-supporting nodes, such as those shown in Figure 1.1, an alarming number of label update attempts will occur due to the circular structure of the justification links. For example, consider the following series of label update attempts made by the algorithm after *node1* has been updated to include the new environment  $\{C\}$  in its label as a new support. First, the algorithm attempts to update the labels of *node1*'s consequent nodes, *node2* and *node3*:

- For *node2*'s label, the new environment of *node1*, namely  $\{C\}$ , and the environment of *node3*, namely  $\{A, B\}$ , are combined by taking the union to produce the new environment  $\{A, B, C\}$ , which is subsumed by  $\{B\}$  and hence not added to *node2*'s label.
- For *node3*'s label, the new environment of *node1*, namely  $\{C\}$ , and the environment of *node2*, namely  $\{B\}$ , are combined to produce the new environment  $\{B, C\}$  which is included in *node3*'s label.

Since *node3*'s label has changed (i.e., actually updated), the algorithm will now recur and attempt to update the labels of *node3*'s consequent nodes, *node1* and *node2*:

- For *node1*'s label, the new environment of *node3*, namely  $\{B, C\}$ , and the environment of *node2*, namely  $\{B\}$ , are combined to produce the new environment  $\{B, C\}$ , which is subsumed by  $\{C\}$  and hence not added to *node1*'s label.
- For *node2*'s label, the new environment of *node3*, namely  $\{B, C\}$ , and the environments of *node1*, namely  $\{A\}$ , and  $\{C\}$ , are combined to produce the new environments  $\{A, B, C\}$  and  $\{B, C\}$  both of which are subsumed by  $\{B\}$  and hence not added to *node2*'s label.

In this example, the label-update algorithm attempted to update four labels, out of which three resulted in computing redundant environments.

The example given above does not demonstrate the worst case. The worst case occurs when new support arrives on a derived ATMS node, such as *node3*—the algorithm must traverse every justification of every node. Since there are  $\binom{n}{2}$ , or  $\frac{n(n-1)}{2}$  equalities, where  $n$  is the number of terms in an equivalence class, and there are  $n - 2$  ways to justify an equality, the number of label update attempts made by the algorithm is  $\frac{n(n-1)(n-2)}{2}$  or  $\Theta(n^3)$ . The best case occurs when the algorithm terminates after attempting to update just one label upon either deriving a *nogood* (an environment which supports a contradictory fact), or deriving an environment which was subsumed by the node's original label.

One approach to reducing the generation of redundant equality assertions is to employ typed consumers. The basic idea is to postpone construction of the circular dependency links until they are needed to allow label propagation and updating. The example used by de Kleer [1986c] is the relation *plus*( $x, y, z$ ). Such relations are implemented by a set of constraint consumers, one for each variable that computes its value from the values of the other variables. For example, when  $x$  and  $y$  are known, a constraint consumer computes the value for  $z$ . However, this

value for  $z$  will be used with  $x$  (or  $y$ ) and another constraint consumer to recompute  $y$  (or  $x$ ). To avoid such redundancies, a special mechanism was proposed by de Kleer that involved assigning a unique *type* to each constraint consumer of a relation and barring the use of data derived from such consumers to satisfy other consumers of the same type. This prevents the circular justifications and redundant assertions from being created until additional support is given to the value for  $z$ . At that point, the justifications will be created so that this new support can be propagated to  $x$  and to  $y$ .

Because redundant assertions and circular justifications are eventually created, typed consumers do not improve the worst-case behavior of this approach to equality reasoning.

## Chapter 2

### A Specialized ATMS for Equivalence Relations

Both of the approaches given above for implementing equality reasoning under multiple contexts are inefficient either because they use the implicit justification structure of equivalence classes or because they construct explicit justification links. For the remainder of this thesis, a new solution to the problem of equality reasoning under multiple contexts is described. The new solution avoids both of the problems stated above by using a weaker kind of equivalence class and exploiting special properties of the ATMS labels. It does not construct any explicit justification links. There are three components to this specialized ATMS: the equality database (hereafter, ED), the problem solver, and the label-update algorithm.

#### 2.1 The Equality Database

The equality database consists of *equality nodes* and *equivalence class nodes*. The equality node is like the ATMS node, but it has no justifications, and its datum is an equality assertion such as  $x = y$ . All equality assertions, whether given or derived, are explicitly represented by equality nodes. Hence, in the worst case, we will have  $O(m^2)$  equality nodes in ED, where  $m$  is the total number of terms known.

The equivalence class node lists the terms (and assertions) that belong to that equivalence class. The notion of equivalence class employed for the remainder of the paper is the following: a *weak equivalence class* is a maximal set of terms that are weakly equivalent. Two terms  $t_1$  and  $t_2$  are *weakly equivalent* if there *exists* an environment under which  $t_1 = t_2$  is true. Note that the environment in question

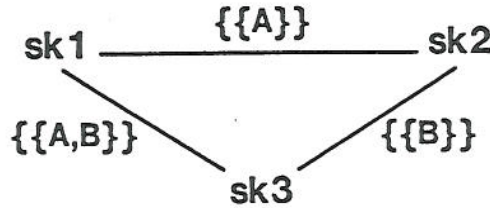


Figure 2.1: The equivalence class of Figure 1.1

need not be the same for all pairs of terms in the class.

The terms of an equivalence class under this definition form the nodes of a complete graph. The edges of the graph are equality assertions. The edge from node  $t_1$  to node  $t_2$  asserts that  $t_1 = t_2$ . Figure 2.1 shows the equivalence class of Figure 1.1 using this notation. The edges are labeled with the labels for the corresponding equality nodes. Since an equivalence class forms a complete graph, the number of equality nodes in a class is  $\frac{n(n-1)}{2}$ , where  $n$  is the number of terms in an equivalence class.

## 2.2 The Problem Solver

The problem solver of the specialized ATMS is given equalities of the form  $t_1 = t_2$  with their corresponding labels. Its task is to create and maintain equivalence class nodes by deriving all possible new equality nodes from the given assertion. To differentiate the nodes derived by the problem solver from the nodes given to the problem solver, we will call the latter the *primitive equalities*, and their environments, the *primitive environments*. Let us assume for now that each of the primitive environments introduced to the problem solver is disjoint.

For the purpose of describing how the new equality nodes are derived, let  $eq$  be the primitive equality  $t_1 = t_2$ , with  $l_{eq}$  as its label consisting of only primitive environments, and let  $EC_1$  and  $EC_2$  be two separate equivalence class nodes of size  $n_1$  and  $n_2$  respectively. Let  $Label$  be a function which takes an equality and returns its label. Let  $Combine-Labels$  be a function which takes two labels,  $l_1$  and  $l_2$ , and produces a new label by putting in a minimal form the set of environments

$l_{new}$ , where  $l_{new} = \{env1_i \cup env2_j \mid env1_i \in l1 \wedge env2_j \in l2\}$ .

The four cases that must be considered for deriving new equality nodes are given below.

**Case 1:** If neither  $t1$  nor  $t2$  exist in any of the equivalence class nodes in ED, i.e., both  $t1$  and  $t2$  are new terms never before encountered, create and assert into ED an equality node with  $eq$  and  $l_{eq}$ , and an equivalence class node listing  $t1$  and  $t2$ .

**Case 2:** Suppose  $t1 \in EC1$ , but  $t2$  does not exist in ED, that is, one of the terms (in this case  $t1$ ) has been previously encountered while the other is being introduced for the first time. Let  $EC1' = EC1 - \{t1\}$ . Then  $\forall t_i \in EC1'$ , for  $i = 1 \dots n_1 - 1$ , create and assert into ED an equality node with the equality  $t2 = t_i$ , where its label is computed as  $Combine-Labels(l_{eq}, Label(t1 = t_i))$ . Then, create and assert into ED an equality node for  $eq$  and  $l_{eq}$ , and add  $t2$  to  $EC1$ . Note that the number of new equality nodes derived is  $n - 1$  since a term of an equivalence class, when viewed as a vertex of a complete graph, has a degree of  $n - 1$ .

**Case 3:** Suppose  $t1 \in EC1$  and  $t2 \in EC2$ , that is, both terms were previously encountered but were never previously equated. Let  $EC1' = EC1 - \{t1\}$ , and  $EC2' = EC2 - \{t2\}$ . Then  $\forall t_i \in EC1'$ , for  $i = 1 \dots n_1 - 1$ , and  $\forall t_j \in EC2'$ , for  $j = 1 \dots n_2 - 1$ , create and assert into ED the following:

- An equality node with  $t_i = t_j$  and its label computed as:  
 $Combine-Labels(l_{eq}, Combine-Labels(Label(t1 = t_i), Label(t2 = t_j)))$ .
- An equality node with  $t2 = t_i$  and its label computed as:  
 $Combine-Labels(l_{eq}, Label(t1 = t_i))$ .
- An equality node with  $t1 = t_j$  and its label computed as:  
 $Combine-Labels(l_{eq}, Label(t2 = t_j))$ .

Hence, the number of new equalities derived from joining  $EC1$  and  $EC2$ , is  $(n_1 - 1)(n_2 - 1) + (n_1 - 1) + (n_2 - 1) = n_1 n_2 - 1$ . Then, create and assert into ED an equality node for  $eq$  and  $l_{eq}$ , and update  $EC1$  to be  $EC1 \cup EC2$ .

**Case 4:** When  $t1, t2 \in EC1$ , that is, both terms were previously encountered and were also equated, the label-update procedure is called, since  $eq$  is providing new environment(s) to be added to  $l_{eq}$ .

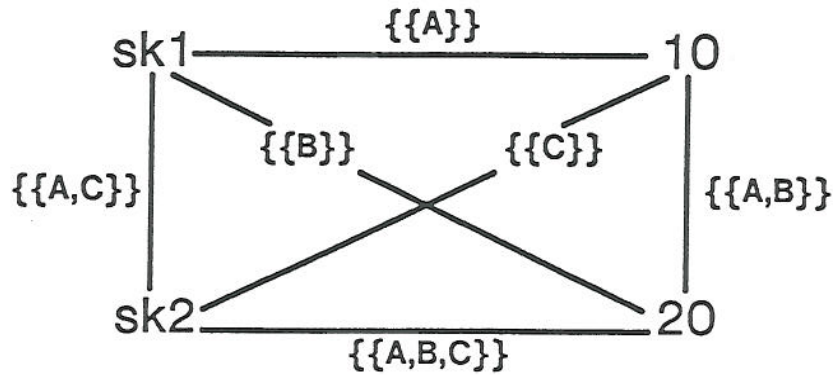


Figure 2.2: An equivalence class with a contradictory equality

While deriving new equality nodes, if the problem solver detects a derived equality between two different (non-Skolem) constants this must be declared a contradiction. The label for this equality is declared nogood. However, the equality node must still be created so that the problem solver may properly derive new equalities in the future. To illustrate this, consider the equivalence class shown in Figure 2.2 constructed from asserting the following two equalities:  $sk1 = 10$  with  $\{\{A\}\}$ , and  $sk1 = 20$  with  $\{\{B\}\}$ . The equality node for  $10 = 20$  is created with the label  $\{\{A,B\}\}$ . Now, suppose that a new equality  $sk2 = 10$  with  $\{\{C\}\}$  is given to the problem solver. Then,  $EC1 = \{sk1, 10, 20\}$ , and  $EC1' = \{sk1, 20\}$ , and the following are derived:

- An equality node with  $sk2 = sk1$ ) and its label computed as:  
 $Combine-Labels(\{\{C\}\}, Label(sk1 = 10))$ .
- An equality node with  $sk2 = 20$  and its label computed as:  
 $Combine-Labels(\{\{C\}\}, Label(10 = 20))$ .

If the equality node with  $10 = 20$  did not exist along with its nogood label,  $\{\{A,B\}\}$ , the equality node with  $sk2 = 20$  would not have a correct label. Hence the contradictory equality nodes are also maintained by the problem solver.

## 2.3 The Label-Update Algorithm

### 2.3.1 The Algorithm

The label-update procedure is given an existing equality node, called the *entry node*, along with a new primitive environment,  $env_{new}$ . Let  $l_{entry}$  be the existing label of the entry node, and let  $L\text{-Updates}$  be the set of all labels in the equivalence class containing the entry node, but not including  $l_{entry}$ . Its task is to update all of the labels in  $L\text{-Updates}$ , and then to add  $env_{new}$  to  $l_{entry}$ . The procedure is as follows:

- For each  $l_i \in L\text{-Updates}$  do:
  - For each  $env_{i,j} \in l_i$  do:
    - For each  $env_k \in l_{entry}$  do:
      1. If  $(env_k \cap env_{i,j}) = \emptyset$ , do nothing.
      2. Else, compute a new environment to be added to  $l_i$  as:
        - $(env_k \oplus env_{i,j}) \cup env_{new}$ <sup>1</sup>
        - If the newly computed environment is not subsumed by any environment in  $l_i$  then add it to  $l_i$ .

Note that during label update, if an equality supported by  $l_i$  is a contradictory equality (an equality between two different non-Skolem constants), the new labels computed for it will be declared nogood.

### 2.3.2 An Example

Consider the equivalence class shown in Figure 2.3. Suppose a new environment  $\{D\}$  arrives on the label for  $sk1 = sk2$ . The updated label for this equality is  $\{\{A\}, \{D\}\}$ , and  $env_k$ , for  $k = 1$ , is  $\{A\}$  and  $env_{new}$  is  $\{D\}$ . The other labels in the equivalence class shown in Figure 2.3 are updated as prescribed by the label-update algorithm given above. The results of applying the steps are summarized in Table 2.1. The updated equivalence class of Figure 2.3 is shown in Figure 2.4. Note that in this example the algorithm did not compute any redundant environments.

---

<sup>1</sup> $\oplus$  is the disjoint union operation defined as:  $A \oplus B = (A - B) \cup (B - A)$ .



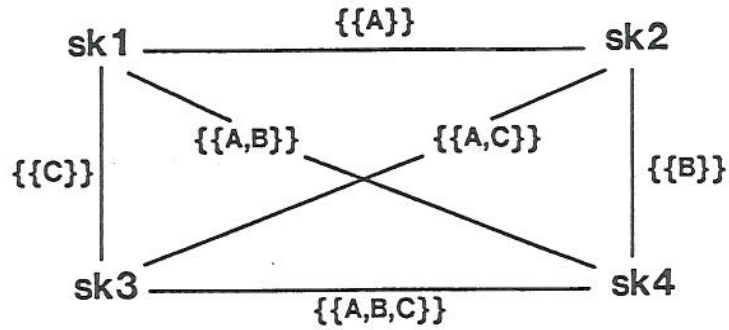


Figure 2.3: Before the label updates

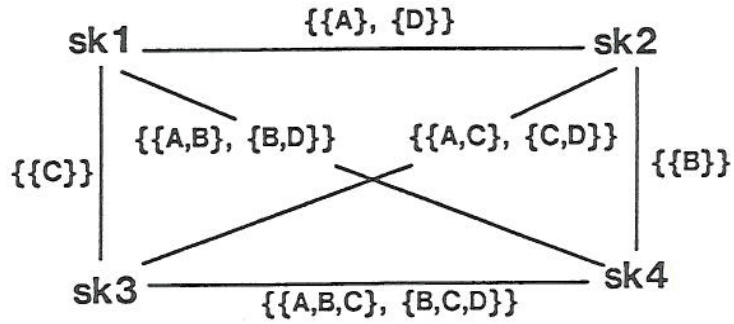


Figure 2.4: After the label updates

Table 2.1: Summary of the label-update process

| i | j | $env_{i,j}$   | Result of $(env_k \cap env_{i,j})$ ,<br>where $env_k = \{A\}$ | Result of $((env_k \oplus env_{i,j}) \cup env_{new})$ ,<br>where $env_k = \{A\}$ , and<br>$env_{new} = \{D\}$ |
|---|---|---------------|---------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------|
| 1 | 1 | $\{B\}$       | $\emptyset$                                                   | not computed                                                                                                  |
| 2 | 1 | $\{C\}$       | $\emptyset$                                                   | not computed                                                                                                  |
| 3 | 1 | $\{A, B\}$    | $\{A\}$                                                       | $\{B, D\}$                                                                                                    |
| 4 | 1 | $\{A, C\}$    | $\{A\}$                                                       | $\{C, D\}$                                                                                                    |
| 5 | 1 | $\{A, B, C\}$ | $\{A\}$                                                       | $\{B, C, D\}$                                                                                                 |

### 2.3.3 An Explanation

The algorithm succeeded in updating the labels correctly without explicitly traversing the justification links, since the labels alone were able to provide the algorithm with the necessary information to propagate the new support. This is due to the fact that the labels implicitly hold the dependency structure. So, by performing computations directly on the labels, the algorithm can easily determine which labels to update and how to update them.

Determining which labels to update is based on the following observation. When a new equality,  $eq$ , is asserted with a new primitive environment  $env_{new}$ , all new equalities derived from this new assertion are supported by the environments that contain  $env_{new}$ . Under such circumstances, for any two environments  $env1$  (supporting  $eq1$ ) and  $env2$  (supporting  $eq2$ ), if they have a common assumption, i.e.,  $env1 \cap env2 = S$  (where  $S \neq \emptyset$ ), then either  $eq2$  was one of the equalities that originally derived  $eq1$ , or  $eq1$  was one of the equalities that originally derived  $eq2$ , or both  $eq1$  and  $eq2$  were originally derived from some equalities (or an equality) that are supported by the individual assumptions in the set  $S$ . Hence, in general, when an existing equality's label  $l$  is updated to include a new support,  $env_{new}$ , the equality facts affected by  $env_{new}$  are the ones with labels that have assumptions in common with  $l$ 's environments. For this reason, the algorithm performs intersection operations to determine which labels to update.

Once the algorithm determines which labels to update, computing new environments to be added to them is based on the following observation. Consider Figure 2.5, which shows a portion of an equivalence class. Let us focus on the two derived equalities  $t1 = t2$  and  $t5 = t6$ . The environment for  $t5 = t6$  can be viewed as a connected path from  $t5$  to  $t6$  containing only primitive environments. In this case, the path is  $\langle t5, t3, t4, t6 \rangle$ , which gives us the environment  $\{B, C, E\}$ . Similarly, the environment for  $t1 = t2$  is the connected path  $\langle t1, t3, t4, t2 \rangle$ , which gives us  $\{A, C, E\}$ . Second, the intersection of the environments for  $t1 = t2$  and  $t5 = t6$ ,  $\{C\}$ , is the *shared* environment—that is, the shared path. Suppose that an environment,  $\{F\}$ , is given as new support for  $t1 = t2$ . The label-update algorithm is now

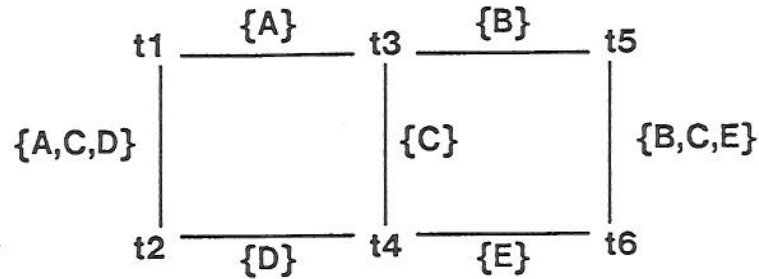


Figure 2.5: A partial equivalence class with a shared support

expected to update  $t5 = t6$ 's label (among other labels) to incorporate the new support  $\{F\}$ . However, that is analogous to finding a new connected path from  $t5$  to  $t6$  (besides  $\langle t5, t3, t4, t6 \rangle$ ) that passes through the newly supported equality,  $t1 = t2$ . That new path is  $\langle t5, t3, t1, t2, t4, t6 \rangle$ , and it can be computed by first subtracting the path (from  $t3$  to  $t4$ ) shared by the two equalities  $t1 = t2$  and  $t5 = t6$ , and then including the newly supported path (from  $t1$  to  $t2$ ). The label-update algorithm does just that when it computes  $(\{A, C, D\} \oplus \{B, C, E\}) \cup \{F\} = \{A, B, D, E, F\}$  for the label of  $t5 = t6$  equality. In effect,  $\{F\}$ , along with  $\{A\}$  and  $\{D\}$ , is substituted for the old shared environment,  $\{C\}$ , to provide a new supporting environment for  $t5 = t6$ . The entire calculation was performed without explicitly traversing paths or justification links, since the labels implicitly hold the dependency structure.

The fact that we are using the labels to obtain the dependencies among the equalities requires that we must retain the nogood environments within the labels. In fact, a nogood environment cannot be removed from a label until it can be replaced with a non-nogood environment that implicitly holds the same dependency structure (see Chapter 3).

### 2.3.4 Computational Costs

Since there are  $\frac{n(n-1)}{2}$  equalities in an equivalence class with  $n$  terms, and since the algorithm always attempts to update all but one of the labels for those equalities, the number of label update *attempts* is  $\Theta(n^2)$ . This figure is significantly better than the  $\Theta(n^3)$  label *computations* performed by de Kleer's algorithm. Moreover,

note from the algorithm that not all label update attempts will result in a label computation (since  $(env_k \cap env_{i,j}) = \emptyset$  may be true). The number of actual label computations depends on the number of environments that will intersect with the entry node's existing environments,  $env_k \in l_{entry}$ . The worst case occurs when  $env_k$  is a derived environment that consists of every primitive environment of an equivalence class. Such an environment will intersect with all  $\frac{n(n-1)}{2}$  equality labels' environments, and hence will result in  $\Theta(n^2)$  label computations.

The best case is obtained when  $env_k$  is one of the primitive environments. Recall from Section 2.2, that when a new equality,  $eq$ , is introduced to the problem solver with a primitive label,  $l_{eq}$ , the problem solver derives exactly  $n_1 - 1$  new equalities for an equivalence class with  $n_1$  terms and  $\frac{n_1(n_1-1)}{2}$  equalities. The resulting equivalence class will have  $n_2 = n_1 + 1$  terms and  $\frac{n_2(n_2-1)}{2}$  equalities. Next, if  $eq$  is updated, an environment of  $l_{eq}$  will intersect exactly  $n_1 - 1$  times, or will intersect with  $n_2 - 2$  environments among the  $\frac{n_2(n_2-1)}{2}$  equality labels. (This can be guaranteed since every primitive environment introduced to the equality database is disjoint.) Hence, the cost of label computation for the best case is  $O(n)$ .

### 2.3.5 Proof of Correctness

We demonstrate the algorithm's correctness by an inductive proof.

First we consider the base case—a three term equivalence class. Given any two equalities  $x = y$  (in environment  $env1$ ) and  $y = z$  (in environment  $env2$ ) the third equality  $x = z$  can be derived using the transitivity axiom. (We will refer to these simple three way equalities as 'triangles' since they form triangles in the graphical notation introduced earlier.) Since  $x = z$  was derived from the equalities supported with  $env1$  and  $env2$ , the derived environment  $env3$ , which supports  $x = z$ , is defined as:  $env3 = env2 \cup env1$ . Since we have assumed that  $env1$  and  $env2$  are disjoint environments, the following relationships hold for the three environments in a triangle:

$$env3 = env1 \oplus env2 \tag{2.7}$$

$$env2 = env1 \oplus env3 \quad (2.8)$$

$$env1 = env2 \oplus env3 \quad (2.9)$$

We now prove that for any triangle in an equivalence class, equations (2.7), (2.8) and (2.9) hold. The proof is by induction on  $n$ , the size of the equivalence class. Consider the equivalence class of  $n$  terms illustrated in Figure 2.6. The new equality added between  $t2$  and the existing term  $t1$  will result in  $n-1$  triangles being added to the equivalence class. Since each new triangle is computed in exactly the same way as the simple triangle above, and we assume that each new environment  $envs$  is unique, then the relationships of (2.7), (2.8), and (2.9) must hold for each new triangle added. Hence, by induction, the relationships of (2.7), (2.8), and (2.9) hold for all triangles in an equivalence class.

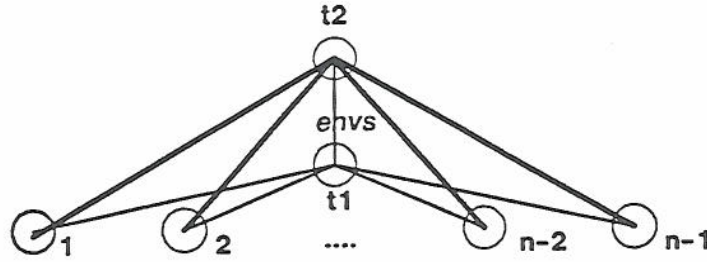


Figure 2.6: Incremental extension of an equivalence class

Suppose an equality  $eq1$  is in an equivalence class of size  $n$ . Let  $eq2_i$  and  $eq3_i$  be the equalities that form the  $n-1$  triangles with  $eq1$ . Now consider a new support  $env1_{new}$  arriving on  $eq1$ . To update this equivalence class, the labels of  $eq2_i$  and  $eq3_i$  for each of the triangles are updated. Let  $env1$ ,  $env2$ , and  $env3$  be the pre-existing environments of  $eq1$ ,  $eq2_i$ , and  $eq3_i$ ; respectively. According to de Kleer, the new environments to be added to the labels of  $eq2_i$  and  $eq3_i$ ; (referred to as  $env2_{new}$  and  $env3_{new}$  respectively) are computed as follows:

$$env2_{new} = env3 \cup env1_{new} \quad (2.10)$$

$$env3_{new} = env2 \cup env1_{new} \quad (2.11)$$

From (2.7) we can substitute into (2.10), and from (2.8) we can substitute into (2.11) to obtain the following two equations:

$$env2_{new} = (env1 \oplus env2) \cup env1_{new} \quad (2.12)$$

$$env3_{new} = (env1 \oplus env3) \cup env1_{new} \quad (2.13)$$

The equations (2.12) and (2.13) directly correspond to the disjoint union and union step of the label-update algorithm. Hence, we have shown that the algorithm behaves correctly.

## Chapter 3

### Extending the Method

It is clear from the proof given above that the label-update algorithm will behave incorrectly if any of the primitive environments are not disjoint, since the relationships (given above) among the environments of a triangular equivalence class will not hold. To accommodate non-disjoint primitive environments, the incoming primitive environments are made disjoint by an equality token mechanism described below.

#### 3.1 Equality Tokens

The primitive environments can be guaranteed to be disjoint by assigning globally unique names, which we will call *equality tokens*, to each and every environment introduced to the equality database, either through new equality assertions or as new support for an existing equality. This assignment of globally unique names can be viewed as a substitution where each environment,  $\{A_1, A_2, \dots, A_i\}$ , is replaced with  $\{T_j\}$ , where each  $T_j$  is the globally unique equality token. Under this design, label updates, as well as the computation of labels for the newly derived equalities, will be done on labels containing equality tokens, not ATMS assumptions.

For example, suppose two equality assertions are given to the problem solver:

- Assert  $sk1 = sk2$  with  $\{\{A, B\}\}$ ,
- Assert  $sk2 = sk3$  with  $\{\{B, C\}\}$ .

(Notice that the environments of these two assertions are not disjoint.) Then, the following renaming, denoted as  $\rightarrow$ , will occur:

- $\{A, B\} \rightarrow \{1\}$ ,
- $\{B, C\} \rightarrow \{2\}$ .

The newly derived equality node for the  $sk1 = sk3$  equality will have  $\{\{1, 2\}\}$  as its label instead of  $\{\{A, B, C\}\}$ . When a new support, say  $\{D\}$ , on  $sk1 = sk2$  is introduced, it will be renamed as  $\{3\}$ . The label-update algorithm will proceed as usual, but using the equality tokens, and will cause  $\{2, 3\}$  to be included in the  $sk1 = sk3$  label. (One can see that this update is correct since  $\{2, 3\}$  maps to  $\{B, C, D\}$ .)

The equality tokens must be translated back to their equivalent ATMS form for the purposes of queries into the equality database<sup>1</sup> to determine if an environment consisting of equality tokens is a nogood. The mapping from the equality tokens to their corresponding ATMS environments can be done efficiently by storing the mapping from the individual equality tokens to their corresponding ATMS environments.

### 3.2 Optimization

A significant cost in both de Kleer's and the label-update algorithm of the specialized ATMS is the subsumption check that must be performed for each of the newly derived environments. However, there are certain cases where the subsumption checks can be skipped in the label-update algorithm of the specialized ATMS because the derived environments are guaranteed to be non-redundant.

Suppose an entry node  $N$  which contains a primitive environment consisting of single equality token,  $\{T_{old}\}$ , within its existing label, is given a new support—a new primitive environment,  $\{T_{new}\}$ . All of the new environments to be added to all other labels in the same equivalence class as  $N$  can be computed by simply substituting  $T_{new}$  in place of all occurrences of  $T_{old}$ . This is because the inferences performed when  $T_{old}$  was propagated during previous updates will be exactly the

<sup>1</sup>The translation will also be necessary during label updates if the specialized ATMS is linked to the standard ATMS.



same inferences needed for  $T_{new}$  to be propagated. Therefore  $T_{new}$  may simply replace  $T_{old}$ .

Consider the alternate case in which the entry node,  $N$ , contains only derived environments within its label. Suppose it is given the environment  $\{T_{new}\}$ , as a new support. If, during the label update process, we encounter a node  $M$  whose label contains a primitive environment  $\{T_{old}\}$ , we can completely update  $M$ 's label by only considering  $\{T_{old}\}$  in combination with the existing tokens of  $N$ . We do not need to consider the other tokens in  $M$ 's label. Furthermore, the newly computed environments for  $M$  do not need to be checked for subsumption.

The first optimization is applicable whenever an equality node receives multiple external supporting environments. When our specialized equality ATMS is embedded within a de Kleer-style ATMS, this happens often, because each supporting ATMS environment is mapped into a unique primitive equality token.

### 3.3 The Revised Label-Update Algorithm

The revised label-update algorithm with optimization is given in Figure 3.1. The revised label-update algorithm is given an existing equality node, the entry node, along with a new primitive environment,  $Penv_{new}$ , to be added to the entry node's existing label,  $l_{entry}$ . But before updating the labels,  $Penv_{new}$  is made disjoint by creating a new globally unique equality token for it. Then, the labels  $L$ -Updates, which consist of all labels in the equivalence class containing the entry node, but not including  $l_{entry}$ , are updated as prescribed by the revised algorithm. Finally,  $Penv_{new}$  is included in  $l_{entry}$ .

### 3.4 Removing Inconsistencies from the Labels

As stated before, unlike de Kleer's ATMS, which removes inconsistent environments (nogoods) from the labels, the label-update algorithm of the specialized ATMS must retain them in order to encode the necessary dependency structure. However, as the number of nogoods in the equality database increases, the efficiency of the spe-

1. If there is a primitive environment  $Penv_{exist} \in l_{entry}$  do:
  - For each  $l_i \in L\text{-Updates}$  do:
    - For each derived environment  $Denv_{i,j} \in l_i$  do:
      - a. If  $(Penv_{exist} \cap Denv_{i,j}) = \emptyset$ , do nothing.
      - b. Else, compute a new environment to be added to  $l_i$  as:
        - $(Penv_{exist} \oplus Denv_{i,j}) \cup Penv_{new}$ .
2. Else do:
  - For each  $l_i \in L\text{-Updates}$  do:
    - a. If there is a primitive environment  $Penv_j \in l_i$  do:
      - For each derived environment  $Denv_k \in l_{entry}$  do:
        - i. If  $(Penv_j \cap Denv_k) = \emptyset$ , do nothing.
        - ii. Else, compute a new environment to be added to  $l_i$  as:
          - $(Penv_j \oplus Denv_k) \cup Penv_{new}$ .
    - b. Else do:
      - For each derived environment  $Denv_{i,j} \in l_i$  do:
        - For each derived environment  $Denv_k \in l_{entry}$  do:
          - i. If  $(Denv_{i,j} \cap Denv_k) = \emptyset$ , do nothing.
          - ii. Else, compute a new environment to be added to  $l_i$  as:
            - $(Denv_{i,j} \oplus Denv_k) \cup Penv_{new}$
            - If the newly computed environment is not subsumed by any environment in  $l_i$  then add it to  $l_i$ .

Figure 3.1: The revised label-update algorithm

cialized ATMS decreases (i.e., the more environments the labels have, the more time will be required to construct new environments—both during label updates and when deriving new equalities). Therefore, to retain the efficiency of this system, we identify conditions that introduce nogoods, and those cases where nogood environments can be removed, without losing dependency structure information:

Nogoods are introduced in the following circumstances:

1. If equivalence classes have many non-Skolem constants, and hence consists of contradictory equalities.
2. If equivalence classes have many equalities derived from one of the contradictory equalities.
3. If the problem being solved by this system introduces many nogoods outside of the nogoods introduced from the contradictory equalities.

The first circumstance poses no additional inefficiency to the system, since those nogoods are required for the labels when deriving new equalities and hence do not need to be removed. The second and third circumstances introduce the inefficiencies that are completely avoided in de Kleer's system.

### 3.4.1 Removing Primitive Nogood Environments

From the first optimization method given in Section 3.2, we know that the label-update algorithm can use just one of the primitive environments of the entry node's label to update other labels correctly—the remaining primitive environments of such labels are not necessary. Therefore, if any of the remaining primitive environments are also known to be nogood, they can be removed from the label without affecting the correctness of the label-update algorithm. Furthermore, any of the derived nogood environments subsumed by a primitive nogood environment can be removed from a label. For example, consider the equivalence class shown in Figure 3.2, constructed from the following three equality assertions, and a nogood declaration resulting from the third circumstance given above:

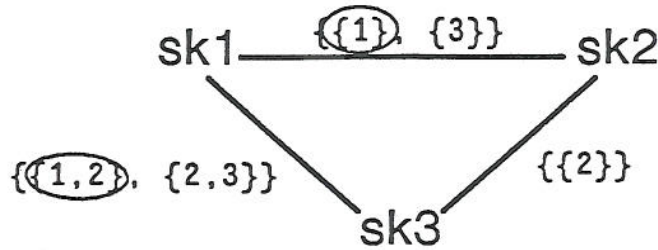


Figure 3.2: An equivalence class with the nogood environments encircled

- $sk1 = sk2$  with  $\{\{A\}\} \rightarrow \{\{1\}\}$
- $sk2 = sk3$  with  $\{\{B\}\} \rightarrow \{\{2\}\}$
- $sk1 = sk3$  with  $\{\{C\}\} \rightarrow \{\{3\}\}$
- $\{A\}$  is a nogood, so  $\{1\}$  is a nogood

Since the label with the primitive nogood environment  $\{1\}$  includes a non-nogood primitive environment  $\{3\}$ , both  $\{1\}$  (of the  $sk1 = sk2$  equality) and  $\{1, 2\}$  (of the  $sk1 = sk3$  equality) can be removed from the labels.

### 3.4.2 Removing Derived Nogood Environments

Suppose a derived environment,  $Denv = \{T_1, T_2, \dots, T_i\}$ , is declared to be a nogood resulting from the third circumstance given above. Such an environment cannot be removed from a label until all primitive environments consisting of one its individual equality tokens, i.e.,  $T_1, T_2, \dots, T_i$ , have been intersected with other environments (during label updates) due to the labels containing them receiving a new support. That is, if each  $T_j \in \{T_1, T_2, \dots, T_i\}$  correspond to the primitive environment,  $Penv_{exist}$  of the  $(Penv_{exist} \cap Denv_{i,j}) = \emptyset$  step of the algorithm given in Section 3.3, then  $denv$  can be removed from a label.

For example, consider the equivalence class shown in Figure 3.3, constructed under the following scenario:

- Assert  $sk1 = sk2$  with  $\{\{A\}\} \rightarrow \{\{1\}\}$ .

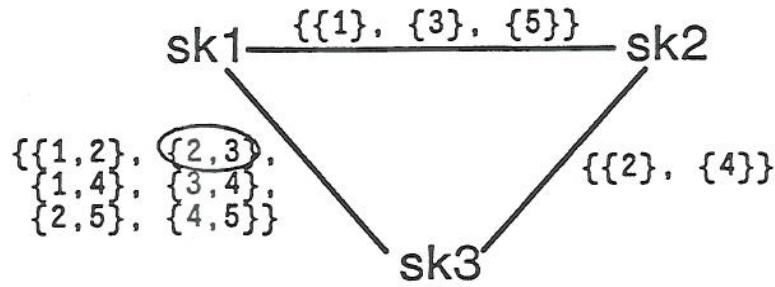


Figure 3.3: An equivalence class with the nogood environment encircled

- Assert  $sk2 = sk3$  with  $\{\{B\}\} \rightarrow \{\{2\}\}$ .
- Assert  $sk1 = sk2$  with  $\{\{C\}\} \rightarrow \{\{3\}\}$ .
- Assert  $sk2 = sk3$  with  $\{\{D\}\} \rightarrow \{\{4\}\}$ .

The last two assertions resulted in two label updates using the environments  $\{1\}$  and  $\{2\}$ , respectively, as  $Penv_{exist}$ . Next, suppose  $\{B, C\}$  is declared nogood (due to the third circumstance given above), and hence  $\{2, 3\}$  becomes a nogood. According to the condition given above, this derived nogood environment,  $\{2, 3\}$ , cannot be removed until *both*  $\{2\}$  and  $\{3\}$  have participated as  $Penv_{exist}$  during label updates. We know that  $\{2\}$  has been used as  $Penv_{exist}$  but not  $\{3\}$ . Hence  $\{2, 3\}$  cannot be removed yet. Later, if the label-update algorithm receive the following:

- Assert  $sk1 = sk2$  with  $\{\{E\}\} \rightarrow \{\{5\}\}$ ,

and uses  $\{3\}$  to compute two new environments,  $\{2, 5\}$  and  $\{4, 5\}$ , the derived nogood environment,  $\{2, 3\}$ , can be removed from the label for the  $sk1 = sk3$  equality. The final resulting equivalence class is shown in Figure 3.4. However, now, the algorithm (with the optimization) cannot arbitrarily choose any one of the primitive environments of an entry node's label to be  $Penv_{exist}$ . For example, again consider the equivalence class shown in Figure 3.4. Suppose another support,  $\{F\} \rightarrow \{6\}$ , arrives on the  $sk1 = sk2$  equality. The correctly updated equivalence class is shown in Figure 3.5. Notice that this label update introduced two new environments,  $\{2, 6\}$  and  $\{4, 6\}$ , to the label for the  $sk1 = sk3$  equality. These two new environments

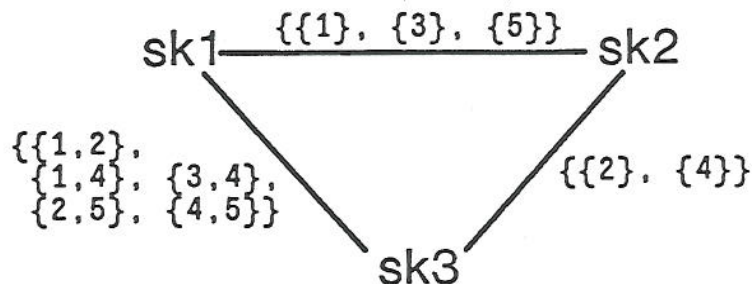


Figure 3.4: An equivalence class of Figure 3.3 with the derived nogood environment removed

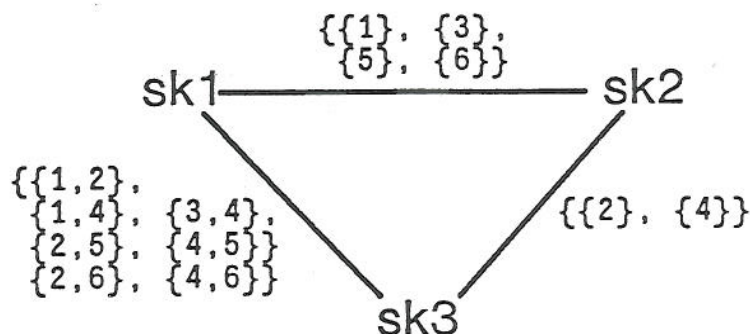


Figure 3.5: An equivalence class of Figure 3.4 with a new support

can only be produced by the algorithm if it chose either  $\{1\}$  or  $\{5\}$  (of the entry node's label:  $\{\{1\}, \{3\}, \{5\}\}$ ) to be  $Penv_{exist}$ . If  $\{3\}$  were arbitrarily chosen to be  $Penv_{exist}$ , only  $\{4,6\}$  would have been produced. (This is because the label for the  $sk1 = sk3$  equality no longer has the nogood environment,  $\{2,3\}$ , to intersect with  $\{3\}$ .) Hence, in order to update correctly, the algorithm must not choose a primitive environment to be  $Penv_{exist}$  if it subsumes any one of the derived nogood environments which have been removed from a label. Implementing the restriction on choosing an appropriate primitive environment to be  $Penv_{exist}$  (from an entry node's label) is easily accomplished by always choosing a primitive environment most recently added to the entry node's label. This implementation suffices because any primitive environment that subsumes a derived nogood environment cannot be the most recently added primitive environment of a label, if it has participated as  $Penv_{exist}$ .

### 3.4.3 A Concluding Remark on Removing Inconsistencies

Although the cases where inconsistencies, or the nogood environments, can be removed from labels have been identified, it is difficult to determine, without applying the specialized ATMS to various problems, how often such cases will occur. Furthermore, the environments consisting of equality tokens (primitive or derived) can be recognized as nogoods only after mapping the equality tokens to their corresponding ATMS forms. Hence, removing the inconsistencies from the labels will not necessarily benefit the specialized ATMS.

## Chapter 4

### Implementation Issues

#### 4.1 The Equality Database Data Structure

The data structure of the equality database is designed so that the following tasks can be performed efficiently:

1. Given an equality, determine if it already exists in the equality database.
2. Given an existing equality, retrieve the corresponding equality node.
3. Given an entry node, retrieve the set of labels of the equalities in the same equivalence class as the entry node.
4. Given a new equality  $t1 = t2$ , retrieve the set (or sets) of terms in the same equivalence class (or classes) as  $t1$  or  $t2$  to derive new equalities.
5. Join two equivalence class nodes into one when the terms in those classes are equated.

The data structure that allows the problem solver to efficiently perform the first two tasks classifies the equality nodes according to the terms they contain. The resulting structure is a discrimination net, that can be efficiently implemented using hash arrays. See Figure 4.1 for the structure created for the equivalence class of Figure 2.3. (The hash arrays are the areas containing the terms, and the equality nodes are the boxes with equal signs. The remaining structure, the equivalence class node, is explained later.) Given an equality, determining if it exists is trivial: hash on one of the terms of the equality and then on the other, in canonical order.



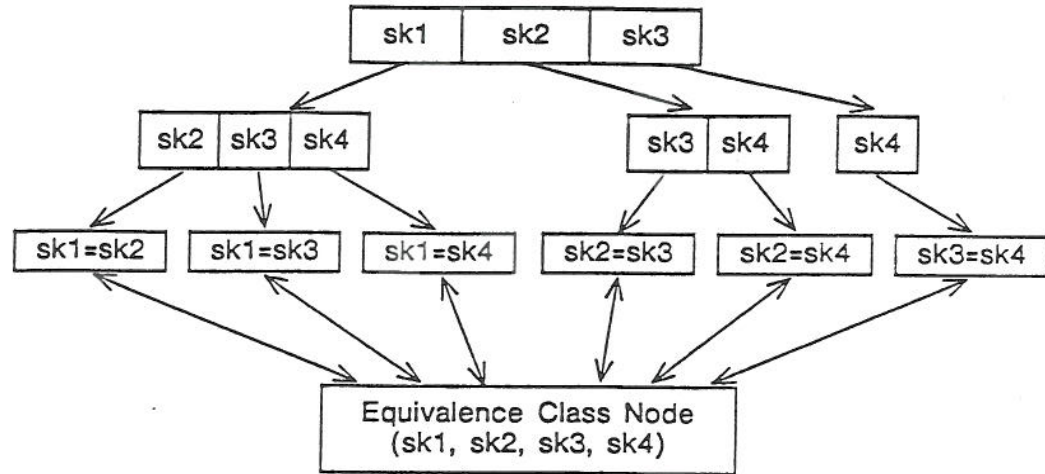


Figure 4.1: A partial data structure for the equivalence class in Figure 2.3

Then, follow the pointer to retrieve the corresponding equality node. Giving this equality node as an entry node, the label-update procedure must obtain the set of labels of the equality nodes in the same equivalence class as the entry node. (This is the third task given above.) Hence, it becomes necessary to link all equality nodes to their equivalence class nodes. Note that all of the equality nodes in Figure 4.1 point to the equivalence class node.

The fourth task requires that the set of terms in an equivalence class be accessible given any one of the terms in that set. Ideally, such a set can be efficiently accessed if there is a link from each term to its equivalence class node (since it lists the complete set of terms belonging to that class). However, since the links to and from the equivalence class nodes are modified whenever two equivalence classes are joined, it is recommended that such links be minimized. Therefore, instead of linking a term directly to its equivalence class node, it is linked to any one of the equality nodes that involves that term, allowing the equivalence class node to be accessed via the equality node. See Figure 4.2 for the data structure resulting from incorporating the links from the terms to the equality nodes.

Joining the two equivalence class nodes into one class node (the fifth task) can become time consuming if the individual equivalence classes become very large.

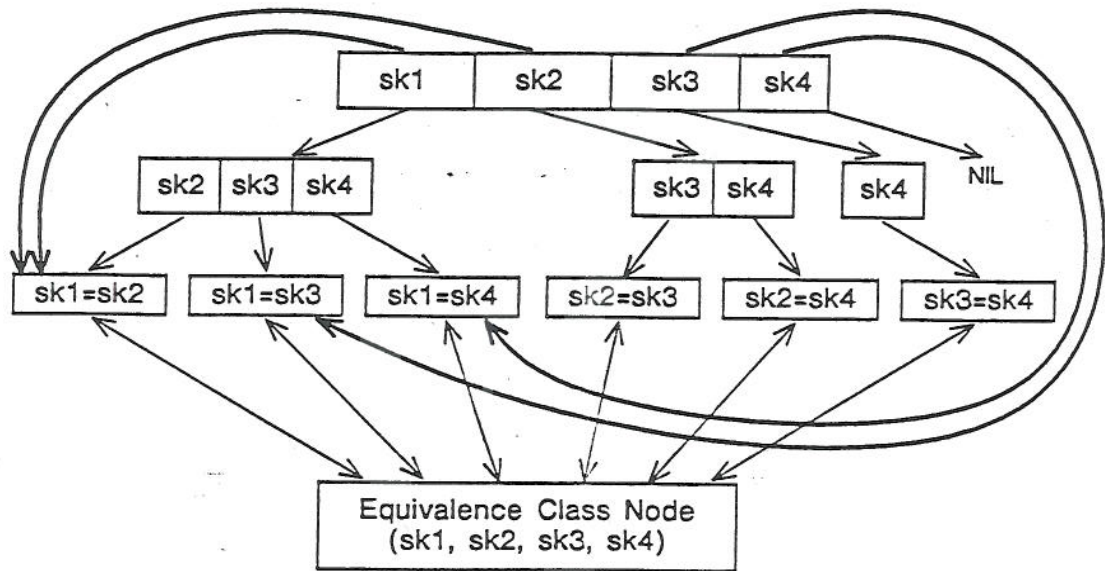


Figure 4.2: A complete data structure for the equivalence class in Figure 2.3

Hence, to minimize the re-linking of the pointers to and from the equivalence class nodes, the UNION-FIND algorithm is applied to the equivalence class nodes.  $\text{UNION}(EC_1, EC_2, EC_3)$  will join two equivalence class nodes,  $EC_1$  and  $EC_2$ , by creating a new equivalence class node,  $EC_3$ , which will have pointers to  $EC_1$  and  $EC_2$ . This will result in constructing a tree of equivalence class nodes. Next,  $\text{FIND}(term)$  will locate an equivalence class that contains  $term$ . Finally, to reduce the cost during the FIND operation, the path compression algorithm is applied on the equivalence class tree [Aho, Hopcroft, & Ullman, 1974].

#### 4.1.1 The Equality Node Data Structure

The equality nodes of the equality database consist of the following fields: the equality datum, the label, and an equivalence class node that the equality belongs to. Furthermore, the label must be separated into two types of environments: primitive and derived, so that the revised label-update procedure may perform label updates using just primitive environments or just derived environments. Note that no distinction is made between the equality nodes that represent contradictory equalities and the ones that do not.

### 4.1.2 The Equivalence Class Node Data Structure

The equivalence class nodes consist of two fields: the list of terms in the equivalence class and the list of equality nodes that represent the equalities among the terms of the class. The list of equality nodes is separated into two types of lists: the contradictory equality node list and the non-contradictory equality node list, so that when updating both types of equality node labels, the new environments computed for the contradictory equality node labels can be declared nogood directly.

## 4.2 Interfacing the Specialized ATMS with a de Kleer-style ATMS

Some of the techniques for interfacing the specialized ATMS with a de Kleer-style ATMS are discussed here.

The equality nodes in the equality database can be interfaced with the ATMS dependency structure through ATMS nodes that have equality nodes as their consequences or as their justifications. Under this combined structure, de Kleer's label-update algorithm is applied just to the ATMS nodes by traversing the justification links until it reaches an equality node—upon which the control is given to the specialized ATMS's label-update procedure to update the equality nodes. The control is returned when any of the equality nodes being updated have an ATMS node as one of their consequences.

The links from the ATMS nodes to the equality nodes are constructed whenever the antecedent of a rule that has an equality fact as its consequence is satisfied. The links to the ATMS nodes from the equality nodes are constructed whenever the antecedent of a rule is satisfied by an equality fact in the equality database.

## Chapter 5

### Summary

#### 5.1 Advantages and Disadvantages

The advantages of the specialized ATMS are summarized by comparing it to the approach of incorporating the transitivity axiom into de Kleer's ATMS (described in Section 1.1.1):

- The worst case time complexity of the label-update algorithm has been reduced from  $\Theta(n^3)$  to  $\Theta(n^2)$  label update attempts. In addition, since not all of these attempts result in label computations, the actual number of these label computations can be significantly lower.
- Through optimization techniques, the label-update algorithm can skip subsumption checks in many cases.
- The problem solver that derived two redundant equalities for every new equality derived has been replaced by one that derives only the necessary equalities.
- The space required to store the justification links is eliminated.

The disadvantages of the extended version of the specialized ATMS, are summarized as follows:

- Nogood environments can be removed from the labels only under a few specific conditions.
- Each environment consisting of equality tokens must be translated to its corresponding ATMS form before it can be determined to be a nogood.

## 5.2 Future Research Issues

There are several issues that can be pursued from the results of this thesis.

First of all, the apparent tradeoff between the performance of de Kleer's ATMS and the specialized ATMS when applied to non-trivial problems needs to be explored: the label-update algorithm of the specialized ATMS performs efficiently when the problem produces few nogoods and many disjoint primitive environments. In contrast, de Kleer's label-update algorithm performs efficiently when the problem produces many nogoods or if it produces very few disjoint primitive environments.

This tradeoff can be explored by empirically studying the performance of both methods when applied to a variety of problems that vary the following problem characteristics: the ratio of non-nogoods to nogoods, the ratio of internal nogoods (i.e., those found through contradictory equalities) to external nogoods, and the distribution of primitive to derived environments.

Other issues to be explored include:

- Extending the equivalence relations to include equalities among constructors (lists) and functions. Such equality relations complicate the problem solver's task by requiring it to derive new equality nodes from two or more constructors that are equal to one another. (I.e., if the two constructors  $C(sk1, 10, sk2)$  and  $C(5, sk3, sk4)$  are equal to one another, then the problem solver must derive the following three equalities:  $sk1 = 5, sk3 = 10, sk2 = sk4$ .)
- Investigating an alternative approach to equality tokens so that all nogood environments can be safely removed from the labels.
- Finding a method that will eliminate the remaining subsumption checks during label updates, which occur when two derived environments are intersected.
- Designing efficient interface techniques between the specialized ATMS and de Kleer-style ATMS.

## Bibliography

- Aho, A. V., Hopcroft, J. E., and Ullmann, J. E., 1974. The Design and Analysis of Computer Algorithms. *Addison-Wesley, Reading, Mass.*
- Arden, B. W., Galler, B. A., and Graham, R. M., 1961. An Algorithm for Equivalence Declaration. *Comm. ACM*, 4 (7), pp. 310-314.
- de Kleer, J., 1986a. An Assumption-based TMS. *Artificial Intelligence*, 28 (2), pp. 127-162.
- de Kleer, J., 1986c. Problem-solving with the ATMS. *Artificial Intelligence*, 28 (2), pp. 197-224.
- Flann, N. S., Dietterich, T. G., and Corpron, D. R., 1987. Forward Chaining Logic Programming with the ATMS. *AAAI*, pp. 24-29.
- Galler, B. A., and Fisher, M. J., 1964. An Improved Equivalence Algorithm. *Comm. ACM*, 7 (5), pp. 301-303.
- McAllester, D., 1982. Reasoning Utility Package User's Manual. Artificial Intelligence Laboratory, AIM-667, MIT, Cambridge, MA.

## Appendix

## Appendix A

### Glossary of Terms

**Assumed ATMS node.** An ATMS node representing an assumed premise. The label will contain an environment which in turn contains only one assumption symbol.

**Assumption.** A symbol that is associated with a premise that is declared to be an assumed fact. Normally each premise is assigned a unique assumption symbol. A set of assumptions is an ATMS environment.

**ATMS.** Stands for an assumption-based truth maintenance system. Developed by de Kleer. Its key feature is that it is a truth maintenance system that allows problem solving to take place efficiently under multiple contexts simultaneously.

**ATMS database.** A database for the ATMS nodes.

**ATMS node.** A data structure used by the ATMS to represent data asserted into the ATMS database. It consists of three components: datum, label, and justifications.

**Complete labels.** See Label completeness.

**Consistent environment.** An environment that does not contain any contradictory facts. A non-nogood environment.

**Consistent labels.** See Label consistency.

**Consumer, and the consumer architecture.** The consumer architecture is the problem solver of de Kleer's ATMS. It allows problem solving to take place in small inference steps. When an antecedent of an implication is satisfied by a set of facts in the database, a job called a consumer is placed on the problem solver's global agenda. Each consumer is assigned an ATMS node that is justified by the ATMS nodes of the facts that satisfied the antecedent pattern. A consumer contains the *detached* consequent of the implication formed by instantiating the consequent of the implication with the bindings generated from the antecedents. Problem solving is done by picking consumers off the agenda and running them. Running a consumer, in turn, triggers the creation of other consumers—and so on.



**Context.** The set of all facts that are true in given environment. Computed by finding all ATMS nodes whose label contain an environment that is a subset of the given environment.

**Contradictory equality.** An equality between two distinct non-Skolem constants.

**Datum.** A fact stored in an ATMS node or an equality node.

**Dependency structure.** A structure formed by the ATMS nodes that describes the dependencies among the data of the ATMS nodes. Pictorially, it forms a directed graph where the vertices of the graph correspond to the ATMS nodes, and the edges correspond to the justifications to and from the nodes.

**Derived ATMS node.** An ATMS node that is derived from other ATMS nodes, or justified by some other ATMS nodes.

**Derived environment.** An environment derived from two or more primitive environments. It is a set which consists of more than one assumption symbol or equality token. It is created when deriving new equalities and when updating labels.

**Derived equality.** An equality derived from two or more primitive equalities. It is created by the problem solver.

**Entry node.** An equality node of an existing equality fact that receives a new primitive environment as a new support. It is an input to the specialized ATMS's label-update algorithm that triggers the updating of the labels of other equality nodes in the same equivalence class.

**Environment.** A set of assumption symbols in de Kleer's ATMS or a set of equality tokens in the specialized ATMS.

**Equality database.** A database that stores equality nodes and equivalence class nodes of the specialized ATMS. It is maintained by the specialized ATMS's problem solver and by the label-update algorithm.

**Equality node.** A data structure used by the specialized ATMS to represent the equality fact asserted into the equality database. It consists of two components: datum and label.

**Equality token.** A globally unique name given to each and every ATMS environment introduced into the specialized ATMS's equality database—either through new equality assertions or as new support for an existing equality node label.

**Equivalence class.** A set of terms equal to one another in a single context. See also Weak equivalence class.

**Equivalence class node.** A data structure used by the specialized ATMS to represent the equivalence classes formed from the equality nodes. Its primary purpose is to let the problem solver and the label-update algorithm easily access the equality nodes in the equality database. An equivalence class node contains pointers to the terms and equality nodes that belong to that equivalence class.

**Extended version of the specialized ATMS.** The specialized ATMS with all of its features, plus the equality tokens and the optimization techniques.

**Hash array.** An Interlisp data structure that allows arbitrary lisp objects, the hash key, to be associated with other values, such that the value associated with a particular hash key can be quickly obtained.

**Inconsistent environment.** An environment that contains a contradictory fact. An environment subsumed by any nogood environment is also nogood, and therefore is inconsistent.

**Inconsistent label.** A label that contains an inconsistent environment. See **Label consistency**.

**Justification.** One of the three components of an ATMS node. Describes how an ATMS node is derivable from other ATMS nodes. Represented as a list of ATMS nodes.

**Label.** A set of environments associated with every ATMS node, as well as the equality nodes. The label describes the assumptions the datum ultimately depends on.

**Label completeness.** Guarantees that every context will contain every datum it should.

**Label computation.** A label computation occurs when a new environment is derived for a label through a disjoint union and the union steps of the specialized ATMS's label-update algorithm. In de Kleer's label-update algorithm, a label computation occurs for each label update attempt.

**Label consistency.** One of the four requirements imposed on the labels of the ATMS nodes. A label is consistent if it does not contain any nogood environments.

**Label minimality.** One of the four requirements imposed on the labels of the ATMS nodes. A label is minimal if none of the environments of the label is a superset of another environment of the label. A label is made minimal by performing a subsumption check during the label update process.

**Label-update algorithm.** An algorithm applied by both ATMSs when a label of an existing node is updated to include a new environment.

**Label update attempt.** A label update attempt occurs when an environment is taken intersection with one of the environments of an entry node label by the label-update algorithm of the specialized ATMS. In de Kleer's label-update algorithm, a label update attempt occurs when the newly derived environment of a label is discarded because it is subsumed by one of the label's existing environments.

**Minimal form.** See Label minimality.

**Multiple contexts.** A fact (or datum) exists under multiple contexts if it can be believed to be true under more than one set of assumptions. I.e., an ATMS node or an equality node with a label containing more than one environment.

**New support.** When a label of an existing ATMS node or equality node is given a new environment to be added to the label (since the fact of that node is now believed to be true under another environment), the new environment is called a new support. Adding a new support on a label initiates the label update process.

**Nogood.** An environment that supports a contradictory fact (or datum).

**Nogood database.** A special database used by de Kleer's ATMS to store the nogood environments.

**Ordinary constant.** Non-Skolem constants, such as numbers, atoms, and strings.

**Primitive environment.** An environment of a new equality fact given to the problem solver of both de Kleer's and the specialized ATMS, or a new supporting environment given to the label-update algorithm of both ATMSs.

**Primitive equality.** An equality fact given to the problem solver of both de Kleer's and specialized ATMSs.

**Problem solver.** The problem solver of de Kleer's ATMS (and for most truth maintenance systems) is the component that makes inferences about the domain. The consumer architecture is a problem solver for de Kleer's ATMS. In the specialized ATMS, the problem solver has a specialized task: to maintain the equivalence class nodes and derive new equality nodes.

**Reflexive axiom.** One of the three equality axioms:  $\forall x \quad x = x$ .

**Skolem constant.** A logical symbol denoting a constant value. Skolem constants are usually introduced to stand for unknown, but, constant values. For example, in the equation  $x^2 + 4 = 4$ ,  $x$  is a Skolem constant. Subsequent reasoning can determine that either  $x = 2$  or  $x = -2$ . Technically, a Skolem constant is a zero-argument Skolem function created when the existentially quantified variables are removed from a well-formed formula. Operationally, the difference between Skolem constants and ordinary constants is the following. When

two distinct ordinary constants are equated (e.g.,  $2 = 4$ ), we have a contradiction. When two distinct Skolem constants are equated (e.g.,  $x = y$ ), they are now constrained to denote the same constant value.

**Subsumed environment.** An environment of a label,  $l$ , is subsumed if it is a superset of any one of the other environments in  $l$ . See **Subsumption check**.

**Subsumption check.** A procedure that takes place during label update by the label-update algorithm of both ATMSs. When a new environment is computed for a label, if it is subsumed by the label's existing environment(s), the new environment is discarded. It is performed so that the labels can be kept in minimal form.

**Symmetry axiom.** One of the three equality axioms:  $\forall x, y \ x = y \supset y = x$ .

**Triangular equivalence class.** A weak equivalence class constructed from three terms and the equalities between those three terms. It is called such since it forms a triangular shape when the equivalence class is represented as a complete graph.

**Transitive axiom.** One of the three equality axioms:  $\forall x, y, z \ x = y \wedge y = z \supset x = z$ .

**Transitive closure.** Given a set of equivalence relations,  $S$ , a transitive closure of  $S$  is the set,  $S^+$ , that consists of all possible equivalence relations between each of the terms given in  $S$ . In the specialized ATMS, a transitive closure is computed by constructing a complete graph among the equalities, where the terms of the equalities correspond to the vertices of the graph, and the equalities correspond to the edges of the graph.

**Weak equivalence class.** A special kind of equivalence class employed by the specialized ATMS. It forms a maximal set of terms that are weakly equivalent. Two terms,  $t1$  and  $t2$ , are weakly equivalent if there exists an environment under which  $t1 = t2$  is true.