Differencing and Merging of Architectural Views

Marwan Abi-Antoun Jonathan Aldrich Nagi Nahas
Bradley Schmerl David Garlan

August 2005
CMU-ISRI-05-128

Institute for Software Research International
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

As architecture-based techniques become more widely adopted, software architects face the problem of reconciling different versions of architectural models. However, existing approaches to differencing and merging architectural views are based on restrictive assumptions, such as requiring view elements to have unique identifiers or explicitly log changes between versions.

To overcome some of the above limitations, we propose differencing and merging architectural views based on structural information. To that effect, we generalize a published polynomial-time tree-to-tree correction algorithm (that detects inserts, renames and deletes) into a novel algorithm to additionally detect restricted moves and support forcing and preventing matches between view elements. We implement a set of tools to compare and merge component-and-connector (C&C) architectural views, incorporating the algorithm. Finally, we provide an empirical evaluation of the algorithm and the tools on case studies with real software, illustrating the practicality of the approach to find and reconcile interesting divergences between architectural views.

This work was supported in part by NASA cooperative agreements NCC-2-1298 and NNA05CS30A, NSF grant CCR-0204047, a 2004 IBM Eclipse Innovation Grant, the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems", and was performed as a joint research project in Strategic Partnership between Carnegie Mellon University and Jet Propulsion Laboratory.

Keywords: View differencing, view merging, view synchronization, tree-to-tree correction, unordered labeled trees.

1. INTRODUCTION

The software architecture of a system defines its high-level organization as a collection of runtime components, connectors and constraints on their interaction, along with their additional properties defining the expected behavior, commonly referred to as a component-and-connector (C&C) view. Over the past decade, numerous architecture description languages (ADLs) have been developed and applied to real-world systems.

As architecture-based techniques become more widely adopted, software architects face the problem of reconciling different versions of architectural models, including differencing and sometimes merging architectural views— i.e., using the difference information from two versions to produce a new version that includes changes from both earlier versions. For instance, during analysis, a software architect may want to reconcile two C&C views representing two variants in a product line architecture [CCG+03]. Once the system is implemented, an architect may want to compare a high-level conceptual C&C view with a C&C view retrieved from the implementation (using a variety of architectural recovery techniques): the architect might be interested in implementation-level violations of the architectural styles or other intent [AAG05], or in a change impact analysis [KPS+99]. At runtime, the difference information could be used to perform architectural repair [DHT02]. Finally, during evolution, the architect may use the difference information to better focus regression testing efforts [MDR05].

A number of techniques and tools for differencing and merging C&C views have been proposed. Some of these techniques detect only a small number of differences. For instance, ArchDiff [CCG+03] only detects insertions and deletions, possibly leading to the loss of information when elements are moved or renamed. Many of these techniques are also limited in their ability to detect differences based purely on structural information; they assume that elements have unique identifiers (every time an element is changed, even when only its type changes, it gets a new unique identifier [AP03][OWK03]), or only match two elements if both their labels and their types match [CCG+03]. Other approaches (e.g., Mae [RHM+04]) rely on the environment tracking all changes using fine-grained element-level versioning. Although such environments may provide the ability to infer high-level operations such as merges, splits or clones, in addition to the low-level operations such as inserts and deletes, they require a heavy upfront investment in tool building and integration, and have not become widely adopted. Similarly, one can maintain a record of the structural changes introduced to a view and replay it against another view [Jim05].

In this paper, we propose an approach that overcomes some of the above limitations. Our main contributions are:

- An approach for differencing and merging two architectural views based on structural information, using tree-to-tree correction algorithms to identify matches and classify the changes between the two views. Optional type information can prevent matches between incompatible view elements, speeding execution and improving the quality of the output.

- A generalization of a recently published tree-to-tree correction algorithm for unordered labeled trees [THP05] (that detects renames, inserts and deletes) into a novel polynomial-time tree-to-tree correction algorithm that additionally detects restricted moves and supports forcing and preventing matches between view elements.

- A set of tools incorporating such algorithms for the semi-automated synchronization of C&C views.

- An empirical evaluation of the algorithms and the associated tools on realistic case studies.

The paper is organized as follows. Section 2 describes the challenges in differencing and merging structural views, the underlying assumptions and the limitations of our approach. Section 3 describes our novel tree-to-tree correction algorithm. Section 4 describes tools that incorporate tree-to-tree correction algorithms to synchronize C&C views. Sections 5 and 6 present two case studies on real systems. Finally, we discuss related work and conclude.

2. CHALLENGES

A view can generally be described as a graph. View differencing and merging can then be cast as a problem in graph matching. Hierarchical architectural views have aspects of both graphs and trees—i.e., they have a tree-like hierarchy but there are cross-links that form a general graph. In this section, we consider the benefits of both graph and tree differencing approaches, with graph algorithms being more general, but tree algorithms more scalable. Having chosen trees for scalability, we describe a new algorithm in the next section that meets our requirements.

2.1 Differencing and Merging

Graph matching, in the general case, is NP-complete [CFS+04]. However, certain classes of graphs do not suffer from the exponential complexity. For instance, graphs characterized by the existence of unique node labels can be processed efficiently [DBB+04]. In addition, efficient algorithms have been proposed for trees. A widely used measure of the similarity between two graphs is the notion of graph edit distance [CFS+04]. The approach relies on using a set of edit operations that model inconsistencies by transforming one graph into another. Typical graph edit operations include the deletion, insertion and substitution of nodes and edges. Often a cost is assigned to each edit operation. The costs are application dependent and used to model the likelihood of the corresponding inconsistencies (typically, the more likely a certain inconsistency is to occur, the lower is its cost). If a cost is assigned to each edit operation, then the edit distance of two graphs g_1 and g_2 is found by searching for the sequence of edit operations with the minimum cost that transform g_1 into g_2. A similar problem formulation can be used for trees; however, tree edit distance differs from graph edit distance in that operations are carried out only on nodes and never directly on edges. In Section 3, we describe a novel algorithm based on tree edit distance that meets the requirements of the problem domain.

2.2 Assumptions

Before we do that, we discuss some of the assumptions in our approach and how they generalize those of existing approaches.

No Unique Identifiers. For maximum generality, we match elements based on their structure and do not require elements to have unique identifiers, as in ArchDiff. In many applications, such unique identifiers do not exist. Adding this assumption gives the problem of graph edit distance a polynomial-time complexity, as recently shown in [DBB+04]. As an optimization, persistent unique identifiers could be assigned to view elements to quickly match them between invocations.

No Ordering. In the general case, an architectural view has no inherent ordering among its elements. Assuming an architectural view is represented as a tree, this suggests that an unordered tree-to-tree correction algorithm might perform better than one for ordered trees. Ordered labeled trees (i.e., rooted trees in which the children of each node are ordered) have been studied extensively with many efficient algorithms available (e.g., [SZ97]). However, tree-to-tree correction for unordered trees is MAX SNP-hard [ZJ94]. Some algorithms for unordered trees achieve polynomial-time complexity, either through heuristic methods (e.g., [WDC03][CG97]) or through an exact solution under additional assumptions (e.g., [THP05]).

Support Disconnected/Stateless Operation. For maximum generality, we assume a disconnected and stateless operation, i.e., no monitoring of structural changes is taking place while the user is modifying a given view (e.g., Mae [RHM+04]) and no trace is kept of the set of changes made to a view (e.g., [Jim05]).

Detect Renames. For maximum generality, we do not require labels to match exactly. Names are often modified during software development and maintenance: a name may turn out to be inappropriate or misleading due to either careless initial choice or name conflicts from separately developed sub-systems [AC94]. In some application domains, some view elements may not have persistent names or may be assigned automatically generated names. This suggests that the algorithms should be able to handle sparse or incomplete labels and handle renames. A number of existing algorithms detect renames, but either assume that a strong majority of nodes will have exactly matching semantic information (labels and types) or have only been tested under such a condition: e.g., at least 80% of nodes have exactly matching semantic information in [CG97], and at least 99% of nodes have exactly matching semantic information in [RRL+04].

Detect Hierarchical Moves. Architects often use hierarchy to control complexity, and many views are hierarchical: e.g., in C&C views, the hierarchy corresponds to the system's decomposition. However, architects differ in their use of hierarchy: components expressed at the top level in one view could be nested within another component in some other view. A hierarchical move shifts a node up or down N levels in the tree, changing its parent. The ability to detect hierarchical moves is one of the main features which distinguish our proposed algorithm from the algorithm described in [THP05].

Allow Manual Overrides. Since having a correct mapping between view elements is critical for the merge operation, user control over the structural matching process is important: in particular, the user should be able to force a match between elements that cannot be structurally matched, as well as prevent matches between elements that, although structurally similar, are in fact incompatible. Note that manual overrides must be taken into account by the algorithm itself, and cannot happen as a post-processing step since there are dependencies in the mapping (e.g., two view elements a_1 and a_2 in View A may not both

map to the same element b_1 in view *B*, even if a_1 is forced to match b_1). This feature also distinguishes our algorithm from existing ones.

Type Information for Optimization Only. Unlike other approaches (e.g., ArchDiff), matching the type information is not critical to the operation of the algorithm; it should be able to deal with views containing untyped elements, as well as views at different levels of abstraction with possibly different type systems. The algorithm should be able to recover a correct mapping from structure alone if necessary, or structure and type information if type information is available. However, the algorithm can take advantage of the type information (when available) to prune the search tree, significantly speed convergence towards the optimal solution and improve the quality of the matching. If the view elements are represented as typed nodes, at the very least, the algorithm should not match nodes of incompatible types (e.g., do not match connector *x* to component *y*). In some cases, additional architectural type information may be available and could be used for similar purposes (e.g., do not match a component of type *Filter* from a Pipe-and-Filter style to a component representing a *Repository* from a Shared Data style).

In order to remain tractable, our approach makes the following restricting assumptions:

Hierarchical Views. In the general case, the differencing and merging of non-hierarchical views corresponds to *error-correcting* or *inexact subgraph isomorphism* [CFS+04], a problem proved to be NP Complete. The most ambitious *optimal* algorithms (i.e., if a global minimum of the matching cost exists, it will be found) can handle at most a few dozen nodes. We take advantage of the tree hierarchy in architectural views and recast the problem into one that is more tractable, using trees instead of graphs. In C&C views, hierarchy corresponds to nested sub-architectures or decomposition. Other architectural views, such as module views [CBB+03], have similar characteristics.

Similar and Comparable Views. The two views being compared and merged have to be somewhat structurally similar. When comparing two completely different views, the algorithm could produce a trivial edit script that deletes all elements of one view and then inserts all the elements in the other view. In addition, the two views being compared and merged must be of the same type, i.e., comparable without any view transformation. This also allows the approach to be more applicable than just C&C views, at least in principle.

Merging/Splitting Not Supported. Our approach does not currently detect the merging or splitting of view elements.

3. TREE-TO-TREE CORRECTION

In this section, we describe in detail a novel tree-to-tree correction algorithm for unordered labeled trees. The reader only interested in its applications can skim this section. Our TreeMDIR (Tree Move-Delete-Insert-Rename) algorithm generalizes a recently published algorithm [THP05], denoted as THP. We also implemented THP for experimental comparison with our implementation of TreeMDIR.

3.1 Problem Definition

Let us first give an unambiguous definition of the problem, adapted from [SZ97]. We denote the i^{th} node of a labeled tree T in the postorder node ordering of T by T[i]. |T| denotes the number of elements of T. We define a triple ($\mathcal{M}$, T_1, T_2) to be a mapping from T_1 to T_2, where $\mathcal{M}$ is any set of pairs of integers (i,j) satisfying:

1) $1 <= i <= |T_1|$, $1 <= j <= |T_2|$;

2) For any pair of (i_1,j_1) and (i_2,j_2) in $\mathcal{M}$,

a) $i_1 = i_2$ if and only if $j_1 = j_2$ (one-to-one)

b) $T_1[i_1]$ is an ancestor of $T_1[i_2]$ if and only if $T_2[j_1]$ is an ancestor of $T_2[j_2]$ (ancestor order preserved).

We will use $\mathcal{M}$ instead of ($\mathcal{M}$,T_1,T_2) if there is no confusion. To delete a node N in tree T, we remove node N and make its children become the children of the parent of N. To insert a node N in tree T as a child of node M, we make N one of the children of M, and we make a subset of the children of M become children of N (See Figure 1). Renaming a node only updates its label. In the following discussion, a matched node means a node with an exactly matching label or a renamed node. The edit operations that we refer to as restricted moves correspond to deletion and insertion operations in the middle of

the tree: sequences of node deletions in the middle of the tree result in nodes moving up a number of levels in the hierarchy, and sequences of node insertions in the middle of the tree result in nodes moving down in the hierarchy (by becoming children of the inserted nodes). TreeMDIR does not currently support arbitrary node moves. THP does not allow any insertions or deletions in the middle of the tree and works under the assumption that if two nodes match, so do their parents (i.e., only subtrees can be inserted or deleted).

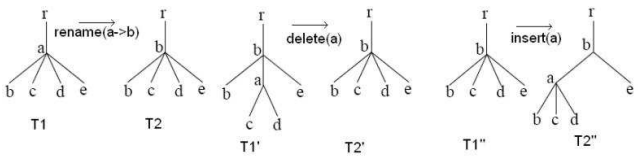

Figure 1: Edit operations in tree-to-tree correction [SZ97].

Suppose we obtain a mapping $\mathcal{M}$ between trees T_1 and T_2. From this mapping we can deduce an edit script to turn T_1 into T_2. First, we flag all unmatched nodes in the first tree as deleted and all unmatched nodes in the second tree as inserted. We order the operations so that all deletion operations precede all insertion operations, delete the nodes in order of decreasing depth (deepest node first), and insert them in increasing depth order.

We still have to define the cost of an edit script (which is a sequence of edit operations): for each node in the source tree, we choose a cost of deletion (not necessarily the same for all nodes); for each node in the destination tree we choose a cost of insertion (again, not necessarily the same for all nodes), and for each pair of nodes *(n, m)* where *n* is some node in T_1 and *m* in T_2, we choose a cost of changing the label of *n* into the label of *m* (for example, to change *"banana"* into "*ananas"*, we might choose a cost of two using string-to-string correction [WF74]). The cost of the edit script is then equal to the sum of the costs of insertion, deletion, and renaming operations it contains. Therefore, any given mapping has a unique cost. So, in order to find an optimal edit sequence, it is sufficient to find an optimal mapping.

3.2 Explanation of the Algorithm

The algorithm pseudocode is given in Section 3.3 below. Let *C(i,j)* be the cost of the optimal mapping from the subtree rooted at *i* to the subtree rooted at *j*. A set of nodes *S(i)* is a successor set of node *i* if it is a subset of the set of descendents of *i* and none of the elements of *S(i)* is an ancestor of another, and each node of the subtree rooted at *i* is either a descendent or an ancestor of an element of *S(i)*. Given two sets *S(i)* where *i* belongs to T_1, and *S(j)* where *j* belongs to T_2, it is possible to define the optimal mapping of *S(i)* to *S(j)* as a one to one function from a subset of *S(i)* into *S(j)* with least cost, where the cost of mapping element *k* of *S(i)* to element *l* of *S(j)* is equal to cost of the optimal mapping of the subtree rooted at *k* to the subtree rooted at *l*, and the cost of leaving an element *k* of *S(i)* without image is equal to the cost of deleting the whole subtree rooted at *k*, and the cost of having an umatched element *l* in *S(j)* is equal to the cost of inserting the entire subtree rooted at *l*. This suggests that if we know all the costs $C(d_1,d_2)$ where d_1 is a descendent of *i* and d_2 is a descendent of *j*, it is possible to compute *C(i, j)* by considering all possible pairs of sets *(S(i),S(j))*, and for each such pair, getting the minimum weight bipartite matching defined by the entries of the cost matrix *C* corresponding to the elements of *S(i)* and *S(j)*. Finally, let *L(i,j)* be the cost of changing the label of node *i* in the source tree to the label of node *j* in the destination tree. The minimum cost obtained added to *L(i, j)* will be equal to *C(i, j)*. *L(i,j)* uses string-to-string correction to evaluate the intrinsic degree of similarity between the labels of two nodes, using the standard dynamic programming algorithm to find the longest common subsequence [WF74].

We choose the best pair *(S(i),S(j))* using a *branch-and-bound* backtracking algorithm. Let *DESC(i)* denote the set of descendents of *i*. We try to choose a subset *Q* of *DESC(i)*x*DESC(j)* with minimal cost. This is done by trying to add to *Q* one element of *DESC(i)*x*DESC(j)* such that the new element in *Q* is consistent with the previous elements (no same node can be matched to 2 different nodes, nor can a node appear in an element of *Q*, if either a descendent or an ancestor already appears in some element of Q). The algorithm backtracks each time it determines that there are no more valid pairs to add, or when it determines that the cost of the current branch will be too large to match the best solution already discovered to date. As the problem is NP-complete, the approach outlined above can quickly become computationally infeasible without additional constraints.

We chose to enforce an upper bound *B* on the sum of distances between elements of *S(i)* and the closest child of *i* (respectively, *S(j)* and *j*) with *B* typically a small integer. The reasoning behind this constraint is that nodes are not usually moved too far from their original positions in a hierarchy, and it is relatively rare for several non-leaf siblings to be deleted at the same time. The bound *B* has the additional benefit that only relatively small neighborhoods of each node have to be considered for the computation of the optimal cost of a single subtree pair, enabling us to perform many operations very

efficiently using bit manipulation. For example, during the backtracking search, checking whether a node is still available is a single bitwise AND operation instead of a time-consuming loop over an array.

TreeMDIR can be considered a generalization of THP because THP only handles the case where *B=0* (i.e., only the children of a node can be in a successor set of that node), producing a fully polynomial time algorithm that is typically much faster than our generalized algorithm. But being able to handle non-zero values of B allows our algorithm to detect hierarchical moves. TreeMDIR is guaranteed to find the optimal matching within the constraints of the bound *B*, provided it is allowed to run long enough. Unfortunately, on a number of instances (especially, on trees with more than a few hundred nodes and when the average degree of a non-leaf node is greater than four), it is necessary to limit the running time by enforcing a bound *R* on the number of recursive calls of the backtracking search corresponding to a given subtree pair. This bound removes the guarantee of optimality. Nevertheless, we found that the algorithm still obtains good results when we limit the number of recursive calls, because usually the backtracking search finishes very quickly when we compare similar subtrees. Since the algorithm uses the branch-and-bound technique, a good match allows for tight bounds and therefore early cutting of branches. The search terminates normally for matrix entries actually corresponding to good matches, and is interrupted only when the match is not good, which often allows the algorithm to return an optimal match even though the backtracking search was interrupted for the computation of some of the cost matrix entries (as these matrix entries correspond to bad matches which are not part of the optimal solution).

3.3 Pseudo Code of the Algorithm

In the following pseudo code of the TreeMDIR algorithm, arguments that are passed by reference are indicated by **ref**. In order to reduce the complexity of the pseudo-code, the parameter *R*, and the ability to force and prevent matches are not reflected here. For efficiency reasons, bit vectors are stored in integers (with 0 meaning **false**, and 1 meaning **true**) in and bitwise manipulations are used heavily.

```
Procedure: TreeMDIR // Main procedure
Input:
Tree T1: first tree to compare
Tree T2: second tree to compare (turn T1 into T2)
Output:
BestGlobalMatch: contains the best mapping from T1 to T2
Declare:
CostMatrix: CostMatrix[i][j] is the cost of the optimal mapping from the subtree rooted at i
            to the subtree rooted at j
BestGlobalMatch[]: array of pairs of nodes corresponding to the least cost mapping from T1 to T2
BestSuccessor[][]: a 2D array of sets of pairs of nodes
                   (m,n)∈ BestSuccessor[i][j] means (m,n) is a match between one element of the
                   successors of i and one element of the successors of j in an optimal mapping
                   from the subtree rooted at i to the subtree rooted at j
L(i,j): cost of changing the label of node i in T1 to the label of node j in T2
        using string-to-string correction
Begin
Postorder T1 and T2 nodes
for(i = 1 to T1.size)
  for(j = 1 to T2.size)
    BestSuccessor[i][j] = Search(i, j, ref CostMatrix)
    CostMatrix[i][j]= BestSuccessor[i][j].cost + L(i,j)
GetBestMatching(BestSuccessor, ref BestGlobalMatch,
            T1.size, T2.size)
End

Procedure: Search // Setup data structures for calling Backtrack
Input:
i: index in tree T1
j: index in tree T2
CostMatrix: cost matrix, same as for TreeMDIR
Output:
CostMatrix[i][j]: updated entry in the cost matrix
return a set of node pairs representing the best found mapping of the nodes of a successor set of i to
the nodes of a successor set of  j
Declare:
Asc1[], Des1[]: arrays of integers where the nth bit in the  mth integer indicates whether mth node is an
                ascendant (respectively, descendent) of nth node in T1
Asc2[], Desc2[]: same as above, for T2
BestSolution[]: set of optimal matches, implemented as a Boolean array: nth entry is true if the nth node
                pair in the set of all node pairs sorted by merit belongs to the best matching
                (merit is a measure of the quality of the matching)
CurrentSolution[]: set of matches being built, encoded in the same way as BestSolution[]
BestCost: variable
```

```
Unavailable1: integer where the nth bit is set if the nth node in tree T1 is unavailable for inclusion in
              CurrentSolution because an ascendant or descendent is already included in CurrentSolution
Unavailable2: same as Unavailable1 but for tree T2
Begin
Get the list L of all pairs (p,q) where p is a descendent of i and q is a descendent of j
Sort the list by decreasing match merit
   (merit represents the percentage of subtree weight that is matched when two nodes are compared)
foreach node among the descendents of i and j
  Associate an integer. Make the bit sequence correspond
  to the set of of descendents/ascendents of the nodes
  Store the integers in the Desc/Asc arrays, respectively
  Initialize BestSolution and CurrentSolution arrays to 0
  Initialize BestCost to an infinite value
  Initialize Unavailable1 to 0, Unavailable2 to 0
  BACKTRACK(0  /* index*/, L, Asc1, Asc2, Desc1, Desc2, Unavailable1,Unavailable2, CostMatrix,
          0 /* CurrentCost*/, ref BestCost, ref BestSolution, ref CurrentSolution)
Convert BestSolution bit vector to a set of node pairs
return set of node pairs
End

Procedure: BACKTRACK //SEARCH FOR A GOOD MAPPING BETWEEN SUBTREES
Input:
index: position reached in list L
L: list of pairs of nodes (m,n) sorted by merit
Asc1[], Asc2[], Desc1[], Desc2[]: same as for SEARCH
Unavailable1, Unavailable2: same as for SEARCH
CostMatrix: cost matrix, same as for SEARCH
CurrentCost: current cost of the mapping being built
  (i.e., the subset of Cartesian product of the set of
   descendents of i and j)
ref BestCost: same as for SEARCH
ref BestSolution[]: same as for SEARCH
ref CurrentSolution[] : same as for SEARCH
Output: BestCost, BestSolution, CurrentSolution: updated
Begin
Base Case:
  if (no element of L can be added to CurrentSolution)
   if (CurrentCost + cost of deleted subtrees < BestCost)
        BestSolution = CurrentSolution
        BestCost = CurrentCost
    return
foreach element l= (m,n) in L starting at position index
  Check whether l.first and l.second are still available
  if not continue
  if ( adding l to current mapping violates bound B )
    continue
  Add cost of match to CurrentCost to obtain NewCost
  Get a lower bound E of remaining cost using match merit
      If ( E + NewCost >= BestCost ) continue
  Add l to CurrentSolution (by setting the corresponding
    entry in CurrentSolution to l)
  NewUnavailable1 = Unavailable1 OR Desc1(m) OR Asc1(m)
  NewUnavailable2 = Unavailable2 OR Desc2(n) OR Asc2(n)
  BACKTRACK(index+1, L, Asc1, Asc2, Desc1, Desc2,
    NewUnavailable1, NewUnavailable2, CostMatrix,
    NewCost, BestCost, BestSolution,ref CurrentSolution);
  Remove l from CurrentSolution
End

Procedure: GETBESTMATCHING // DEDUCE THE OPTIMAL MAPPING
Input:
BestSuccessor[][]: same as for TREEMDIR
ref BestGlobalMatch[]: same as for TREEMDIR
i, j: indices of a pair of nodes that belong to the best possible mapping between the two trees
Output: BestGlobalMatch: updated
Begin
foreach e = (m, n) in BestSuccessor[i][j]
  Add e to to BestGlobalMatch
  GETBESTMATCHING(BestSuccessor, ref BestGlobalMatch, m, n)
End
```

3.4 Forcing and Preventing Matches

Manual overrides are not a standard operation in most tree-to-tree correction algorithms. We added to TreeMDIR the ability to force and prevent matches between a node in tree T_1 and another node in tree T_2. Preventing a match between two nodes *i* and *j* is easy—just assign a very large cost to the corresponding entry in the cost matrix *C[i][j]*. But forcing a match between two nodes is more difficult. At first glance, it would seem that preventing the match of either of these two nodes with any

node other than the required one, and making the cost of deletion and insertion of these nodes very high, would be enough. It would be enough if the algorithm did not have to handle the additional constraint concerning the distance to the subtree root. Since this constraint exists, it is often necessary to delete entire subtrees at a time. So we have to prevent that one of the nodes involved in the forced match is deleted in one of those subtree deletions. A possible solution would be to prevent the deletion of all the ancestors of the forcibly matched node. This is indeed the best solution if we used THP. But in our case, this solution could produce a very sub-optimal edit script, because it is quite possible that a few ancestors got deleted, while the forcibly matched node isn't deleted. This requires distinguishing between individual delete operations and mass delete operations.

We therefore allow the deletion of ancestors of the forcibly matched node, on the condition that this deletion operation is not part of a subtree deletion operation, i.e., whenever an ancestor is deleted, at least one of its descendents which is itself an ancestor of the forcibly matched node must be part of the successor set. We enforce that constraint in the base case of the recursive BACKTRACK procedure. When computing the best cost for the *(i,j)* entry of the cost matrix, if *i* is an ancestor of a forcibly matched node, BACKTRACK does not record in *BestSolution* any mapping that deletes the branch leading to the forcibly matched node, although it records a mapping that deletes a few intermediate nodes on the path from *i* to the forcibly matched node. This feature is not shown in the pseudo-code to keep it manageable.

3.5 Time and Memory Complexity

An upper bound on the running time of the TreeMDIR algorithm is as follows: let X be the set of nodes of both trees, x be an element of X, p be the maximum allowable size of a connected subgraph of the tree that can be deleted or inserted in the middle of the tree, $f(x,p)$ be the number of nodes that lie within a distance of $(p+1)$ from x, and $F(a) = max\{f(x,p): x \in X \text{ and } p=a\}$.

TreeMDIR has a worst case running time of $O((2*F(a))!\ N^2)$. In our implementation, pruning the search tree by using both tree structure and additional semantic information (e.g., type information) and being able to limit the running time by returning a possibly suboptimal solution, make the average case considerably faster than the worst case. In practice, the observed runtime is $O(K N^2)$ where K is a large constant, but not quite as large as the theoretical worst case bound would let one imagine. In comparison, THP has a running time of $O(d^3 N^2)$.

Regarding memory requirements: although both THP and TreeMDIR can be implemented in $O(N^2)$ space at the expense of increased implementation complexity, we implemented THP in $O(d N^2)$ where d is the max degree of a tree, and TreeMDIR in $O(b N^2)$, where b is the number of bits in an integer.

3.6 Empirical Evaluation

In this section, we present an empirical evaluation of the performance and the accuracy of TreeMDIR. Evaluating the accuracy of the algorithm is necessary because bounds B and R remove the guarantee of optimality. The test data was built as follows: 1) generate a random tree with random labels (taken from a pool of 10 possible names so as to be non-unique); 2) copy the tree; 3) delete a random number of nodes in the copy (both internal and leaf nodes); 4) rename a number of nodes in the copy; 5) and finally, compare the two trees using THP and TreeMDIR. The deletion operations in the middle of the tree correspond to the restricted moves that TreeMDIR detects. In the interest of full disclosure, however, we did not check that at least some of the randomly generated test cases do not violate THP's assumption, namely, that if two nodes match, so do their parents. Additional details can be found in Appendix A.

Table 1: Empirical evaluation of TreeMDIR (R = 100K)

Case	# Nodes	Ops	THP		TreeMDIR	
			Ops	Time	Ops	Time
Rename	640	569	770	2	569	64
	1280	857	1509	7	963	442
Delete	640	492	701	2	492	50
	1280	1113	1397	5	1114	169
Move	640	441	1076	3	1093	215
	1280	652	2407	9	735	471
Degree	640	288	712	2	288	65
	1280	576	1194	10	576	248

The length of an optimal edit script must necessarily be equal to the sum of the number of deletion added to the number of renaming operations, since there is a tree which lacks a certain number of nodes, and it has a number of nodes which doesn't exactly match any of the nodes in the other tree and each of these nodes needs at least one edit operation to be taken into account. Table 1 shows for different tree node sizes, the length

of the optimal edit script, the length of the edit script produced by THP (including the time), and the length of the edit script produced by TreeMDIR (including the time). All times are in seconds.

On average, THP produced edit scripts sub-optimal by about 120%, whereas TreeMDIR produced edit scripts sub-optimal by about 7%. In the worst case, THP produced a suboptimal edit script by about 400% whereas TreeMDIR's worst case performance resulted in an edit script sub-optimal by around 150%. In both cases, accuracy deteriorated significantly when nodes of large degree were allowed or when the trees were very different. TreeMDIR's worst case was on a source tree of 640 nodes separated from its target by an optimal edit script of 440 operations containing both deletions and renames. In that case, the returned edit script was 2.5 times longer than the optimal edit script. This behavior, however, was far from typical and TreeMDIR produced good results with most trees, even when the optimal edit script involved 2/3 of the number of nodes. Finally, with up to 85% of the nodes renamed (no deletions), TreeMDIR produced excellent edit scripts within less than 1% of the optimal script length on trees of 640 nodes, providing us with the evidence that it can recover the mapping from tree structure alone.

The improved match quality comes at a heavy runtime cost. With bound R set to a large value (100 K), TreeMDIR was about 60 times slower than THP on average and up to 200 times slower in the worst case. As predicted, setting bound R to a much smaller value often produced only slightly sub-optimal edit scripts for a noticeably reduced running time: on a tree of 1280 nodes with an optimal edit script of 396 edits, THP produced an edit script of 1775 edit in 7 seconds. TreeMDIR (with R=100K) produced an edit script of size 459 in 6 minutes, whereas TreeMDIR (with R = 5K) produced an edit script of size 479 in 4 minutes. Finally, we would like to point out that we have avoided premature optimization in our current implementation to allow for easier debugging, so we think that the running time can be improved.

4. SYNCHRONIZING C&C VIEWS

We illustrate an application of the algorithm by incorporating it in a set of scalable tools to synchronize C&C views.

4.1 C&C View Differencing and Merging

We represent the structural information in a C&C view as a cross-linked tree structure that mirrors the hierarchical decomposition of the system. The tree also includes information to improve the accuracy of the structural comparison. For instance, the subtree of a node corresponding to a port or role includes all the port's or the role's involvements, i.e., all components (and their ports) or connectors (and their roles) reachable from that port or role through attachments or bindings. Cross-links refer back to the defining occurrence of each element and allow the user to navigate the architectural graph. We also add to each element various properties (such as type information). The type information, if provided, is used to build a matrix of incompatible elements that may not be matched.

A graph representing a C&C view can generally have cycles in it. Representing an architectural graph as a tree causes each shared node in the architectural graph to appear several times in several subtrees, with cross-links referring back to their defining occurrences. These redundant nodes greatly improve the accuracy of the tree-to-tree correction; however, they may be inconsistently matched with respect to their defining occurrences (either in what they refer to, or in the associated edit operations). We post-process the edit script to eliminate inconsistent matches using two passes. During the first pass, we synchronize the strictly hierarchical information (e.g., components, connectors, ports, roles, and representations); during the second pass, we synchronize attachments and bindings. The post-processing step is very simple, since at that point, the mapping between the nodes in the two graphs is known.

4.2 Tool Support

Synchronization follows the following five-step process: 1) Setup the synchronization; 2) View and match types (optional); 3) View and match instances; 4) View and modify the edit script (optional); 5) Confirm and apply the edit script (optional). Because steps 1 and 5 are straightforward, we will only discuss steps 2-4 in more detail below.

In Step 2, matching the type structures between the two views (See Figure 2), currently a manual step, can produce semantic information that speeds up the comparison, but is otherwise optional. It also reduces the amount of data entry for assigning types to the elements to be created by the edit script.

In Step 3, matching instances uses tree-to-tree correction to compare the tree-structured data from the two views to find structural differences and produce an edit script. It consists of: a) retrieve tree-structured data from the first C&C view; b) retrieve tree-structured data from the second C&C view; c) use the tree-to-tree correction algorithm for unordered labeled trees to identify matches and structural differences (classified as inserts, deletes, renames and moves– See Figure 3), and obtain an edit script to make one view more consistent with the other.

The differences found during structural matching are shown in each tree by overlaying icons on the affected elements (see Figure 3). If an element is renamed, the tool automatically selects and highlights the matching element in the other tree; for inserted or deleted elements, the tool automatically selects the insertion point by navigating up the tree until it reaches a matched ancestor.

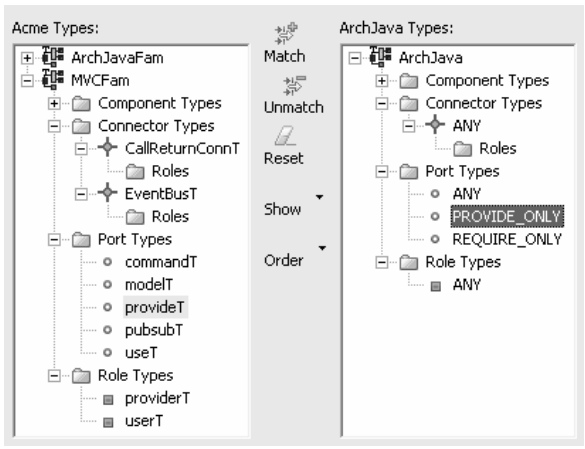

Figure 2: Matching Types Structures: the user manually specifies arbitrary matches in a view that shows the type hierarchies in both views flattened and shown side-by-side: e.g., the user assigns any ArchJava port with only provided methods the *provideT* Acme type defined in the *MVCFam*, a Model-View-Controller style.

The tool provides various features to restrict the size of the trees and therefore, significantly reduce the comparison time:

- **Start at Component:** the architect can have the trees corresponding to the system decomposition start at certain selected components to significantly reduce their sizes.
- **Restrict Tree Depth:** an architect is often interested, at least initially, in only comparing the top-level elements. So the trees can be restricted to not include elements beyond a certain tree depth.
- **Elide Elements:** the architect can selectively exclude entire subtrees from comparison. Elision can be instance-based or type-based, where all elements of a given type are excluded at once (e.g., only match components and ports). Elision is temporary and does not generate any edit actions.

Various features give the user additional manual control:

- **Forced matches:** the architect can manually force a match between two elements that cannot be structurally matched.
- **Manual overrides:** the architect can override any edit action suggested by the comparison, e.g., cancel a delete action.

In Step 4, the edit script is used to produce a common supertree to preview the merged view. This step can be used to supplement the edit script with additional semantic information. For instance, the user can assign types to elements to be created, change the types of existing elements, or override automatically inferred types. Finally, the user can cancel any unwanted edit actions.

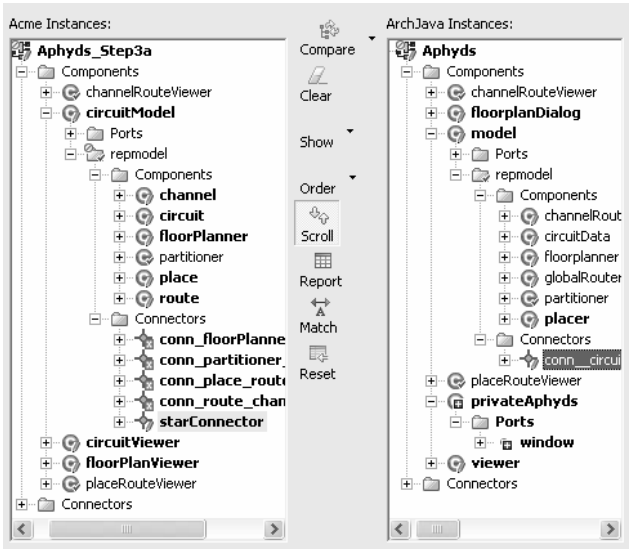

Figure 3: Structural comparison of architectural instances in a C&C view retrieved from Acme and a C&C view retrieved from ArchJava: component *privateAphyds* exists in ArchJava but not in Acme; similarly, connector *starConnector* matches a connector in ArchJava with an automatically generated name (highlighted nodes).
Symbols: Match (✓), Insert (+), Delete (×), Rename (/)

Acme and ArchJava C&C Views. One specialized tool based on this approach can synchronize a C&C view described in an Architectural Description Language (ADL), Acme [GMW00], with a C&C view retrieved from an implementation in ArchJava [ACN02]. We chose Acme, since it is a general purpose ADL with good tool support; we chose ArchJava since it allows recovering a C&C view from an existing implementation. Furthermore, both AcmeStudio [SG04], a domain-neutral architecture modeling environment for Acme, and ArchJava's development environment are Eclipse plugins [Ecl03], thus reducing the tool integration barrier. We have completed the functionality needed to make an Acme model incrementally

consistent with an ArchJava implementation. We still need to change the ArchJava infrastructure to support making incremental changes to an existing ArchJava implementation.

This problem domain clearly requires going beyond insertions and deletions to support renames and moves. There will always be name differences of the same structural information between Acme and ArchJava. As an illustration, even if code generation is used to automatically produce a skeleton implementation from an architectural model, connector names and role names are lost during code generation (since ArchJava does not even name those elements). Identifying a renamed element in one view as being deleted and then re-inserted, while producing structurally equivalent views, results in losing properties about view elements that are crucial for architectural analyses (such style and type information, or other architectural properties).

Matching the type structures in this context is particularly useful. Acme has a predicate-based type system: an element is an instance of any type whose properties and rules it satisfies, and one type is a subtype of another if the predicate of the first type implies the predicate of the second type. Implementation-level type systems such as the ones provided by C2SADL [MOR+96] or ArchJava are more like programming-language type systems. We allow the user to manually specify arbitrary matches between the two type hierarchies in the two systems in a view that shows the type hierarchies in both views flattened and shown side-by-side (See Figure 2).

Matching type structures between Acme and ArchJava can take several forms:

- Match explicit types when possible: e.g., match an ArchJava component type with one or more Acme component types;
- Assign types to instances when no explicit type is available: e.g., assign types to individual ports on an ArchJava component type;
- Assign types to special wildcards: e.g., using the ArchJava connector type *ANY*, one can assign the Acme type *CallReturnT* to all ArchJava implicit connector instances; similarly, one can assign a specific Acme type to a port with only required and no provided methods (e.g., *useT*) or with only provided and no required methods (e.g., *provideT*);
- Finally, infer types when possible: e.g., infer the types of implicit ArchJava roles based on Acme connection patterns optionally defined for an architectural style: if the architect assigns types to components, ports and connectors, the role type (e.g., *providerT*) is inferred based on the source component type (e.g., *ANY*), source port type (e.g., *provideT*), and connector type (e.g., *ANY*)**.**

Two Acme C&C Views. Another specialized tool can more generally synchronize two C&C views represented in Acme: one view could correspond to a documented architecture, and the second could correspond to a C&C view recovered using any architectural recovery technique (e.g., [YGS+04]), another version of the Acme model retrieved from a configuration management system or to another variant in a product line.

Detecting moves across levels of the hierarchy is often helpful, since two architects will often differ in their use of hierarchy, so that components expressed at the top level in one C&C view are nested within another component in some other C&C view. For example, one architect may use hierarchy to hide certain decision decisions from some parts of the system [Par72], but a designer may flatten the hierarchy for efficiency reasons. In an Acme system, this would correspond to replacing an architectural element with its representation (a nested system).

5. CASE STUDY: APHYDS

We illustrate the first tool on an ArchJava implementation of a pedagogical circuit layout application, Aphyds [ACN02]. The goal of this case study is to compare the architecture based on an informal drawing by the developer to the extracted architecture from the ArchJava implementation.

Building the Conceptual Architecture. The starting point was an informal drawing (See Figure 4) of the desired conceptual architecture which loosely followed the Model-View-Controller style, with the *views* consisting of user interface elements and the *model* consisting of a circuit database and a set of computational components. The architect converted the informal diagram into a C&C view (See Figure 5a): he created a single Acme component to represent the *circuitModel* and added all the computational components to a representation of *circuitModel* (See Figure 5b). In the informal diagram, some arrows were meant to represent control flow and others data flow. The architect did not want to distinguish between data and control flow, so he converted all the arrows in the original diagram to connectors in the Acme model.

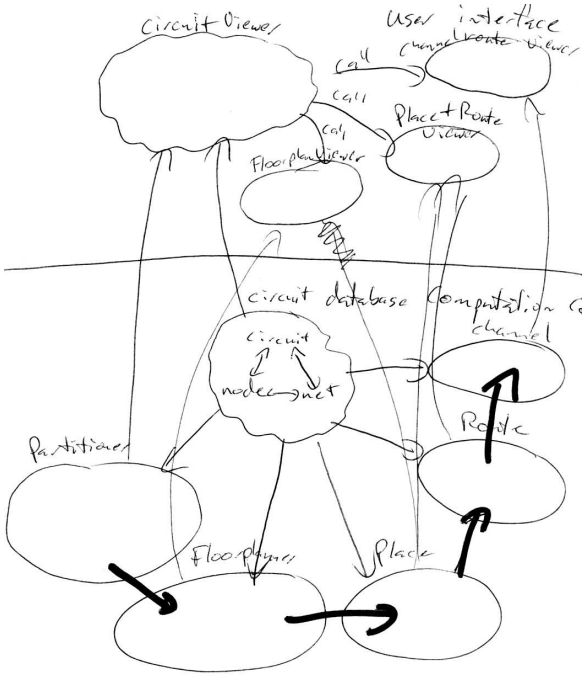

Figure 4: Original Java developer's model.

Matching Types. The architect was interested in the control flow so he assigned the *provideT, useT, provreqT* Acme types to ArchJava ports which only provide, only require, or have both methods, respectively; he assigned the generic *TierNodeT* Acme type to all components and the *CallReturnT* Acme type to all the implicit ArchJava connectors.

Matching Instances. The architect let the synchronization tool compare the two views: he noticed a few renames, e.g., ArchJava uses *model* instead of *circuitModel,* and in that representation, ArchJava uses *globalRouter* instead of *route* (See Figure 3). The Acme architect was the least sure about how he represented the *circuitModel* component in Acme; facing a number of name differences certainly did not raise his confidence level. So, he decided to focus on the *circuitModel* Acme component instance which was matched to the *model* ArchJava component instance. Running the structural comparison showed that the Acme representation for *circuitModel* had more connectors than the ArchJava implementation, i.e., the tool only matched *starConnector* in the middle of Figure 5, modulo renaming (See Figure 3). The architect investigated this further and confirmed that the dataflow arrows in the informal Aphyds boxes-and-lines diagram are not actually in the implementation, so he accepted the edit actions to delete the extra connectors from the Acme model (See Figure 5b).

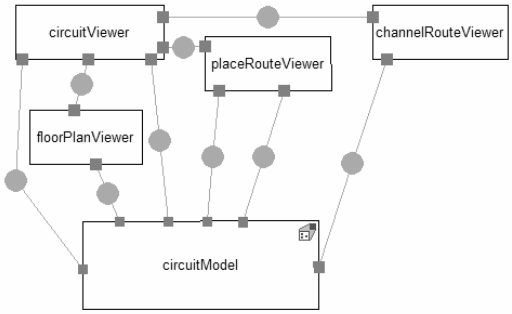

Figure 5a: Original developer's model in Acme.

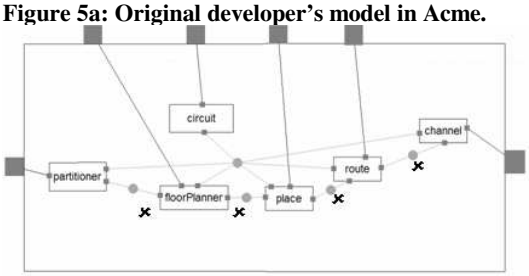

Figure 5b: Acme representation for the *circuitModel* component. Extra connectors are marked with ✖.

Merging Instances. The architect next turned his attention to the additional top level component, shown as *privateAphyds* in Figure 3). *privateAphyds* represents a private *window* port in ArchJava and the corresponding glue. By looking at the control flow, the architect decided to assign that subsystem the publish-subscribe style, so he renamed component *privateAphyds* as *window* and renamed the added connector to *windowBus*, and assigned it the *EventBusT* connector type from the Publish-Subscribe style. The architect also decided to use the same component names as the ArchJava implementation to avoid future confusion, so he let the tool apply the edit script.

Discussion. Figure 6 shows the resulting C&C view after it has been manually laid out in AcmeStudio. Unlike the original architect's model (Figure 4), Figure 6 shows bi-directional communication taking place between components *placeRouteViewer* and *model*; upon further investigation, the architect traced that to a callback. Since Aphyds is a multi-threaded application with long running operations moved onto worker threads, the architect made note of the fact that

developers should not carelessly add callbacks from a worker thread onto the user interface thread. Finally, the architect decided to use the up-to-date C&C view with types and styles as the basis for evolving the system in the future.

Performance Evaluation. On an Intel Pentium4® CPU 3GHz with 1GB of RAM, comparing an Acme tree of around 650 nodes with an ArchJava tree of around 1,150 nodes (as in Figure 3) currently took under 2 minutes, whereas our implementation of THP took around 30 seconds but produced less accurate results: in particular, THP did not treat component *privateAphyds* as an insertion and mismatched all the top-level components. In this case study, the edit script consisted of over 300 renames, over 600 inserts and over 100 deletes.

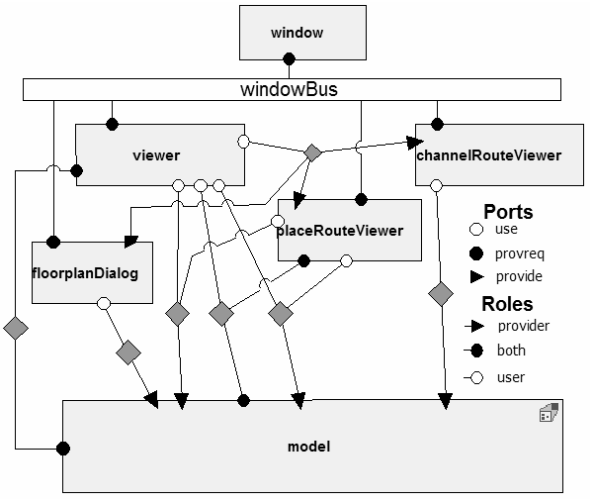

Figure 6: Acme model with styles and types.

6. CASE STUDY: DUKE'S BANK

We illustrate the tool to compare two C&C views using the Duke's Bank Application, a simple Enterprise JavaBeans (EJB) banking application created as a demonstration of EJB functionality [EJB]. Duke's Bank allows bank customers to access their account information and transfer balances from one account to another. It also provides an administration interface for managing customers and accounts. In this case study, the architect wanted to compare the architecture presented in the documentation with the actual architecture discovered by instrumenting the running system as explained in [YGS+04].

The architect defined an Acme family (or style) and types based on the EJB specification. The architect converted a boxes-and-lines diagram documented in a tutorial [J2EE] into an Acme model (See Figure 7).

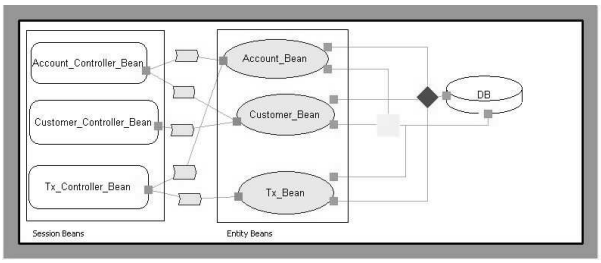

Figure 7: Duke's Bank documented architecture in Acme; the components were added inside the Acme representation of an EJB *container* (shown as a thick border).

As mentioned earlier, the two views must be comparable without any view transformation. Since the model recovered by instrumentation includes each session and entity bean instance created at runtime, the architect post-processed it to eliminate duplicates and consolidate multiple instances into one instance with a property indicating multiplicity (not shown) in Figure 8, to match the documented architecture where each component instance represents a number of run-time components.

The architect ran the synchronization tool between the two Acme C&C views. The tool was able to match all the elements between the two views, despite the large number of renames (automatically generated by the recovery tool). Furthermore, the tool correctly detected all the moves corresponding to replacing the EJB *container* component in one view with its representation in the other view (See Figure 9). The tool also enabled the architect to quickly detect the additional undocumented port on *Account_Controller_Bean*, which is communicating to the *DB* component through a *DbWriter* connector. Figure 7 does not show any connections between the session beans and the database, which implies that all database access is through the entity beans, as recommended by the EJB specification: the architect planned to investigate this apparent violation using source code analysis techniques.

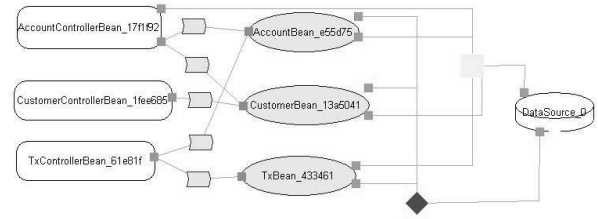

Figure 8a: Duke's Bank recovered architecture in Acme.

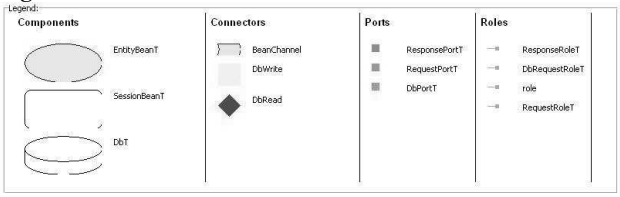

Figure 8b: Legend.

Performance Evaluation. On an Intel Pentium4® CPU 3GHz with 1GB of RAM, TreeMDIR took around 30 seconds to

compare the two Acme trees, one with around 330 nodes, and one with around 390 nodes. In this case, the edit script consisted of over 250 renames and over 50 inserts. As expected, THP did not correctly identify any of the moved view elements in this case.

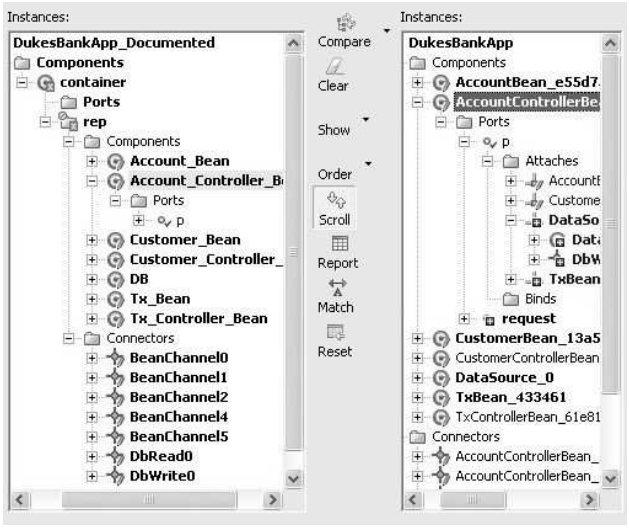

Figure 9: Comparison of the documented and the recovered C&C views for the Duke's Bank application. Symbols: Match (✓), Insert (+), Delete (×), Rename (/)

7. RELATED WORK

In addition to the related work previously mentioned throughout the paper, we point out a few related results.

Program Differencing. Tree-to-tree correction algorithms have been used for finding differences between programs; most approaches consider abstract syntax trees (ASTs) as ordered trees with several polynomial time algorithms available (e.g., [SZ97]).

The Difference Extract (Dex) [RRL+04] includes an algorithm that supports two kinds of move operations: a move that changes parents (a match between nodes whose parents are not matched to each other), and a move that changes order (a match between two nodes with matching parents but different sibling ranks). This work is probably the closest to ours. Although intended to solve the differencing problem for ordered trees, Dex includes a bottom-up algorithm which is vaguely similar to THP as a subroutine that solves an unordered tree problem. Dex purports to support arbitrary moves, but the authors warn that no guarantee can be given that the obtained edit script is optimal because Dex is only a heuristic. This is a reasonable choice for Dex as it typically handles trees that are several orders of magnitude larger than our typical inputs.

There are several important differences between TreeMDIR and Dex, one being that Dex targets inputs where less than 1% of the nodes are affected by edit operations (e.g., at most 200 changed nodes with tree sizes reaching 200,000 nodes or so). The remaining nodes are matched exactly, including their labels. The labels or the semantic information associated with the nodes, which represent the control flow and data of a computer program, have very few duplicates, if any. This enables a linear time subroutine in Dex, called top-down matching, to identify 94% of the matches, and the remaining matches can be deduced by other subroutines. This makes the running time of Dex grow linearly until around 100,000 nodes. In contrast, our implementation of THP, while much slower than Dex, would still work even in the total absence of semantic information (i.e., using tree structure only) without significant impact on running time. Our implementation of TreeMDIR would see its running time increase in practice, but it would still work. And if semantic information is only present on few nodes or is otherwise incomplete, TreeMDIR would still be able to make full use of it. Typical inputs for our algorithms often have more than half of their nodes renamed, and may have a large number of duplicates (for example if the only semantic information we have about the nodes is their type) which would make the Dex top-down subroutine ineffectual, because it would lack the information that lets it quickly match a node to another. Also, TreeMDIR provides the capability of forcing and preventing matches manually, and can find the optimal matching within these user-imposed constraints, assuming these constraints are consistent. This feature does not exist in Dex and we are not sure how difficult it might be to add it. Finally, Dex converts an Abstract Syntax Graph (ASG) into a tree simply by removing non-tree edges corresponding to type information; through empirical evaluation, adding sub-tree information greatly improves the accuracy of the tree-to-tree correction algorithm.

[CG97] proposes a heuristic solution with a worst-case $O(N^3)$ time that supports arbitrary move, copy and glue operations. However, the approach was only tested on instances of a few hundred nodes where 80% or more of the nodes were matching exactly (same semantic information) without any indication of how many of these labels were unique. Also the largest instance over which the accuracy of the heuristic was tested did not contain more than six edit operations (including renames): even on that, the heuristic returned a suboptimal answer in some cases (about 15% larger than the optimal edit script).

JDIFF [AOH04] bears some vague similarity to TreeMDIR, as hammock graphs can be turned into trees without loss of information or structure. We think that it would be trivial to add the ability to prevent matches to JDIFF, but adding the ability to force matches would be substantially more complicated. JDIFF is faster than TreeMDIR since it relies on matching labels exactly, but it loses the ability to detect renames, one of our requirements.

Tree Alignment vs Tree Edit. Tree differences can be represented using tree alignment instead of tree edit distance. Each alignment of trees actually corresponds to a restricted tree edit in which all the insertions precede all the deletions. There are algorithms based on tree alignment that can detect unbounded deletes (e.g., [JWZ95]). Another advantage of tree alignment is that it can easily generalize to more than two trees, something not easily done with tree edit distance. But the memory requirements of such algorithms are prohibitive for the tree sizes and branching factors that are typical of our inputs: the memory requirements would typically be several orders of magnitude higher than those of TreeMDIR– $O(2^{2d} N^2)$ where d is the maximum degree of the tree. Due to the prohibitive space requirements, there's no need to prefer tree alignment to an algorithm based on tree edit distance.

8. CONCLUSIONS

In this paper, we presented a novel algorithm for finding differences and merging tree-structured data. Given two tree-structured representations, our algorithm identifies, in addition to inserts, deletes, and renames, restricted moves across levels of the hierarchy. The algorithm also supports manually forcing and preventing matches between view elements.

We also presented tools that use the tree-to-tree correction algorithm to compare and merge architectural component-and-connector (C&C) views. Finally, we provided an empirical evaluation of the algorithms and tools with case studies on real programs. The case studies show the practicality of the algorithm and the tool, as they enabled us to find interesting architectural divergences in both cases.

9. ACKNOWLEDGMENTS

This work was supported in part by NASA cooperative agreements NCC-2-1298 and NNA05CS30A, NSF grant CCR-0204047, a 2004 IBM Eclipse Innovation Grant, the Army Research Office grant number DAAD19-02-1-0389 entitled “Perpetually Available and Secure Information Systems”, and was performed as a joint research project in Strategic Partnership between Carnegie Mellon University and Jet Propulsion Laboratory.

10. REFERENCES

[AAG05] Abi-Antoun, M., Aldrich, J., Garlan, D., Schmerl, B., Nahas, N., and Tseng, T. Improving System Dependability by Enforcing Architectural Intent. In WADS, 2005.

[AC94] Ammann, M. M., and Cameron, R.D. Inter-Module Renaming and Reorganizing: Examples of Program Manipulation-in-the-Large. In Proc. ICSM, 1994.

[ACN02] Aldrich, J., Chambers, C. and Notkin, D. ArchJava: Connecting Software Architecture to Implementation. In Proc. ICSE, 2002.

[AOH04] Apiwattanapong, T., Orso, A. and Harrold, M.J. A Differencing Algorithm for Object-oriented Programs. In Proc. Automated Software Engineering, 2004.

[AP03] Alanen, M. and Porres, I. Difference and Union of Models. In Proc. «UML» 2003, 2003.

[CBB+03] Clements, P., Bachman, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J. Documenting Software Architecture: View and Beyond, Addison-Wesley, 2003.

[CCG+03] Chen, P., Critchlow, M., Garg, A., van der Westhuizen, C. and van der Hoek, A. Differencing and Merging within an Evolving Product Line Architecture. In Proc. PFE-5, 2003.

[CFS+04] Conte, D., Foggia, P., Sansone, C., Vento, M. Thirty years of graph matching in pattern recognition. In Int'l J. Pattern Recognition and Artificial Intelligence, 18(3), 2004.

[CG97] Chawathe, S. and Garcia-Molina, H. Meaningful change detection in structured data. In Proc. ACM SIGMOD, 1997.

[DBD+04] Dickinson, P.J., Bunke, H., Dadej, A., and Kraetzl, M. Matching graphs with unique node labels. In Pattern Analysis & Applications. 7(3), pp. 243- 254, 2004.

[DHT02] Dashofy, E. M., van der Hoek, A., and Taylor, R. N. An infrastructure for the rapid development of XML-based architecture description languages. In Proc. ICSE, 2002.

[Ecl03] Object Technology International, Inc. Eclipse Platform Technical Overview, 2003. http://www.eclipse.org/whitepapers/eclipse-overview.pdf

[EJB] Sun Microsystems. Enterprise JavaBeans. http://java.sun.com/products/ejb/docs.html

[GMW00] Garlan, D., Monroe, R., and Wile, D. Acme: Architectural Description of Component-Based Systems. In Foundations of Component-Based Systems, Cambridge University Press, 2000.

[J2EE] Sun Microsystems. J2EE Tutorials. Duke's Bank. http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Ebank2.html

[Jim05] Jimenez, A. M. Change Propagation in the MDA: A Model Merging Approach. M.S. Thesis. University of Queesland, 2005.

[JWZ95] Jiang, T., Wang, L., and Zhang, K., Alignment of trees— an alternative to tree edit. In Theoretical Computer Science, 143:137--148, 1995.

[KPS+99] Krikhaar, R., Postma, A., Sellink, A., Stroucken, M., Verhoef, C. A Two-Phase Process for Software Architecture Improvement. In Proc. ICSM, 1999.

[MDR05] Muccini, H., Dias, M. and Richardson, D. Towards Software Architecture-based Regression Testing. In WADS, 2005.

[OWK03] Ohst, D., Welle, M., and Kelter, U. Differences between Versions of UML Diagrams. In Proc. FSE, 2003.

[Par72] Parnas, D. On the Criteria for Decomposing Systems into Modules. In Communications ACM 15 (12), 1972.

[RHM+04] Roshandel, R., van der Hoek, A., Mikic-Rakic, M. and Medvidovic, N. Mae A System Model and Environment for Managing Architectural Evolution. In TOSEM, 2004.

[RRL+04] Raghavan, S., Rohana, R., Leon, D., Podgurski, A. and Augustine, V. Dex: a semantic-graph differencing tool for studying changes in large code bases. In Proc. ICSM, 2004.

[SG04] Schmerl, B. and Garlan, D. AcmeStudio: Supporting Style-Centered Architecture Development. In ICSE, 2004.

[SWZ+94] Shasha, D., Wang, J., Zhang, K., Shih, F. Exact and approximate algorithms for unordered tree matching. In IEEE Trans. Sys. Man. Cyber. 24(4): 668-678, 1994.

[SZ97] Shasha, D., Zhang, K. Approximate Tree Pattern Matching, in Pattern Matching Algorithms, Apostolico, A. and Galil, Z., Eds., Oxford University Press, 1997.

[THP05] Torsello, A., Hidovic-Rowe, D. and Pelillo, M. Polynomial-Time Metrics for Attributed Trees. In IEEE Trans. Pattern Analysis and Machine Intelligence, 2005.

[WDC03] Wang, Y., Dewitt, D.J. and Cai, J.-Y. X-Diff: An Effective Change Detection Algorithm for XML Documents. In Proc. 19th Int'l Conf. Data Eng., 2003.

[WF74] Wagner, R.A. and Fischer, M.J. The string to string correction problem. Journal of the ACM, 21:168--173, 1974.

[YGS+04] Yan, H., Garlan, D., Schmerl, B., Aldrich, J. and Kazman, R. DiscoTect: A System for Discovering Architectures from Running Systems. In ICSE, 2004.

[ZJ94] Zhang, K., and Jiang, T. Some MAX SNP-hard results concerning unordered labeled trees. In Information Processing Letters, 49, pp. 249–254, 1994.

Appendix A

The test cases were built as follows:

1) generate a random tree with random labels (taken from a pool of 10 possible names so as to be non-unique);

2) copy the tree;

3) delete a random number of nodes in the copy (both internal and leaf nodes);

4) rename a number of nodes in the copy;

5) and finally, compare the two trees using THP and TreeMDIR.

TreeMDIR was run once with bound R = 100K, and another time with bound R = 5K. Bound R was left unchanged from its default value in all runs.

This appendix contains the test results.

Table 1: List of abbreviations

Abbreviation	Description
N	Number of Nodes
R	Number of Renames
D	Number of Deletes
ID%	Percentage of Internal Deletes
DE	Tree Degree
O	Number of Optimal Edit Operations
AE	Actual Number of Edit Operations
S%	Percentage by which the generated edit script is suboptimal
T	Running time (in seconds)
F	Slow down factor (compared to THP)

Table 2: List of tables.

Table 2	Testing Renames
Table 3	Testing Deletes
Table 4	Testing Internal Deletes
Table 5	Testing Node Degree
Table 6	Summary

Table 2: Testing Renames.

N	R	O	THP			TreeMDIR (R = 100K)				TreeMDIR (R = 5K)			
			AE	S%	T	AE	S%	T	F	AE	S%	T	F
320	0	96	350	265%	0.594	114	19%	17.781	29	114	19%	15.078	24
320	3	99	460	365%	0.438	143	44%	34.625	78	143	44%	16.265	36
320	6	102	367	260%	0.484	112	10%	19.11	38	112	10%	16.609	33
320	9	105	526	401%	0.484	109	4%	27.109	55	109	4%	16.141	32
320	12	108	240	122%	0.484	109	1%	14.625	29	109	1%	14.563	29
320	19	115	326	183%	0.469	116	1%	15.593	32	116	1%	15.141	31
320	25	121	402	232%	0.484	125	3%	16.735	34	125	3%	15.843	32
320	35	131	473	261%	0.5	138	5%	31.672	62	138	5%	16.297	32
320	44	140	336	140%	0.469	141	1%	14.296	29	141	1%	14.266	29
320	57	153	368	141%	0.484	158	3%	19.234	39	158	3%	17.329	35
320	73	169	437	159%	0.453	171	1%	26.375	57	171	1%	17.265	37
320	92	188	360	91%	0.485	188	0%	16.437	33	188	0%	15.688	31
320	118	214	433	102%	0.469	214	0%	16.156	33	214	0%	15.485	32
320	150	246	480	95%	0.468	250	2%	25.125	53	250	2%	16.563	34
320	188	284	516	82%	0.453	295	4%	32.375	70	295	4%	17.391	37
640	0	192	895	366%	2.032	309	61%	67.312	32	309	61%	56.766	27
640	6	198	586	196%	1.75	221	12%	71.172	40	221	12%	64.75	36
640	12	204	707	247%	1.875	216	6%	85.219	44	216	6%	64.547	33
640	19	211	789	274%	1.906	239	13%	80.563	41	239	13%	64	33
640	25	217	827	281%	1.907	239	10%	140.421	73	239	10%	71.422	36
640	38	230	1058	360%	1.906	266	16%	136.812	71	266	16%	64.063	33
640	51	243	883	263%	1.907	248	2%	88.687	46	248	2%	65.922	34
640	70	262	817	212%	1.968	310	18%	126	63	310	18%	64.141	32
640	89	281	1041	270%	1.828	323	15%	158.297	86	323	15%	70.797	38
640	115	307	634	107%	1.829	307	0%	74.578	40	307	0%	65.484	35
640	147	339	983	190%	1.782	400	18%	86.64	48	400	18%	61.906	34
640	185	377	802	113%	1.984	405	7%	109.125	54	405	7%	74.422	37
640	236	428	924	116%	1.906	445	4%	109.579	56	445	4%	65.735	33
640	300	492	994	102%	1.891	529	8%	98.609	51	529	8%	65.625	34
640	377	569	1030	81%	1.875	591	4%	95.765	50	591	4%	69.141	36
1280	0	384	1879	389%	8.14	552	44%	446.141	54	552	44%	260.641	31
1280	12	396	1879	374%	7.141	408	3%	341.891	47	408	3%	264.39	36
1280	25	409	2111	416%	7.125	413	1%	334.703	46	413	1%	259.375	35
1280	38	422	1689	300%	7.672	775	84%	431.578	55	775	84%	252.907	32
1280	51	435	2095	382%	7.11	471	8%	419.687	58	471	8%	281.922	39
1280	76	460	2097	356%	7.063	483	5%	454.547	63	483	5%	256.687	35
1280	102	486	2041	320%	7.469	628	29%	390	51	628	29%	261.328	34
1280	140	524	1891	261%	7.125	739	41%	360.391	50	739	41%	255.875	35
1280	179	563	2138	280%	7.109	747	33%	596.25	83	747	33%	266.203	36
1280	230	614	1792	192%	7.422	1306	113%	357.156	47	1310	113%	257.735	34
1280	294	678	1930	185%	7.297	849	25%	344.875	46	849	25%	246.313	33
1280	371	755	1819	141%	7.266	867	15%	427.359	58	867	15%	266.625	36
1280	473	857	2114	147%	7.172	875	2%	480.328	66	875	2%	279.672	38
1280	601	985	1831	86%	8.359	1058	7%	413.969	49	1058	7%	259.891	30
1280	755	1139	2133	87%	7.235	1166	2%	468.797	64	1166	2%	292.422	39

Table 3: Testing Deletes.

N	D	O	THP			TreeMDIR (R = 100K)				TreeMDIR (R = 5K)			
			AE	S%	T	AE	S%	T	F	AE	S%	T	F
320	0	128	128	0%	0.625	128	0%	19.578	30	128	0%	19.609	30
320	3	131	131	0%	0.985	131	0%	19.859	19	131	0%	19.891	19
320	6	134	134	0%	0.593	134	0%	19.157	31	134	0%	19.171	31
320	9	137	137	0%	0.594	137	0%	19.031	31	137	0%	19.047	31
320	12	140	140	0%	0.594	140	0%	20.453	33	140	0%	20.485	33
320	19	147	147	0%	0.562	147	0%	19.609	34	147	0%	19.625	34
320	25	153	153	0%	0.579	153	0%	18.078	30	153	0%	18.125	30
320	35	163	166	2%	0.547	163	0%	16.765	30	163	0%	16.781	30
320	44	172	184	7%	0.547	172	0%	17.719	31	172	0%	17.75	31
320	57	185	190	3%	0.484	185	0%	15.922	32	185	0%	15.938	32
320	73	201	215	7%	0.453	201	0%	15.469	33	201	0%	15.5	33
320	92	220	276	25%	0.437	220	0%	13.922	31	220	0%	13.938	31
320	118	246	266	8%	0.406	246	0%	11.391	27	246	0%	11.453	27
320	150	278	372	34%	0.344	278	0%	10.016	28	278	0%	10.031	28
640	0	256	256	0%	2.328	256	0%	80	33	256	0%	79.578	33
640	6	262	262	0%	2.422	262	0%	79.672	32	262	0%	79.672	32
640	12	268	268	0%	2.812	268	0%	76.781	26	268	0%	76.813	26
640	19	275	275	0%	2.406	275	0%	71.328	29	275	0%	71.453	29
640	25	281	281	0%	2.297	282	0%	79.563	34	282	0%	79.703	34
640	38	294	294	0%	2.282	294	0%	76.859	33	294	0%	76.453	33
640	51	307	309	1%	2.219	307	0%	74.343	33	307	0%	74.25	32
640	70	326	331	2%	2.531	326	0%	72.953	28	326	0%	72.953	28
640	89	345	352	2%	2.125	345	0%	67.907	31	345	0%	67.437	31
640	115	371	384	4%	2.203	371	0%	67.594	30	371	0%	67.578	30
640	147	403	440	9%	1.844	403	0%	64.047	34	403	0%	64.093	34
640	185	441	495	12%	2.031	441	0%	56.375	27	441	0%	56.344	27
640	236	492	745	51%	1.562	493	0%	50.469	31	493	0%	50.609	31
640	300	556	806	45%	1.313	556	0%	42.703	32	556	0%	42.484	31
1280	0	512	512	0%	10.81	513	0%	330.953	30	513	0%	330.719	30
1280	12	524	524	0%	10.41	525	0%	322.734	30	525	0%	322.329	30
1280	25	537	537	0%	10.59	537	0%	308.75	28	537	0%	308.235	28
1280	38	550	553	1%	10.3	550	0%	322.015	30	550	0%	321.469	30
1280	51	563	567	1%	10.3	563	0%	313.907	29	563	0%	308.578	29
1280	76	588	589	0%	10.19	588	0%	301	29	588	0%	299.766	28
1280	102	614	619	1%	10	614	0%	289.672	28	614	0%	289.422	28
1280	140	652	661	1%	8.625	654	0%	285.172	32	654	0%	285.14	32
1280	179	691	699	1%	9.25	691	0%	278.86	29	691	0%	278.657	29
1280	230	742	773	4%	7.765	743	0%	274.485	34	743	0%	274.609	34
1280	294	806	873	8%	8.5	806	0%	243.625	28	806	0%	243.64	28
1280	371	883	1030	17%	6.781	883	0%	230.453	33	883	0%	230.016	33
1280	473	985	1379	40%	6.094	986	0%	200.203	32	986	0%	200.015	32
1280	601	1113	1435	29%	5.734	1115	0%	168.094	28	1115	0%	167.438	28

Table 4: Testing Internal Deletes.

N	ID%	O	THP			TreeMDIR (R = 100K)				TreeMDIR (R = 5K)			
			AE	S%	T	AE	S%	T	F	AE	S%	T	F
320	0	128	128	0%	0.625	128	0%	21.14	33	128	0%	21.125	33
320	1	131	135	3%	0.61	131	0%	21.265	34	131	0%	21.281	34
320	2	134	244	82%	0.625	134	0%	21.032	33	134	0%	20.734	32
320	3	137	213	55%	0.625	137	0%	19.234	30	137	0%	19.282	30
320	4	140	193	38%	0.609	140	0%	18.875	30	140	0%	18.859	30
320	6	147	278	89%	0.579	149	1%	26.031	44	149	1%	23.015	39
320	8	153	407	166%	0.563	154	1%	24.469	42	154	1%	22.062	38
320	11	163	409	151%	0.562	192	18%	23.922	42	192	18%	18.922	33
320	14	172	552	221%	0.531	176	2%	42.906	80	176	2%	26.016	48
320	18	185	524	183%	0.531	200	8%	50.328	94	200	8%	23.328	43
320	23	201	510	154%	0.453	308	53%	57.532	126	308	53%	19.265	42
320	29	220	523	138%	0.641	350	59%	51.25	79	350	59%	19.578	30
640	0	256	256	0%	2.437	256	0%	78.828	31	256	0%	79.032	31
640	1	262	294	12%	2.843	262	0%	75.969	26	262	0%	76.031	26
640	2	268	433	62%	2.516	268	0%	80.609	31	268	0%	79.875	31
640	3	275	409	49%	2.485	275	0%	79.5	31	275	0%	79.187	31
640	4	281	476	69%	2.64	282	0%	78.782	29	282	0%	78.687	29
640	6	294	484	65%	2.437	296	1%	90.672	36	296	1%	83.656	33
640	8	307	915	198%	2.344	307	0%	112.234	47	307	0%	91.813	38
640	11	326	1106	239%	2.391	392	20%	109.062	45	392	20%	83.047	34
640	14	345	902	161%	2.516	394	14%	174.125	68	394	14%	89.781	35
640	18	371	848	129%	2.015	415	12%	199.297	98	415	12%	89.203	43
640	23	403	1092	171%	2.031	932	131%	202.516	99	933	132%	82.031	39
640	29	441	1033	134%	1.844	612	39%	240.921	130	614	39%	76.875	41
1280	0	512	512	0%	11.86	512	0%	316.453	26	512	0%	316.641	26
1280	1	524	666	27%	10.03	524	0%	328.969	32	524	0%	323.813	31
1280	2	537	675	26%	10.19	537	0%	329.547	31	537	0%	326.39	31
1280	3	550	892	62%	9.906	550	0%	340.797	33	550	0%	330.359	32
1280	4	563	1782	217%	11.27	563	0%	400.562	35	563	0%	342.172	29
1280	6	588	1106	88%	9.797	589	0%	345.984	34	589	0%	337.547	33
1280	8	614	1241	102%	9.578	618	1%	442.375	45	618	1%	340.641	35
1280	11	652	2153	230%	9.203	685	5%	515.969	55	685	5%	361.516	38
1280	14	691	1911	177%	9.437	765	11%	706.313	74	765	11%	374.25	39
1280	18	742	2246	203%	8.437	2328	214%	627.922	73	2328	214%	323.406	37
1280	23	806	2193	172%	8.188	1195	48%	1028.66	125	1195	48%	365.125	44
1280	29	883	2112	139%	7.687	1469	66%	1026.74	133	1477	67%	360.187	46

Table 5: Testing Degree.

N	D	O	THP			TreeMDIR (R = 100K)				TreeMDIR (R = 5K)			
			AE	S%	T	AE	S%	T	F	AE	S%	T	F
320	2	144	392	172%	0.578	144	0%	17.141	29	144	0%	17.203	29
320	3	144	222	54%	0.594	144	0%	21.687	36	144	0%	21.453	35
320	4	144	261	81%	0.656	150	4%	44.047	66	150	4%	23.797	35
320	5	144	258	79%	0.657	144	0%	68.406	103	144	0%	26.156	39
320	6	144	252	75%	0.64	145	1%	109.86	171	146	1%	27.531	42
320	7	144	193	34%	0.625	144	0%	98.547	157	144	0%	25.406	40
320	8	144	278	93%	0.609	151	5%	82	134	151	5%	22.328	36
320	9	144	480	233%	0.578	144	0%	100.172	172	144	0%	25.766	44
320	10	144	235	63%	0.609	144	0%	60.938	99	144	0%	23.515	38
320	11	144	222	54%	0.891	144	0%	113.641	127	144	0%	24.453	26
320	12	144	216	50%	0.562	146	1%	80.875	143	146	1%	23.469	41
320	13	144	403	180%	0.593	144	0%	70.594	118	144	0%	23.125	38
320	14	144	356	147%	0.61	278	93%	67.656	110	278	93%	20.375	32
640	2	288	693	141%	2.063	289	0%	62.969	30	289	0%	63.156	30
640	3	288	488	69%	2.656	288	0%	75.094	27	288	0%	74.578	27
640	4	288	804	179%	2.266	393	36%	204.531	89	393	36%	85.188	37
640	5	288	699	143%	2.625	318	10%	345.766	131	318	10%	112.109	42
640	6	288	632	119%	2.609	289	0%	272.25	103	289	0%	91.328	34
640	7	288	988	243%	2.718	329	14%	306.282	112	329	14%	88.922	32
640	8	288	1007	250%	2.547	296	3%	370.031	144	296	3%	92.203	35
640	9	288	680	136%	2.609	380	32%	337.532	128	380	32%	97.437	36
640	10	288	1022	255%	2.594	320	11%	293.656	112	320	11%	87.219	33
640	11	288	398	38%	2.766	288	0%	315.109	113	288	0%	94.922	33
640	12	288	516	79%	2.593	300	4%	442.844	170	300	4%	106.954	40
640	13	288	546	90%	2.578	294	2%	255.265	98	294	2%	91.11	34
640	14	288	750	160%	2.578	394	37%	320.391	123	394	37%	87.843	33
1280	2	576	1571	173%	8.985	578	0%	246.937	26	578	0%	246.781	26
1280	3	576	1434	149%	9.062	576	0%	309.172	33	576	0%	294.313	31
1280	4	576	1550	169%	9.484	578	0%	863.906	90	578	0%	386.375	40
1280	5	576	1515	163%	9.641	600	4%	1293.05	133	600	4%	390.156	39
1280	6	576	1296	125%	10.61	579	1%	1527.69	143	579	1%	414.984	38
1280	7	576	954	66%	10.06	579	1%	1517.95	150	581	1%	416.703	40
1280	8	576	2327	304%	9.703	576	0%	1420.03	145	576	0%	397.813	40
1280	9	576	2233	288%	9.235	2154	274%	1208.19	130	2154	274%	362.188	38
1280	10	576	1194	107%	9.515	687	19%	1316.38	137	687	19%	360.516	37
1280	11	576	1021	77%	9.375	586	2%	1173.48	124	586	2%	361.86	38
1280	12	576	1160	101%	9.593	722	25%	1324.64	137	683	19%	352.625	36
1280	13	576	1186	106%	9.516	705	22%	1348.83	141	705	22%	374.453	38
1280	14	576	1988	245%	9.375	694	20%	1688.02	179	697	21%	404.672	42

Table 6: Summary.

	THP	TreeMDIR (R = 100K)		TreeMDIR (R = 5K)	
	Sub-Optimal %	**Sub-Optimal %**	**Slowdown Factor**	**Sub-Optimal %**	**Slowdown Factor**
Average	121%	13%	61.86	13%	34
Min	0%	0%	19.16	0%	19
Max	416%	274%	179.06	274%	48