Differencing and Merging of Architectural Views

Marwan Abi-Antoun Jonathan Aldrich Nagi Nahas Bradley Schmerl David Garlan
Institute for Software Research Intl, Carnegie Mellon University, Pittsburgh, PA 15213 USA
{mabianto+, aldrich+}@cs.cmu.edu, nnahas@acm.org, {schmerl+, garlan+}@cs.cmu.edu

Abstract

Existing approaches to differencing and merging architectural views are based on restrictive assumptions such as requiring view elements to have unique identifiers or exactly matching types.

We propose an approach based on structural information by generalizing 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 incorporate the algorithm into tools to compare and merge component-and-connector (C&C) architectural views. Finally, we provide an empirical evaluation of the algorithm on case studies to find and reconcile interesting divergences between architectural views.

1. Introduction

The software architecture of a system defines its high-level organization as a collection of runtime components, connectors, their properties and constraints on their interaction. Such an architecture is commonly referred to as a component-and-connector (C&C) view. 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. For instance, during analysis, a software architect may want to reconcile two C&C views representing two variants in a product line architecture [5]. 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 architectural recovery techniques) to discover implementation-level violations of the architectural intent.

Several techniques and tools have been proposed for differencing and merging C&C views. Most techniques do not detect differences based on structural information: they assume that elements have unique identifiers [2, 13, 17] or match two elements if both their labels and their types match [5], which is not possible when dealing with views at different levels of abstraction. Many techniques detect only a small number of differences: for instance, ArchDiff [5] only detects insertions and deletions, possibly leading to the loss of information when elements are moved or renamed. Tracking changes using element-level versioning [12, 19] can infer high-level operations such as merges, splits or clones in addition to the low-level operations such as inserts and deletes, but requires an upfront investment in tool building and cannot be used on existing models.

In this paper, we propose an approach that overcomes some of these limitations. Our 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 match quality.
- A generalization of a recently published optimal tree-to-tree correction algorithm for unordered labeled trees [22] 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. One tool can synchronize a high-level C&C view with a C&C view retrieved from an implementation. Another tool can more generally synchronize two C&C views.
- An empirical evaluation of the algorithms and the associated tools on realistic programs.

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 de-

scribes how the algorithm is used to synchronize C&C views. Section 5 presents applications of the approach in case studies on real systems. Finally, we discuss related work and conclude.

2. Architectural View Differencing

Software architects rely on multiple architectural views, where a view is a representation of a set of system elements and the relationships between them. Since a view can generally be described as a graph, view differencing and merging is a problem in graph matching.

Graph matching measures the similarity between two graphs using the notion of graph edit distance [7], i.e., produces 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. Typically 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.

Graph matching is NP-complete in the general case [7]: graphs with unique node labels can be processed efficiently [8] which explains why many approaches make this assumption. The most ambitious optimal graph matching algorithms (i.e., if a global minimum of the matching cost exists, it will be found) can handle at most a few dozen nodes [7, 16]. Non-optimal heuristic-based algorithms are more scalable but often place other restrictive assumptions: for instance, the Similarity Flooding Algorithm (SFA) "works for directed labeled graphs only. It degrades when labeling is uniform or undirected, or when nodes are less distinguishable. [It] does not perform well […] on undirected graphs having no edge labels" [14].

Several efficient algorithms have been proposed for trees, a strict hierarchical structure, so we focus on hierarchical architectural views. While not all architectural views are hierarchical, hierarchy is often used for scalability to attain both high-level understanding and detail. In a C&C view, the tree-like hierarchy corresponds to the system decomposition, but cross-links between the system elements form a general graph. Other architectural views such as module views have similar characteristics. Many approaches [4, 18, 25] are hierarchical, so our choice is hardly new. However, we relax the constraints of existing approaches as follows:

- **No Unique Identifiers.** For maximum generality, we do not require elements to have unique identifiers [5, 13]. Making this assumption enables the use of exact and scalable algorithms that handle thousands of nodes [8]. Unfortunately, architectural view elements often do not have unique identifiers.
- **No Ordering.** In the general case, an architectural view has no inherent ordering among its elements. This suggests that an unordered tree-to-tree correction algorithm might perform better than one for ordered trees. Many efficient algorithms are available for ordered labeled trees (e.g., [20]). In comparison, tree-to-tree correction for unordered trees is MAX SNP-hard [27]. Some algorithms for unordered trees achieve polynomial-time complexity, either through heuristic methods (e.g., [6, 18, 24]) or under additional assumptions (e.g., THP [22]).
- **Renames.** Names are often modified during software development and maintenance. Architectural view elements may not have persistent names or may be assigned automatically generated names. This suggests that an algorithm should be able to match renamed elements. A number of existing algorithms claim to detect renames, but assume that a large majority of nodes have exactly matching information [6, 18]. Identifying a renamed element 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. For our purposes, a matched node is a node with either an exactly matching or a renamed label.
- **Hierarchical Moves.** Architects often use hierarchy to control complexity. However, two architects often 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. We would like to detect sequences of node deletions in the middle of the tree resulting in nodes moving up a number of levels in the hierarchy, and sequences of node insertions in the middle of the tree resulting in nodes moving down in the hierarchy (by becoming children of the inserted nodes), as shown in case *T_1'* in Figure 1.
- **Manual Overrides.** It is possible to encounter cases involving structural aberrations that may lead a fully automated algorithm to incorrectly match top-level elements between two trees and produce an unusable output. Because of the dependencies in the mapping, these incorrect matches cannot be easily manually corrected after the fact. Instead, we required a feature not typically found in tree-to-tree correction algorithms: allow the user to force or prevent matches between certain view elements, and have the algorithm take these constraints into

account to produce an improved overall match. The user can specify any set of constraints as long as they preserve the ancestry relation between the forcibly matched nodes, i.e., if *a* is an ancestor of *b*, *a* is forcibly matched to *c*, and *b* is forcibly matched to *d*, then *c* must be an ancestor of *d*.

- **Optional Type Information.** Architectural views may contain untyped elements or have different or incompatible type systems. This is the case when comparing views at different levels of abstraction such as a conceptual-level (as-designed) view with an implementation-oriented (as-built) view. Therefore, an algorithm should not rely on matching type information. It should be able to recover a correct mapping from structure alone if necessary or from structure and type information if type information is available. However, an algorithm could take advantage of type information (when available) to prune the search tree by not attempting to match elements of incompatible types.
- **Disconnected/Stateless Operation.** For maximum generality, we assume a disconnected and stateless operation, i.e., no monitoring or recording of the structural changes is taking place while the user is modifying a given view as in [12, 19].
- **Comparable Views.** The two views being compared and merged have to be somewhat structurally similar. When comparing two completely different views, the algorithm could trivially delete all elements of one view and then insert all the elements in the other view.

3. Tree-to-Tree Correction

In this section, we describe a novel algorithm for unordered labeled trees, MDIR (Move-Delete-Insert-Rename), which generalizes a recent optimal tree-to-tree correction algorithm, denoted as THP [22]

3.1. Problem Definition

We first give an unambiguous definition of the problem, adapted from [20]. 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).

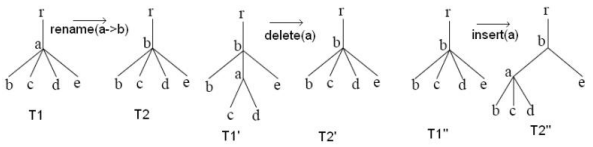

Figure 1: Edit operations (adapted from [20]).

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 and preserves any properties associated with it. In comparison, 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).

Suppose we obtain a mapping $\mathcal{M}$ between trees T_1 and T_2. From this mapping we can deduce an *edit script* (a sequence of edit operations) 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. To define the cost of an edit script, 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, string-to-string correction [23]) changes *"banana"* into *"ananas"* with a cost of two). 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, to find an optimal edit sequence, it is sufficient to find an optimal mapping.

3.2. Explanation of the Algorithm

The algorithm's pseudo-code is given in Figure 2. 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)*.

```
BestSolution: list of node pairs that represents the best discovered matching between
              successor sets of two nodes, where a successor set of node i is denoted by S(i)
CurrentSolution: dynamic list of node pairs that represents a matching being built between
                 successor sets of two nodes
CostMatrix: CostMatrix[i][j] is the cost of the optimal mapping from S(i) to S(j)
BestCost: Cost of the BestSolution matching
BestGlobalMatch[]: array of node pairs corresponding to least cost mapping from T1 to T2
BestSuccessor[][]: 2D array of lists of node pairs
   (m,n)∈ BestSuccessor[i][j] means (m,n) is a match between one element of S(i) and
   one element of S(j) in an optimal mapping from S(i) to S(j)
MatchMerit(i,j): measure of the similarity (i.e., quality of matching, not cost) between nodes i
        and j, deduced from CostMatrix[i][j] as (1 - CostMatrix[i][j]/(sum of subtree weights))

Procedure: MDIR /* Main procedure */
Input:
  Tree T1: first tree to compare
  Tree T2: second tree to compare (turn T1 into T2)
  L(i,j): cost of string-to-string correction to change Label(i) in T1 to Label(j) in T2
Begin
  Postorder T1 and T2 nodes
  for(i = 1 to T1.size)
    for(j = 1 to T2.size)
      BestSuccessor[i][j] = Search(i, j)
      CostMatrix[i][j] = cost(BestSuccessor[i][j]) + L(i,j)
  GetBestMatching(T1.size, T2.size)
End

Procedure: Search  /* Setup data structures for calling Backtrack */
Input:
  i, j: indices in trees T1 and T2 respectively
return
 List of node pairs representing the best found mapping of the nodes of S(i) to the nodes of S(j)
Begin
  Let L be the list of pairs (p,q) where p is a descendent of i and q is a descendent of j
  Sort L according to MatchMerit(p,q)
  Set BestSolution =  empty list
  Set CurrentSolution = empty list
  Set BestCost = infinity
  Backtrack(0 /* index*/, L, 0 /* CurrentCost*/)
  return BestSolution
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
  CurrentCost: sum of the cost of the elements in CurrentSolution
Begin
  if ( no element of L can be added to CurrentSolution ) /* Base case */
    if ( CurrentCost + cost of deleted subtrees < BestCost )
       BestSolution = CurrentSolution
       BestCost = CurrentCost
       return

  foreach element l = (m,n) in L starting at index
    if ( CurrentSolution already contains m, n or any of their ascendants or descendents )
        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 MatchMerit
    if ( E + NewCost >= BestCost )
       continue
    Add l to CurrentSolution
    Backtrack(index+1, L, NewCost)
    Remove l from CurrentSolution
End

Procedure: GetBestMatching /* Deduce the optimal mapping */
Input:
  i, j: pair of nodes that belong to the best possible mapping between the two trees
Begin
  foreach element e = (m, n) in BestSuccessor[i][j]
    Add e to BestGlobalMatch
    GetBestMatching(m, n)
End
```

Figure 2: Pseudo-code of the algorithm: parameter R and forcing/preventing matches are not shown here.

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 unmatched 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 a standard algorithm to find the longest common subsequence [23].

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 $DESC(i) \times DESC(j)$ with minimal cost. This is done by trying to add to Q one element of $DESC(i) \times DESC(j)$ such that the new element in Q is consistent with the previous elements (no same node can be matched to two 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 intractable 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.

MDIR 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. Handling non-zero values of B allows our algorithm to detect hierarchical moves. MDIR is guaranteed to find the optimal matching within the constraints of the bound B, provided it is allowed to run long enough.

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. Although bound R removes the guarantee of optimality by limiting the number of recursive calls, the algorithm still obtains good results empirically. 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. This allows the algorithm to return an optimal match even if the backtracking is interrupted during the computation of cost matrix entries corresponding to matches that are not part of the optimal solution.

Forcing and Preventing Matches. MDIR also supports 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 done by assigning a large cost to the corresponding entry in the cost matrix $C[i][j]$. Forcing a match between two nodes is more difficult, due to the necessity of avoiding the deletion of the forcibly matched nodes and at the same time allowing the deletion of some of their ancestors. Additional details can be found in [1].

Runtime and Memory Complexity. An upper bound on the running time of the MDIR 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 $F(p) = max\{f(x,p): x \in X\}$. Then MDIR has a worst case running time of $O((2*F(p))!\ N^2)$. In our implementation, pruning the search tree by using both tree structure and 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 $O(K N^2)$, with K a large constant. In comparison, THP has a worst case running time of $O(d^3 N^2)$ where d is the maximum degree of a tree and $d << N$ [22]. Regarding memory requirements, both THP and MDIR can be implemented in $O(N^2)$ space at the expense of implementation complexity. We implemented THP in $O(d N^2)$, and MDIR in $O(b N^2)$, where b is a large constant factor.

3.3. Empirical Evaluation

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, including both internal and leaf nodes; 4) rename a number of nodes in the copy; 5) and finally, compare the two trees using THP and MDIR. The deletion operations in the middle of the tree correspond to the restricted moves that MDIR detects. Additional details can be found in the companion technical report [1].

The length of an optimal edit script must necessarily be equal to the sum of the number of deletion and the number of renaming operations. Table 1 shows for different tree node sizes, the length of the optimal edit script, the length of the actual edit script and the running time (in seconds) for both THP and MDIR.

On average, THP produced edit scripts suboptimal by about 120% whereas MDIR produced edit scripts sub-optimal by about 7%. In the worst case, THP produced a suboptimal edit script by about 400% whereas MDIR'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. MDIR'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. MDIR produced good results with most trees, even when the optimal edit script involved 2/3 of the number of nodes. With up to 85% of the nodes renamed and no deletions, MDIR produced edit scripts within less than 1% of the optimal script on trees of 640 nodes, showing that it can recover the mapping from tree structure alone.

Table 1: Evaluation of MDIR (R = 100,000).

Case	# Nodes	Ideal Ops	THP		MDIR	
			Ops	Secs	Ops	Secs
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 improved match quality comes at a heavy runtime cost: MDIR was about 60 times slower than THP on average, with bound R set 100,000. As predicted, setting bound R to 5,000 produced slightly sub-optimal edit scripts for a noticeably reduced running time (See [1] for additional empirical data when varying R).

In summary, MDIR has a dramatically improved accuracy over THP and an acceptable non-interactive performance for most common usage scenarios. Unlike optimal graph matching algorithms, it can scale to thousands of nodes and can handle realistic architectural views, as will be demonstrated by the case studies.

4. General Approach to Synchronization

We use the tree algorithm to synchronize hierarchical graphs corresponding to C&C views. The structural information in a C&C view is represented as a cross-linked tree structure that mirrors the hierarchical decomposition of the system. The tree also includes some redundant information to improve the accuracy of the structural comparison: for instance, the subtree of a node corresponding to a port includes all the port's involvements, i.e., all components (and their ports) reachable from that port. Cross-links refer back to the defining occurrence of each element and allow the user to navigate the architectural graph. Each element is decorated with properties (such as type information). The type information, if provided, is used to build a matrix of incompatible elements that may not be matched. Additional constraints can be user-specified.

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, while increasing the size of the corresponding trees, 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 work around these inconsistent matches using two passes. During the first pass, we synchronize the strictly hierarchical information corre-

sponding to the system decomposition, i.e., components, ports and representations. During the second pass, we synchronize the edges in the graph corresponding to the general graph. The post-processing step is simple since the mapping between the nodes in the two graphs is known at that point.

Synchronization is a five-step process: 1) setup the synchronization; 2) optionally view and match types; 3) view and match instances; 4) optionally view and modify the edit script; 5) confirm and optionally apply the edit script. Because Steps 1 and 5 are straightforward, we will only discuss Steps 2-4. In Step 2, manually matching the type structures between the two views (see Figure 3) can produce semantic information that speeds up the comparison but is otherwise optional. It can also reduce 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 by: a) building tree-structured data from the two C&C views; b) using tree-to-tree correction to identify matches and structural differences (Figure 4); and c) obtaining an edit script that can be used to merge the two views.

The differences found during structural matching are shown in each tree by overlaying icons on the affected elements (see Figure 4). 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. Various features can restrict the size of the trees and help reduce the comparison time:

- **Start at Component:** the trees can be rooted at user-selectable components to reduce their sizes;

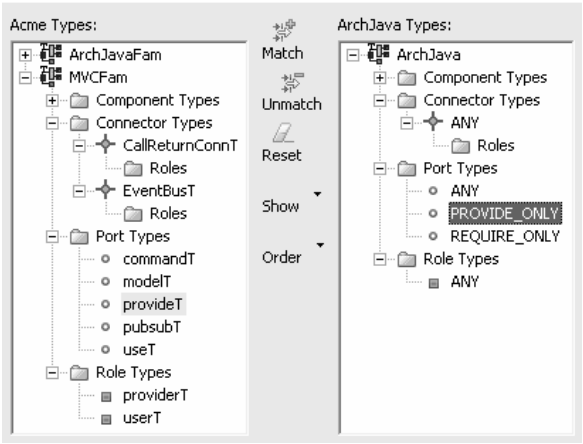

Figure 3: The user manually specifies arbitrary matches, by relating the type hierarchies in both views flattened and shown side-by-side: e.g., the user assigns any ArchJava port with only provided methods the *provideT* Acme type defined in the *MVCFam*, a Model-View-Controller style.

- **Restrict Tree Depth:** the trees can exclude elements beyond a certain user-settable tree depth;
- **Elide Elements:** entire subtrees can be excluded by the user from comparison. Elision is temporary and does not generate any edit actions.

Additional features give the user manual control:

- **Forced matches:** manually force a match between two elements that cannot be structurally matched;
- **Manual overrides:** override any edit action suggested by the comparison.

In Step 4, the edit script produced by tree-to-tree correction is used to produce a common supertree previewing the merged view after the edit actions are applied. 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.

Setting types on elements to be created may affect the processing of the edit script. For instance, when a component instance is assigned a type, it may inherit ports from its assigned type, so the edit script need not create additional ports on the component instance; it may rename a port to match the name declared in the architectural type. The user can rename any architectural element in the edit script. The edit script is also checked for some common problems such as creating any architectural elements without an assigned type.

5. C&C View Synchronization Tools

We now apply the approach just described in two semi-automated tools for C&C view synchronization.

5.1. Case study: Aphyds

The first tool can synchronize an implementation-level architectural view such as one reconstructed using architectural recovery techniques with a conceptual-level architectural view expressed in an Architecture Description Language (ADL). We have chosen Acme [9] and ArchJava [3] to illustrate our approach.

This problem domain clearly requires going beyond insertions and deletions to support renames and moves, as 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 generate a skeleton implementation from the architectural model, connector names and role names are lost during code generation (since ArchJava does not even name those elements). Identifying a renamed element as being deleted and then re-

inserted results in losing type and style information, properties that are crucial for architectural analyses.

We illustrate this tool in a case study on Aphyds [3]. The starting point was an informal drawing (shown in [3]) of the desired conceptual architecture drawn by the developer of the original Java program. In the following discussion, the architect is a third party with no prior experience with the original Java program or the re-engineered ArchJava program. The architect created an Acme model; he represented the *circuitModel* as a single component and added all the computational components to a representation of *circuitModel* (See Figure 5). In the original diagram, some arrows meant to distinguish between control flow and data flow, but the architect decided to show all communication as connectors. The architect let the synchronization tool compare the two views: as he was the least sure about how he represented the *circuitModel* component in Acme, he decided to focus on this component first.

The tool detected a few renames, e.g., ArchJava uses *model* instead of *circuitModel,* and inside that representation, ArchJava uses *globalRouter* instead of *route* (See Figure 4). The architect was particularly intrigued that the Acme representation for *circuitModel* had more connectors than the ArchJava implementation: in Figure 4, the tool only matched the *starConnector* in the middle of Figure 5. The architect investigated this further and confirmed that the dataflow arrows in the informal diagram are not actually in the implementation, so he accepted the edit actions to delete the extra connectors from the Acme model. Finally, the tool helped with detecting additional divergences such as an undocumented bi-directional communication between two components and an entirely missing subsystem.

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 4) with MDIR 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.

5.2. Case Study: Duke's Bank

Two architects will often differ in their use of hierarch: some components expressed at the top level in one C&C view can be nested within another component in some other C&C view. For example, one architect may use hierarchy to hide certain decisions from some parts of the system but a designer may flatten the hier-

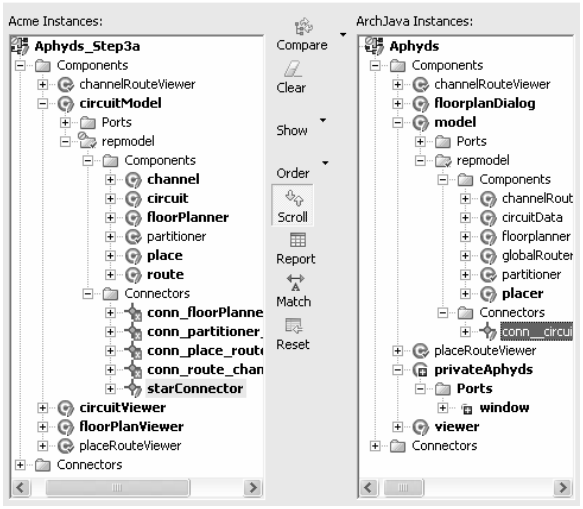

Figure 4: Comparison of Acme C&C view (left) and ArchJava C&C view (right): *starConnector* matches a connector in ArchJava with an automatically generated name (highlighted nodes); *privateAphyds* exists in ArchJava but not in Acme. Match (✓), Insert (+), Delete (x), Rename (/)

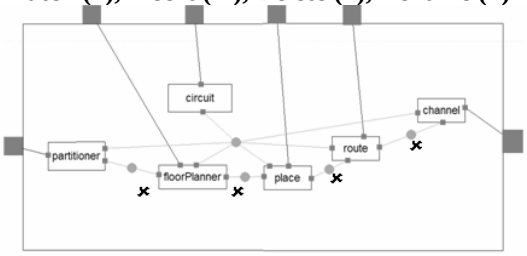

Figure 5: Acme representation of the *circuitModel* component. Extra connectors are marked with ✖.

archy for efficiency reasons. In Acme, this corresponds to replacing an architectural element with its representation. For our next case study, the subject system is Duke's Bank, a simple Enterprise JavaBeans (EJB) banking application. The architect wanted to compare the architecture presented in the documentation with the actual architecture discovered by instrumenting the running system as explained in [26].

The architect converted an informal diagram [21] into an Acme model (See Figure 6). Since the model recovered by instrumentation includes each session and entity bean instance created at runtime, the architect post-processed it to consolidate multiple instances into one instance (See Figure 7) to make it comparable to the documented architecture where each component instance represents a number of run-time components.

The synchronization tool was able to match all the elements between the two views despite the large number of renames. The tool correctly detected the moves corresponding to replacing the *container* component in one view with its representation in the other view. The

tool also detected an undocumented port on *Account_Controller_Bean* communicating with the *DB* component through a *DbWriter* connector. Figure 6 does not show such a connection: indeed, in EJB, all database access is through entity beans. The tool found a violation of the specification in Sun's own example!

On an Intel Pentium4® CPU 3GHz with 1GB of RAM, MDIR 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.

6. Related Work

Several algorithms have been proposed for differencing hierarchical information, in the context of program differencing (e.g., JDIFF [4], Dex [18]) and design differencing (e.g., UMLDiff [25]). These algorithms are based on the assumption that the entities they are trying to match are uniquely named and many nodes match exactly. This enables recognizing the unchanged entities first and using them as landmarks, which makes for efficient algorithms. However, these algorithms are unable to match nodes based on structure alone or based on structure and highly non-unique semantic information such as entity types. For instance, a heuristic solution with a worst-case $O(N^3)$ supporting arbitrary move, copy and glue operations [6] was tested on instances where more than 80% of the nodes matched exactly. As a result, these algorithms are less suitable for comparing architectural views, as they will perform poorly when all the nodes are renamed, or when most of the renamed nodes are concentrated in one area of the tree (e.g., when entire subtrees are renamed). This may be atypical when comparing two versions of a given program or a design model (at a given level of abstraction), but in our architectural views, most names are transient or automatically generated. Both THP and MDIR would still work even in the total absence of semantic information (i.e., using tree structure only). For instance, in both case studies, our inputs had more than half of their nodes renamed. Finally, none of these algorithms offer the ability to manually force or prevent matches. In particular, it may be possible to easily add the ability to prevent matches to some of them (e.g., JDIFF), but adding the ability to force matches could be substantially more complicated.

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. Algorithms based on tree alignment can detect unbounded deletes [11] and can generalize to more than two trees, something not easily done with tree edit distance algorithms. But the memory requirements of tree alignment algorithms, for the tree sizes and branching factors that are typical of our inputs, would be several orders of magnitude higher than those of MDIR– $O(2^{2d} N^2)$ where d is the maximum degree of the tree.

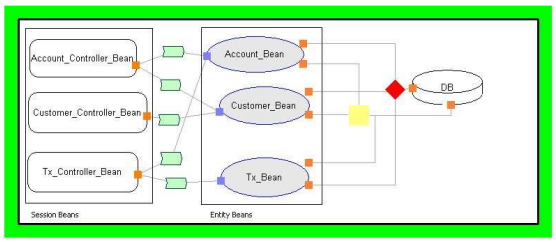

Figure 6: Documented architecture in Acme: components are inside an EJB *container* (thick border). Session and Entity Beans are grouped.

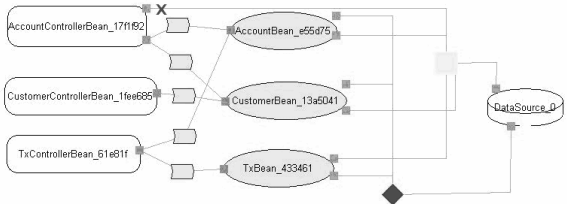

Figure 7: Recovered architecture in Acme: the violation of the specification is marked with x.

Graph Matching Approaches. As mentioned earlier, exhaustive graph matching algorithms based on variations of A* [16] do not scale beyond a few dozen nodes [10]. More scalable, heuristic-based approaches, such as spectral methods perform poorly when the graphs are not nearly isomorphic or may occasionally miss the optimal solution [7]. Others, such as the Similarity Flooding Algorithm (SFA) [14] have a low accuracy (around 50%), while the accuracy of MDIR is above 90%, on a roughly similar range of graph sizes. Furthermore, SFA relies heavily on labels, which are different when the graphs originate from different domains, even if they express the same relationships: "while matching of an XML schema against another XML schema delivers usable results, matching of a relational schema against an XML schema fails" [14].

Probabilistic matching based on label, region, type or position information has been proposed [28], but the approach requires training the evidencers. The authors also admit that using a simple greedy search algorithm does not work in many cases.

Model Transformation. Graph transformation approaches (see [15] and references therein) tackle the same problem using a different set of assumptions. First, in many graph grammars, productions do not

delete vertices and edges, effectively prohibiting insertions and deletions, one of our requirements. Second, graph transformation approaches do not attempt to find the optimal transformation including preserving properties of view elements. Finally, these approaches do not yet offer easy to use *diff*-like tools such as the ones presented in Sections 4 and 5.

7. Conclusions

In this paper, we presented a novel algorithm for differencing and merging tree-structured data that compares favorably to existing algorithms based on empirical evaluation. We used the tree-to-tree correction algorithm to compare and merge hierarchical architectural views specifically component-and-connector (C&C) architectural views. We then presented tools that incorporate the algorithm and showed how our relaxed assumptions match more closely the problem domain. Finally, we evaluated the tools in case studies and showed the practicality of the approach to find interesting architectural differences.

Acknowledgments. This work was supported in part by NASA cooperative agreements NCC-2-1298 and NNA05CS30A, NSF grants CCR-0204047 and CCF-0546550, 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.

8. References

[1] Abi-Antoun, M., Aldrich, J., Nahas, N., Schmerl, B. and Garlan, D. Differencing and Merging of Architectural Views. Technical Report CMU-ISRI-05-128, 2005.

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

[3] Aldrich, J., Chambers, C. and Notkin, D. ArchJava: Connecting Software Architecture to Implementation. In Proc. International Conference on Software Engineering, 2002.

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

[5] 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.

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

[7] 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.

[8] 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.

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

[10] Hlaoui, A. and Wang, S. A new algorithm for graph matching with application to content-based image retrieval. In Proc. Joint IAPR Int. Workshops SSPR and SPR, 2002.

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

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

[13] Mehra, A., Grundy, J. and Hosking, J. A Generic Approach to Supporting Diagram Differencing and Merging. In Proc. Automated Software Engineering, 2005.

[14] Melnik, S., Garcia-Molina, H. and Rahm, E. Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching. In Conference on Data Engineering (ICDE) and Extended Technical Report, 2002.

[15] Mens, T. A taxonomy of model transformation and its application to graph transformation technology. In Proc. Int'l Workshop GraMoT, 2005.

[16] Messmer, B.T. Efficient Graph Matching Algorithms for Preprocessed Model Graphs, Ph.D. Thesis, University of Bern, 1996.

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

[18] 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.

[19] 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.

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

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

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

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

[24] Wang, Y., Dewitt, D.J. and Cai, J.-Y. X-Diff: An Effective Change Detection Algorithm for XML Documents. In Proc. 19th Intl. Conference Data Enginering (ICDE), 2003.

[25] Xing, Z. and Stroulia, E. UMLDiff: An Algorithm for Object–Oriented Design Differencing. In Proc. Automated Software Engineering, 2005.

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

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

[28] Mandelin, D., Yellin, D. and Kimelman, D. A Bayesian Approach to Architectural Model Matching. In Proc. ICSE, 2006.