An Efficient ATMS for Equivalence Relations

Caroline N. Koff
Nicholas S. Flann
Thomas G. Dietterich
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

88-30-1

An Efficient ATMS for Equivalence Relations*

Caroline N. Koff
Nicholas S. Flann
Thomas G. Dietterich
Department of Computer Science
Oregon State University
Computer Science Building 100
Corvallis, Oregon 97331-3902
koff@cs.orst.edu

Topic: Automated Reasoning (equality reasoning, truth maintenance)
Word count: 4000 (approx.)

Abstract

We introduce a specialized ATMS for efficiently computing equivalence relations in multiple contexts. 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 size of the 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. 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 as been implemented as part of the logic programming language FORLOG.

*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. The authors thank Jim Holloway, Giuseppe Cerbone, Marion Hakanson, and Ritchey Ruff for their helpful comments on previous drafts of this paper.

* Submitted to AAAI-88

1 Introduction

Consider the following reasoning problem. Given equality assertions of the form $(= x\ y)$, where x and y are either Skolem constants or ordinary constants, compute the symmetric and transitive closure of the equality relation, detect contradictions, and answer queries of the form $(= x\ y)$. This problem has a long history in computer science, beginning with the need to reason about EQUIVALENCE and COMMON declarations in FORTRAN [Arden, Galler, & Graham, 1961]. The best known solution involves representing equivalence classes (sets of constants known to be equal to one another) as trees spanning from a chosen constant (the class representative) to the other members of the class. This yields the UNION-FIND algorithm [Galler & Fisher, 1964] and subsequent path compression optimizations [Aho, Hopcroft, & Ullman, 1974].

In this paper, we are interested in the case where the various equality assertions are labeled with supporting environments (sets of primitive assumptions) of the kind introduced by de Kleer's ATMS [1986a]. In this case, queries ask whether $(= x\ y)$ is true under some specified set of assumptions. This problem arises in any situation where equality assertions are present and there is a need to investigate multiple contexts (sets of assumptions) 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.

The remainder of this paper explores forward chaining approaches to solving this reasoning problem. First, the existing approaches, including UNION-FIND, are shown to be inefficient. Second, our solution is introduced with an algorithm description, an example problem, worst case and best case analysis, and a proof of correctness. Third, the algorithm is generalized and optimized. Finally, a brief summary is given.

2 Existing Approaches

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

2.1 Encoding the Equality Axioms

The simplest approach is to give the equality axioms direction to an ATMS-based problem solver. Only the transitive axiom must 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.

$$\forall x, y, z\, (= x\ y) \wedge (= y\ z) \supset (= x\ z) \tag{1}$$

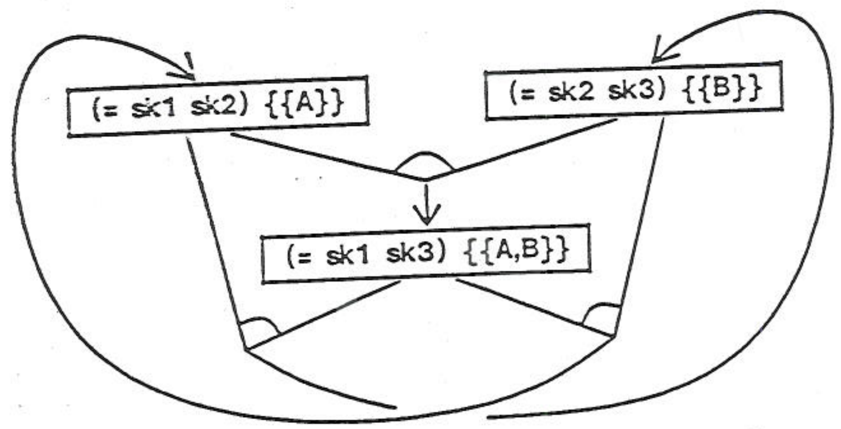

Figure 1: The dependency structure for three equalities

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 database during problem solving, a new assertion is derived and added to the ATMS database. For example, consider the following two equality assertions (presented using the basic ATMS data structure: node:⟨datum, label, justifications⟩):

$$node1 : \langle (= sk1\ sk2)\ \{\{A\}\}\ \{(A)\}\rangle \tag{2}$$

$$node2 : \langle (= sk2\ sk3)\ \{\{B\}\}\ \{(B)\}\rangle \tag{3}$$

These satisfy the antecedents of (1) and produce the following derived node:

$$node3 : \langle (= sk1\ sk3)\ \{\{A, B\}\}\ \{(node1, node2)\}\rangle \tag{4}$$

*Node*3 is the only new equality information derivable from *node*1 and *node*2. But by applying the reflexivity axiom, the newly derived equality in *node*3 will twice satisfy the antecedent of (1) in conjunction with the equalities in *node*1 and *node*2 respectively, and rederive the following two equalities:

$$node1 : \langle (= sk1\ sk2)\ \{\{A, B\}\}\ \{(node2, node3)\}\rangle \tag{5}$$

$$node2 : \langle (= sk2\ sk3)\ \{\{A, B\}\}\ \{(node1, node3)\}\rangle. \tag{6}$$

One of the requirements imposed on the labels by the ATMS is that they be in minimal form. Since the environment $\{A, B\}$ of (5) and (6) is subsumed by $\{A\}$ of *node*1 and $\{B\}$ of *node*2, it is not included in the labels of nodes *node*1 and *node*2. In this sense, the equality derivations of (5) and (6) are redundant. However these redundant derivations allow the problem solver to generate all of the necessary justification links for these three nodes. Without these justification links, the ATMS cannot apply its label-update algorithm correctly. The dependency structure for these three nodes is given in Figure 1. (Justifications for each equality assertion are shown as two links merging to support that assertion.)

Although de Kleer's label-update algorithm [de Kleer, 1986a] guarantees that the labels will be consistent and complete upon termination of the update process, each node may have been updated more than once. By applying this algorithm to a collection of mutually-supporting assertions, such as those shown in Figure 1, an alarming number of label update attempts will occur due to the circular structure of the dependencies. For example, suppose a node is given a new supporting environment. To propagate this environment to the rest of the nodes, the label-update algorithm will recursively update the consequent node labels by traversing justification links. Consider the following series of label update attempts made by the label-update algorithm after *node*1 has been updated to include the environment $\{C\}$ in its label. First, the algorithm attempts to update the labels of its consequent nodes, *node*2 and *node*3:

- For *node*2's label, the new environment of *node*1 and the environment of *node*3 are combined to produce the environment $\{A, B, C\}$, which is subsumed by $\{B\}$.
- For *node*3's label, the new environment of *node*1 and the environment of *node*2 are combined to produce the environment $\{B, C\}$ which is included in *node*3's label.

Since *node*3's label has changed, the algorithm will now attempt to update the labels of *node*3's consequent nodes, *node*1 and *node*2:

- For *node*1's label, the new environment of *node*3 and the environment of *node*2 are combined to produce the environment $\{B, C\}$, which is subsumed by $\{C\}$.
- For *node*2's label, the new environment of *node*3 and the environments of *node*1 are combined to produce the environments $\{A, B, C\}$ and $\{B, C\}$ both of which are subsumed by $\{B\}$.

The example given above does not demonstrate the worst case. This occurs when new support arrives on a derived node, such as *node*3—the algorithm must traverse every justification of every node. Since there are $\left(\begin{array}{c} n \\ 2 \end{array}\right)$, 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 $(n(n-1)/2)(n-2)$ or $\Theta(n^3)$. (The best case occurs when the algorithm terminates after attempting to update just one label upon either deriving a *nogood* 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 (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.

2.2 Extending UNION-FIND

The second approach is to employ some kind of equivalence class data structure like the UNION-FIND tree. 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 context systems like de Kleer's ATMS, the number of equivalence classes explodes. Suppose we have three equality assertions: $\langle(= a\ u)\{A\}\rangle$, $\langle(= a\ v)\{B\}\rangle$, and $\langle(= a\ w)\{C\}\rangle$. In this case, four non-trivial equivalence classes must be constructed: $\{a, u, v\}\{A, B\}$, $\{a, u, w\}\{A, C\}$, $\{a, v, w\}\{B, C\}$, and $\{a, u, v, w\}\{A, B, C\}$. If we only constructed the last class, we would not be able to answer the query $(= u\ v)\{A, B\}$ correctly. What is happening is that every distinct context gets its own equivalence class. Since there are 2^k contexts for k primitive assumptions, this results in an exponential explosion.

3 A Specialized ATMS for Equivalence Relations

Both of the approaches given above for implementing equality reasoning under multiple contexts are inefficient either because they construct explicit justification links or because they use the implicit justification structure of equivalence classes. The method described below avoids both of these problems 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.

3.1 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 $O(n^2)$ equality nodes in ED.

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 $t1$ and $t2$ are *weakly equivalent* if there *exists* an environment under which $(= t1\ t2)$ is true. Note that the environment in question 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 $t1$ to node $t2$ asserts that $(= t1\ t2)$. Figure 2 shows the equivalence class of Figure 1 using this notation. The edges are labeled with the labels for the corresponding equality nodes.

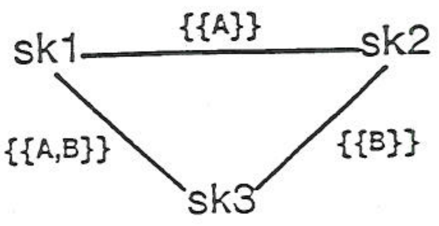

Figure 2: The equivalence class from Figure 1 in the new notation

3.2 The Problem Solver

The problem solver of the specialized ATMS is given equalities of the form $(= t1\ t2)$ with their corresponding labels. Its task is to create and maintain equivalence class nodes by deriving 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 distinct.

For the purpose of describing how the new equality nodes are derived, let Eq be the primitive equality $(= t1\ t2)$, 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 *Combine-Labels* be a function which takes two labels, $l1$ and $l2$, and produces a new label by putting in a minimal form the set of environments env_{xy}, where $env_{xy} = \{env_x \cup env_y \mid env_x \in L1 \wedge env_y \in L2\}$.

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, 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 $t1 \in EC_1$, but $t2$ does not exist in ED. Let $EC_1' = EC_1 - \{t1\}$. Then $\forall t_i \in EC_1'$, for $i = 1 \ldots n_1 - 1$, 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 for $(= t1\ t_i)$). Then, create and assert into ED an equality node for Eq and l_{Eq}, and add $t2$ to EC_1.
Case 3: Suppose $t1 \in EC_1$ and $t2 \in EC_2$. Let $EC_1' = EC_1 - \{t1\}$, and $EC_2' = EC_2 - \{t2\}$. Then $\forall t_i \in EC_1'$, for $i = 1 \ldots n_1 - 1$, and $\forall t_j \in EC_2'$, for $j = 1 \ldots n_2 - 1$, 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 for $(= t1\ t_i)$, label for $(= t2\ t_j)$)).
- An equality node with $(= t2\ t_i)$ and its label computed as:
  Combine-Labels(L_{Eq}, label for $(= t1\ t_i)$).
- An equality node with $(= t1\ t_j)$ and its label computed as:
  Combine-Labels(L_{Eq}, label for $(= t2\ t_j)$).

Hence, the number of new equalities derived from joining EC_1 and EC_2, 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 EC_1 to be $EC_1 \cup EC_2$.
Case 4: When $t1, t2 \in EC_1$, the label-update procedure is called, since Eq is providing new environment(s) to be added to the existing label of Eq.

Table 1: Summary of the label-update process

i	j	env_{ij}	Result of $(env_{ij} \cap \{A\})$	Result of $((env_{i,j} \oplus \{A\}) \cup \{D\})$
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	$\{A, B, C\}$	$\{A\}$	$\{B, C, D\}$

While deriving new equality nodes, if the problem solver detects a derived equality between two different (non-Skolem) constants (a contradiction), its label is declared *nogood* (see [Koff, 1988]).

3.3 The Label-Update Algorithm

3.3.1 The Algorithm

The label-update procedure is given an existing equality node, called the *entry node*, along with a new environment, env_{new}. Its task is to add this new environment to the existing label, l_{old}, of the entry node and to update all the labels of the other equality nodes in the equivalence class. Let $L_{updates}$ be the set of all labels in the equivalence class containing the entry node, but not including l_{old}. The procedure is as follows:

```
• For each $l_i \in L_{updates}$ do:
    • For each $env_{i,j} \in l_i$ do:
        • For each $env_{old,k} \in l_{old}$ do:
            1. If $(env_{old,k} \cap env_{i,j}) = \emptyset$, do nothing.
            2. Else, compute a new environment to be added to $l_i$ as:
                • $(env_{old,k} \oplus env_{i,j}) \cup env_{new}$[1]
                • If the newly computed environment is not
                  subsumed by the environments in $l_i$ then add it to $l_i$.
```

3.3.2 An Example

Consider the equivalence class shown in Figure 3. Suppose new environment $\{D\}$ arrives on the label for $(= sk1\ sk2)$. The updated label for this equality is $\{\{A\}, \{D\}\}$, and $env_{old,1}$ is $\{A\}$ and env_{new} is $\{D\}$. The other labels in the equivalence class shown in Figure 3 are updated as prescribed by the label-update algorithm given above. The results of applying the steps are summarized in Table 1. The updated equivalence class of Figure 3 is shown in Figure 4. Note that in this example the algorithm did not compute any redundant environments.

[1] $\oplus$ is the disjoint union operation defined as: $A \oplus B = (A - B) \cup (B - A)$.

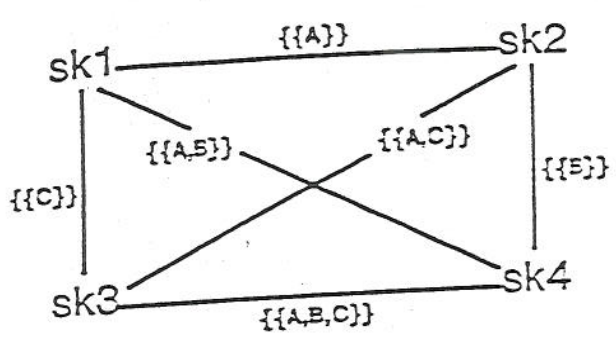

Figure 3: Before the label updates

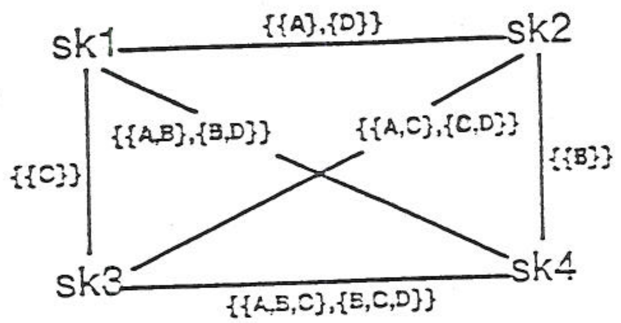

Figure 4: After the label updates

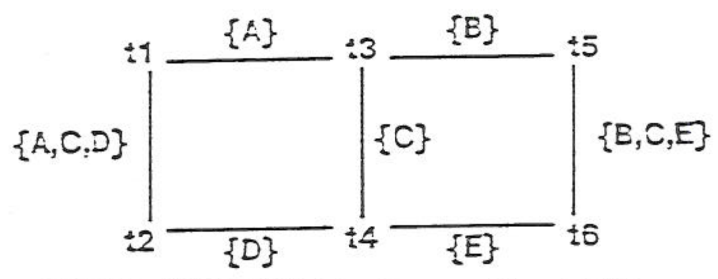

Figure 5: Shared support and label updates

3.3.3 An Explanation

To see why this algorithm succeeds, consider Figure 5, which shows a portion of an equivalence class. All of the equalities with singleton environments are primitive (given) assertions. Let us focus on the two derived equalities $(= t1\ t2)$ and $(= t5\ t6)$. Notice two things. First, the graphical counterpart of the transitivity axiom is a connected path. To compute the environment for $(= t5\ t6)$, we find a 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 $\{B, C, E\}$. 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 will, among other things, update the label for $(= t5\ t6)$ to include the environment $(\{A, C, D\} \ominus \{B, C, E\}) \cup \{F\} = \{A, B, D, E, F\}$. This can be viewed as (a) subtracting the path shared by the two equalities $(= t1\ t2)$ and $(= t5\ t6)$ and (b) computing a new path, $\langle t5, t3, t1, t2, t4, t6\rangle$, that passes through the newly supported equality $(= t1\ t2)$. 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 can be performed without explicitly traversing paths or justification links, since the labels implicitly hold the dependency structure.

It is for this reason that when applying this algorithm, the nogood environments cannot be removed from the labels until they can be replaced with a new non-nogood environment (see [Koff, 1988]).

3.3.4 Computational Costs

Since there are $n(n-1)/2$ 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 $(env_{old,k} \cap env_{i,j}) = \emptyset$ may be true). In fact, it can be shown that in the best case, only $\Theta(n)$ label computations will be performed [Koff, 1988].

3.3.5 Proof of Correctness

We demonstate 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:

$$env3 = env1 \ominus env2 \tag{7}$$

$$env2 = env1 \ominus env3 \tag{8}$$

$$env1 = env2 \ominus env3 \tag{9}$$

We now prove that for any triangle in an equivalence class, equations 7, 8 and 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 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 7, 8, and 9 must hold for each new triangle added. Hence, by induction, the relationships of 7, 8, and 9 hold for all triangles in an equivalence class.

Now consider a new support $env1_{new}$ arriving on equality $Eq1$, where $Eq1$ is in an equivalence class of size n. $Eq1$ will form an edge of $n-1$ triangles with each of the edges termed $Eq2_i$ and $Eq3_i$. To update this equivalence class, the labels of $Eq2_i$ and $Eq3_i$ for each of the triangles will be 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:

$$env2_{new} = env3 \cup env1_{new} \tag{10}$$

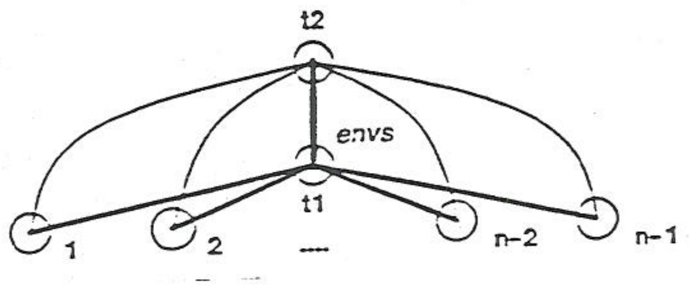

Figure 6: Incremental extension of an equivalence class

$$env3_{new} = env2 \cup env1_{new} \tag{11}$$

From 7 we can substitute into 10, and from 8 we can substitute into 11 to obtain the following two equations:

$$env2_{new} = (env1 \ominus env2) \cup env1_{new} \tag{12}$$

$$env3_{new} = (env1 \ominus env3) \cup env1_{new} \tag{13}$$

The equations 12 and 13 directly correspond to the disjoint union and union step of the label-update algorithm. Hence, we have shown that the algorithm behaves correctly.

4 Extending the Method

It is clear from the proof given above that the label-update algorithm will behave incorrectly if any of the incoming environments are not unique, since the disjoint relationship will not hold among environments in an equality triangle. To accommodate non-unique environments, incoming environments are made unique by an equality token mechanism described below.

4.1 Equality Tokens

Uniqueness can be guaranteed 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. Under this design, label updates, as well as the computation of labels for the newly derived equalities, will be done on environments containing equality tokens, not ATMS assumptions.

For example, suppose two equality assertions $(= sk1\ sk2)$ with $\{A, B\}$ and $(= sk2\ sk3)$ with $\{B, C\}$ are given to the problem solver. Then, the following renaming, denoted as $\rightarrow$, will occur: $\{A, B\} \rightarrow \{1\}$, and $\{B, C\} \rightarrow \{2\}$. The derived equality node $(= sk1\ sk3)$ will have $\{\{1, 2\}\}$ as its label instead of $\{\{A, B, C\}\}$. When the 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 determining *nogoods* and queries into the equality database.[2] The mapping

[2]The translation will also be necessary during label updates if the specialized ATMS is linked to the standard ATMS.

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.

4.2 Optimization

Although the label-update algorithm for the specialized ATMS produces fewer redundant environments, as equality tokens, than de Kleer's label-update algorithm, it must still perform the subsumption check for every equality token environment it produces, since it cannot guarantee that they are in minimal form. However, there are certain cases where the subsumption checks can be skipped because the derived environments are guaranteed to be non-redundant.

Suppose an entry node N which contains a singleton environment (primitive environment) $\{T_{old}\}$, within its existing label, is given the environment $\{T_{new}\}$. All of the other labels in the same equivalence class as N can be *fully* updated 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 same inferences needed for T_{new}. Therefore T_{new} may simply replace T_{old}.

Consider the alternate case in which the entry node, N, contains only derived (non-singleton) environments within its label. Suppose it is given the environment $\{T_{new}\}$, as an initial update. If, during the label update process, we encounter a node M whose label contains a singleton environment $\{T_{old}\}$, we can completely update M's label by only considering $\{T_{old}\}$ in combination with the existing environments of N. We do not need to consider the other environments 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.

5 Summary

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 2.1):

- 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 certain cases.

- The problem solver that derived two redundant equalities for every new equality derived has been replaced by one that only derives the necessary equalities.

- The space required to store the justification links is eliminated.

The specialized ATMS has been implemented as a part of the equality system for FOR-LOG [Flann, et al., 1987] and interfaced with the standard ATMS and the consumer architecture.

6 References

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,* :24-29, 1987.

Galler, B. A., and Fisher, M. J., 1964. An Improved Equivalence Algorithm. *Comm. ACM,* 7 (5), pp. 301-303.

Koff, C., 1988. Forthcoming. An Efficient ATMS for Equivalence Relations. M.S. Thesis, Department of Computer Science, Oregon State University.

McAllester, D., 1982. Reasoning Utility Package User's Manual. Artificial Intelligence Laboratory, AIM-667, MIT, Cambridge, MA.