Autom Softw Eng (2008) 15: 35–74
DOI 10.1007/s10515-007-0023-3

Differencing and merging of architectural views

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

Received: 20 November 2007 / Accepted: 5 December 2007 / Published online: 22 December 2007

Abstract Differencing and merging architectural views is an important activity in software engineering. However, existing approaches are still based on restrictive assumptions, such as requiring view elements to have unique identifiers or exactly matching types, which is often not the case in many application domains.

We propose an approach based on structural information. We generalize a published polynomial-time tree-to-tree correction algorithm that detects inserts, renames and deletes, into a novel algorithm that additionally detects restricted moves. Our algorithm also supports forcing and preventing matches between view elements.

We incorporate the algorithm into tools to compare and merge Component-and-Connector (C&C) architectural views. We provide an empirical evaluation of the algorithm. We illustrate the tools using extended examples, and use them to detect and reconcile interesting differences between real architectural views.

Keywords Tree-to-tree correction · View synchronization · Graph matching

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

This article is an expanded version of the following paper: Abi-Antoun, M., Aldrich, J., Nahas, N., Schmerl, B., and Garlan, D: 2006, 'Differencing and Merging of Architectural Views'. In: *Proceedings of the 21st IEEE International Conference on Automated Software Engineering*, pp. 47–58.

M. Abi-Antoun (✉) · J. Aldrich · N. Nahas · B. Schmerl · D. Garlan
Carnegie Mellon University, Pittsburgh, PA 15213, USA
e-mail: marwan.abi-antoun@cs.cmu.edu

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 (Chen et al. 2003). Once the system is implemented, an architect may want to compare a conceptual as-designed C&C view with an as-built C&C view retrieved from the implementation using various architectural recovery techniques (Eixelsberger et al. 1998; Murphy et al. 2001; Medvidovic and Jakobac 2006; Schmerl et al. 2006). The architect might be interested in implementation-level violations of the architectural styles or other intent (Abi-Antoun et al. 2005), or in a change impact analysis (Krikhaar et al. 1999). A runtime analysis could use the difference information to perform architectural repair (Dashofy et al. 2002). Finally, during evolution, the difference information can focus regression testing efforts (Muccini et al. 2005).

Several techniques and tools have been proposed for differencing and merging architectural or design views. Most techniques do not detect differences based on structural information. Many assume that elements have unique identifiers (Alanen and Porres 2003; Ohst et al. 2003; Mehra et al. 2005). Others match two elements if both their labels and their types match (Chen et al. 2003), which is often infeasible when dealing with views at different levels of abstraction. Many techniques detect only a small number of differences. For instance, ArchDiff only detects insertions and deletions (van der Westhuizen and van der Hoek 2002; Chen et al. 2003), possibly leading to the loss of information when elements are renamed or moved across the hierarchy. Tracking changes, using element-level versioning, helps infer high-level operations, such as merges, splits or clones, in addition to the low-level operations, such as inserts and deletes (Jimenez 2005; Roshandel et al. 2004). But such an approach requires building new tools or changing existing tools, and cannot handle legacy architectural models.

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

- Differencing and merging architectural views based on structural information. Tree-to-tree correction algorithms identify matches, and classify the changes between the two views. Optional type information prevents matches between incompatible view elements, thus speeding execution and improving match quality;
- A novel polynomial-time tree-to-tree correction algorithm. The algorithm adapts a recently published optimal tree-to-tree correction algorithm for unordered labeled trees that detects renames, inserts and deletes (Torsello et al. 2005), and generalizes it to additionally detect restricted moves. Our algorithm also supports forcing and preventing matches between elements in the views under comparison;
- An empirical evaluation of the novel algorithm, and a comparison with the previously published algorithm;
- A set of tools for the semi-automated synchronization of C&C views using these algorithms. One tool can synchronize an as-designed C&C view with an as-built C&C view retrieved from an implementation. Another tool can more generally synchronize two C&C views, regardless of how they were obtained;

- An evaluation of the tools in extended examples to find and reconcile interesting differences in real architectural views.

The paper is organized as follows: Section 2 describes the challenges in differencing and merging architectural views, the underlying assumptions, and the limitations of our approach. Section 3 describes our novel tree-to-tree correction algorithm. Section 4 presents an empirical evaluation of the algorithm. In Section 5, we use the algorithm to synchronize architectural C&C views. Section 6 illustrates the approach using extended examples on real architectural views. Finally, we discuss related work in Section 7 and conclude.

2 Architectural view differencing

Software architects rely on multiple architectural views, where a view represents a set of system elements and the relationships between them. Views can be of different viewtypes, where each viewtype defines the element types and the relationship types used to describe a software system from a particular perspective (Clements et al. 2003). Since a view is generally 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, i.e., it produces a set of edit operations that model inconsistencies by transforming one graph into another (Conte et al. 2004). Typical graph edit operations include the deletion, insertion and substitution of nodes and edges. Each edit operation is assigned a cost. The costs are application-dependent, and model the likelihood of the corresponding inconsistencies. Typically, the more likely a certain inconsistency is, the lower is its cost. 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 (Conte et al. 2004). Unique node labels enable processing graphs efficiently (Dickinson et al. 2004), which explains why many approaches make this assumption, e.g., (Alanen and Porres 2003; Ohst et al. 2003; Mehra et al. 2005). Optimal graph matching algorithms, i.e., those that can find a global minimum of the matching cost if it exists, can handle at most a few dozen nodes (Messmer 1996; Conte et al. 2004). 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" (Melnik et al. 2002).

Several efficient algorithms have been proposed for trees, a strict hierarchical structure, so our approach focuses on hierarchical architectural views. While not all architectural views are hierarchical, many use hierarchy 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 (Clements et al. 2003). Many approaches are hierarchical (Apiwattanapong et al. 2004; Raghavan et al. 2004; Xing and Stroulia 2005). 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, as in other approaches, e.g., (Chen et al. 2003; Mehra et al. 2005). As mentioned earlier, this assumption alone enables the use of exact and scalable algorithms that can handle thousands of nodes (Dickinson et al. 2004). Unfortunately, architectural view elements often do not have unique identifiers.

No ordering. In the general case, an architectural view has no inherent ordering amongst 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., (Shasha and Zhang 1997). In comparison, tree-to-tree correction for unordered trees is MAX SNP-hard (Zhang and Jiang 1994). Some algorithms for unordered trees achieve polynomial-time complexity, either through heuristic methods, e.g., (Chawathe and Garcia-Molina 1997; Wang et al. 2003; Raghavan et al. 2004), or under additional assumptions, e.g., (Torsello et al. 2005).

Renames. A synchronization approach must of course handle elements that are inserted and deleted, as supported by ArchDiff (Chen et al. 2003). But effective synchronization must also go beyond insertions and deletions, and support renames.

Name differences between two C&C views can arise for a variety of reasons. For instance, the architect may update a name in one view, and forget to update another view. 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 modules (Ammann and Cameron 1994). Furthermore, developers tend to avoid using names that may be in use by an implementation framework or library, a minor detail for the architect. Finally, architectural view elements may not have persistent names or their names may be generated automatically by tools.

This suggests that an algorithm should be able to match renamed elements. Identifying an element as being deleted and then inserted when, in fact, it was renamed, would result in losing crucial style and property information about the element, even if this produces structurally equivalent views. These architectural properties, such as throughput, latency, etc., are crucial for many architectural analyses, e.g., (Spitznagel and Garlan 1998). In the following discussion, a *matched* node is a node with either an *exactly matching* label or a *renamed* label.

Hierarchical moves. Architects often use hierarchy to manage complexity. In general, two architects may differ in their use of hierarchy: a component expressed at the top level in one view could be nested within another component in some other view. This suggests that an algorithm should detect sequences of *internal node deletions* in the middle of the tree, which result in nodes moving up a number of levels in the hierarchy. An algorithm should also detect sequences of *internal node insertions* in the middle of the tree, which result in nodes moving down the hierarchy, by becoming children of the inserted nodes, as shown in Figure 1.

Manual overrides. Structural similarities may lead a fully automated algorithm to incorrectly match top-level elements between two trees and produce an unusable out-

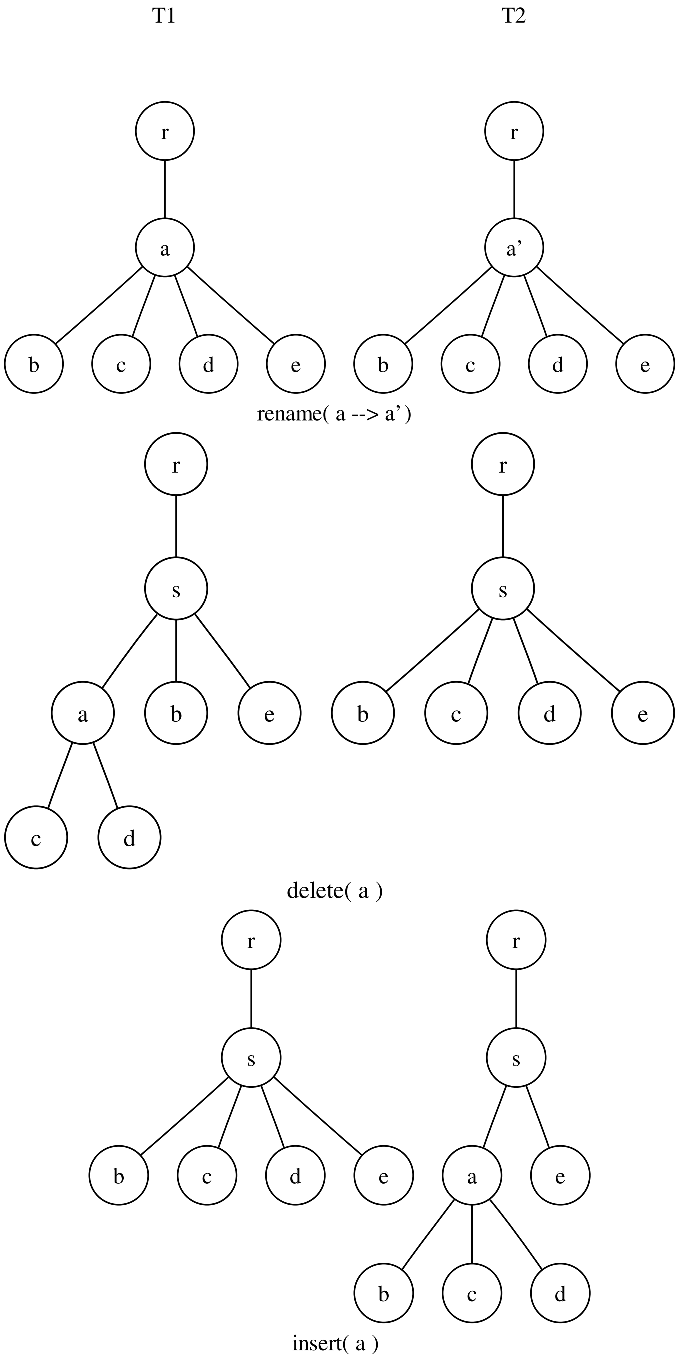


Fig. 1 Tree edit operations

put. Because of the dependencies in the mapping, one cannot easily adjust incorrect matches after the fact. Instead, we added a feature not typically found in tree-to-tree correction algorithms. The feature allows the user to force or prevent matches between selected view elements. The algorithm then takes these constraints into account to improve the overall match. The user can specify any set of constraints, as long as they preserve the ancestry relation between the forcibly matched nodes. In particular, 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 be untyped or have different or incompatible type systems. This is often the case when comparing views at different levels of abstraction, such as an as-designed conceptual-level view with an as-built implementation-level view. Therefore, an algorithm should not rely on matching type information, and should be able to recover a correct mapping from structure alone if necessary, or from structure and type information if type information is available. An algorithm could however take advantage of type information, when available, to prune the search space by not attempting to match elements of incompatible types.

If the view elements are represented as typed nodes, at the very least, an algorithm should not match nodes of incompatible types, e.g., it should not match a connector x to a component y. If architectural style information is available, additional architectural types may be available and could be used for similar purposes. For instance, an algorithm can avoid matching a component of type `Filter`, from a Pipe-and-Filter architectural style, to a component of type `Repository`, from a Shared-Data architectural style (Shaw and Garlan 1996).

Disconnected/stateless operation. For maximum generality, we assume a disconnected and stateless operation. A few approaches require monitoring or recording the structural changes while the user is modifying a given view (Jimenez 2005; Roshandel et al. 2004).

Comparable views. The two views under comparison have to be somewhat structurally similar. When comparing two completely different views, an algorithm could trivially delete all elements of one view, and then insert them in the other view. In addition, the two views must be of the same viewtype, and must be comparable without any view transformation. Checking the consistency of different but related views, such as a UML class diagram and a UML sequence diagram, is a problem in *view integration* (Egyed 2006), and is outside the scope of this paper.

No merging/splitting. Our approach does not currently detect the merging or splitting of view elements. Merging and splitting are common practice, but are difficult to formalize, since they affect connections in a context-dependent way (Erdogmus 1998). We leave merges and splits to future work.

3 Tree-to-tree correction

In this section, we describe a novel algorithm for unordered labeled trees, MDIR (Moves-Deletes-Inserts-Renames), which generalizes a recent optimal tree-to-tree correction algorithm (Torsello et al. 2005), which we will refer to as THP. We also implemented THP for experimental comparison with MDIR (Section 4).

3.1 Problem definition

We first give an unambiguous definition of the problem, adapted from Sasha and Zhang (1997). 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 nodes of T. We define a triple $(\mathbb{M}, T_1, T_2)$ to be a mapping from T_1 to T_2, where $\mathbb{M}$ is any set of pairs of integers (i, j) satisfying the following:

1. $1 <= i <= |T_1|, 1 <= j <= |T_2|$;
2. For any pair of (i_1, j_1) and (i_2, j_2) in $\mathbb{M}$ with:
   - $i_1 = i_2$ if and only if $j_1 = j_2$ (one-to-one)
   - $T_1[i_1]$ 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 $\mathbb{M}$ instead of $(\mathbb{M}, T_1, T_2)$ 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. It 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 $\mathbb{M}$ between trees T_1 and T_2. From this mapping, we can deduce an *edit script*, i.e., 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. Finally, 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 changes "banana" into "ananas" with a cost of two (Wagner and Fischer 1974). 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 shown in Figures 2 and 3. 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, 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 $S(j)$ and 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

```
    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
    $(m, n) \in BestSuccessor[i][j]$ 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))
    L(i, j): cost of string-to-string correction to change LABEL(i) to LABEL(j)

MDIR($T_1$, $T_2$)
    Postorder $T_1$ and $T_2$ nodes
    for $i \leftarrow 1$ to $T_1.size$
        do for $j \leftarrow 1$ to $T_2.size$
            do BestSuccessor[i][j] = SEARCH(i, j)
               CostMatrix[i][j] = COST(BestSuccessor[i][j]) +L(i,j)
    GETBESTMATCHING($T_1.size$, $T_2.size$)

SEARCH($i$, $j$)
    ▷ $i$, $j$: indices in trees $T_1$ and $T_2$ respectively
    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
```

Fig. 2 Pseudo-code of the algorithm

a subset of $S(i)$ into $S(j)$ with least cost. 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. 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. 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

```
BACKTRACK($index, L$)
  ▷ Search for a good mapping between subtrees
  ▷ $index$: 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 $CurrentSolution$
  if ( no element of $L$ can be added to $CurrentSolution$ ) ▷ Base case
     then if ( $CurrentCost$ + cost of deleted subtrees $<$ $BestCost$ )
           then
              $BestSolution = CurrentSolution$
              $BestCost = CurrentCost$
              return
  foreach element $l = (m, n)$ in $L$ starting at $index$
        do
           if ( $CurrentSolution$ already contains $m$, $n$
           or any of their ascendants or descendents )
              then continue
           if ( adding $l$ to current mapping violates bound $B$ )
              then continue
           Add cost of match to $CurrentCost$ to obtain $NewCost$
           Get a lower bound $E$ of remaining cost using $MatchMerit$
           if ( $E + NewCost >= BestCost$ )
              then continue
           Add $l$ to $CurrentSolution$
           BACKTRACK($index + 1, L, NewCost$)
           Remove $l$ from $CurrentSolution$


GETBESTMATCHING($i, j$)
  ▷ Deduce the optimal mapping
  ▷ $i, j$: pair of nodes belonging to best possible
  ▷ mapping between the two trees
  foreach element $e = (m, n)$ in $BestSuccessor[i][j]$
        do Add $e$ to $BestGlobalMatch$
           GETBESTMATCHING($m, n$)
```

Fig. 3 Pseudo-code of the algorithm (continued)

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 (Wagner and Fischer 1974).

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, i.e., 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 when it determines that there are no more valid pairs to add, or the cost of the current branch will be too large to match the best solution 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. It is also 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. This also enables performing many operations 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 loop over an array.

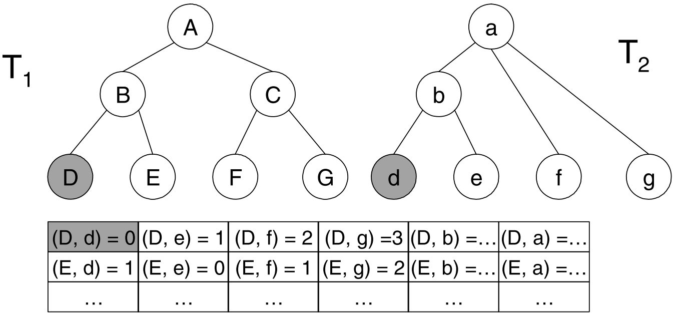
MDIR can be considered a generalization of THP, because THP only handles the case where $B = 0$, i.e., only the children of a node can be in a successor set of that node. This produces a fully polynomial time algorithm that is typically much faster than our generalized algorithm. Handling non-zero values of B allows MDIR 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.

In principle, one could use the same implementation for both THP and MDIR, and adjust MDIR's parameters to simulate THP, e.g., by modifying the SEARCH procedure in Figure 2 accordingly. However, we currently have two separate implementations for MDIR and THP, with some shared procedures.

It is necessary to limit the running time on trees with more than a few hundred nodes, and when the average degree of a non-leaf node is greater than four. We enforce 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, MDIR still obtains good results empirically. Since MDIR 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 that correspond to good matches, and is interrupted only when the match is not good. This allows MDIR 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.

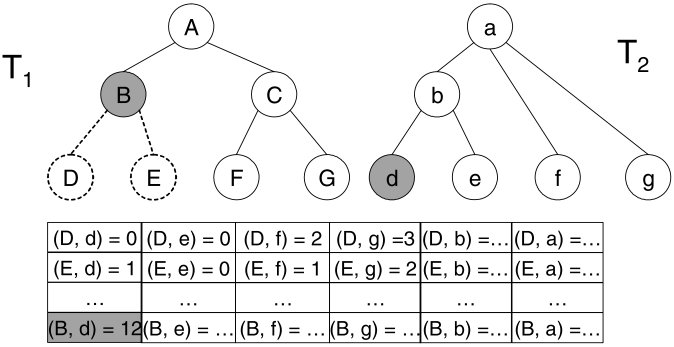
3.3 Illustrative example

In this section, we illustrate the MDIR algorithm on a small example. MDIR exhaustively computes from bottom to top the cost of mapping each node in T_1 to every



(D, d) = 0	(D, e) = 1	(D, f) = 2	(D, g) =3	(D, b) =…	(D, a) =…
(E, d) = 1	(E, e) = 0	(E, f) = 1	(E, g) = 2	(E, b) =…	(E, a) =…
…	…	…	…	…	…

Fig. 4 Cost(D,d) = cost of editing label of D to d, i.e., the measure of similarity between the labels, in this case 0



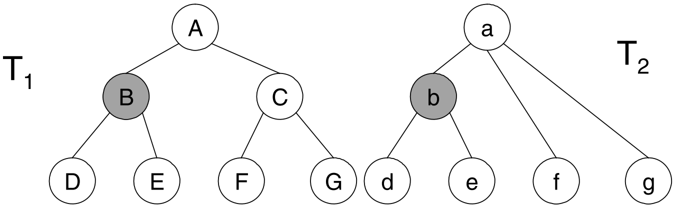
(D, d) = 0	(D, e) = 0	(D, f) = 2	(D, g) =3	(D, b) =…	(D, a) =…
(E, d) = 1	(E, e) = 0	(E, f) = 1	(E, g) = 2	(E, b) =…	(E, a) =…
…	…	…	…	…	…
(B, d) = 12	(B, e) = …	(B, f) = …	(B, g) = …	(B, b) =…	(B, a) =…

Fig. 5 Cost(B,d) $=$ Cost(deleting B's children) $+$ Cost(editing B's label). Assuming the cost of a deletion is 5 times a unit cost, Cost(B, d) $=$ Cost(deleting D) $+$ Cost(deleting E) $+$ Cost(editing B's label) $= 5 + 5 + 2$

other node in T_2. The computed costs are stored in a cost matrix. Following the dynamic programming paradigm, MDIR uses the comparison on the high depth nodes to compare the low depth nodes. The example also illustrates the usefulness of the *successor set* approach, since bipartite matching cannot match subtree nodes, because of the need to preserve the hierarchical constraints.

MDIR starts by computing the cost of matching D to d (Figure 4). Similarly, MDIR computes the costs of matching $(D, e), (D, f), (D, g), \ldots, (E, d), (E, e)$, (E, g). In Figure 5, MDIR computes the cost of matching B to d. Then, MDIR computes the cost of matching B to b (Figure 6). This requires knowing the cost of the optimal *successor set mapping* for B and b. At this point, MDIR has computed the costs of matching every descendent of B to any node in the second tree, because of the post-ordering of the trees.

The optimal successor set mapping corresponding to the pair (B, b) is computed as follows (Figure 7). First, take all the node pairs, where the first item is a descendent of B, and the second item is a descendent of b, i.e., the set $\{(D, d), 4(D, e), (E, d), (E, e)\}$. The optimal mapping will clearly be a subset of this set. To obtain that optimal mapping, we examine all mappings — except the ones that



(D, d) = 0	(D, e) = 1	(D, f) = 2	(D, g) =3	(D, b) =…	(D, a) =…
(E, d) = 1	**(E, e) = 0**	(E, f) = 1	(E, g) = 2	(E, b) =…	(E, a) =…
…	…	…	…	…	…
(B, d) = 12	(B, e) = …	(B, f) = …	(B, g) = …	(B, b) = 0	(B, a) =…

Fig. 6 Cost(B,b) = Cost(successor set mapping of (B,b)) + Cost(editing the label of B to b). Cost(D,d) and Cost(E,e) have been previously computed, thus Cost(B,b) = Cost(D,d) + Cost$(E,e) + 0$

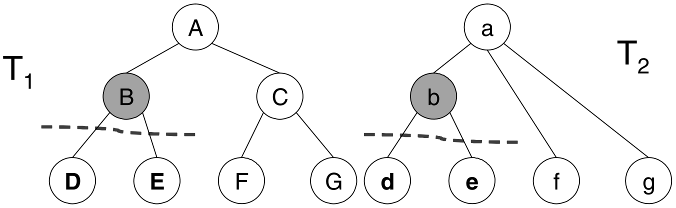


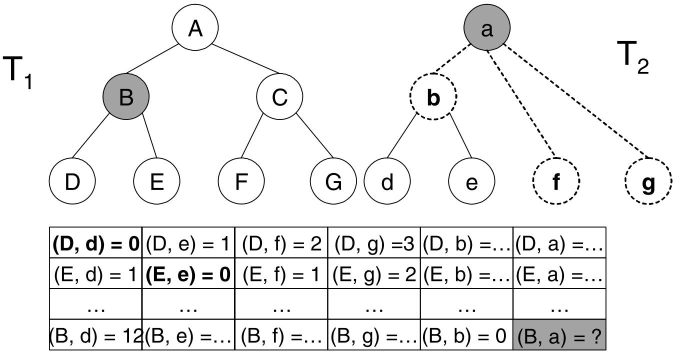
Fig. 7 Computing the cost of matching B to b requires the *successor set mapping* of the pair (B, b). The *successor set mapping* of (B, b) is the set $\{(D, d), (E, e)\}$

have been pruned because the bounds on their cost showed they could not be optimal. The other constraint is: if (x, y) is a pair in a mapping,neither x, nor y, nor any of their ascendents or descendents, can appear in any other pair in the same mapping. Thus, the optimal successor set mapping for (B, b) is $\{(D, d), (E, e)\}$. Finally, in Figure 8, MDIR computes the cost of matching B to a.

At the end of this phase, MDIR has determined the "best" successor set mapping, and stored it for the next phase, when MDIR will retrieve the best matches. MDIR could avoid keeping the optimal successor set mapping for each node pair in the first phase, to reduce the space complexity to $O(N^2)$. But it is simpler conceptually to store this information, and this is how we currently implemented MDIR.

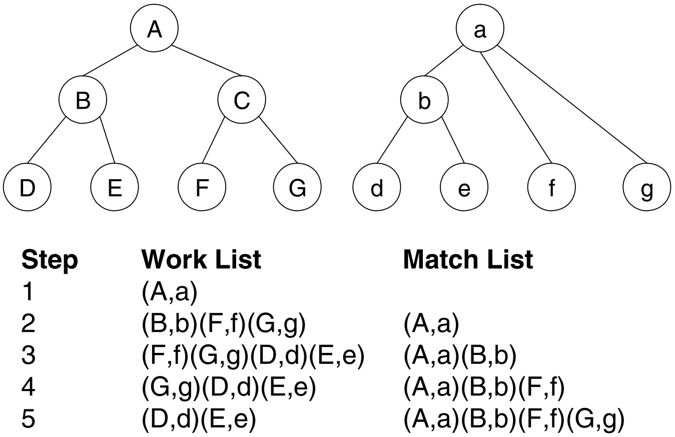
In the second phase, MDIR uses a recursive procedure to compute the match list, i.e., to determine what node corresponds to what other node. MDIR uses the following recursive formulation. The list of matches for subtree pair rooted at (x, y) consists of (x, y), in addition to the list of matches of each pair in the successor set mapping of (x, y).

In Figure 9, MDIR starts with (A, a). The successor set mapping of (A, a) is $\{(B, b), (F, f), (G, g)\}$. So, MDIR first adds (A, a) to the match list, and then adds the pairs (B, b), (F, f), and (G, g) to the work list. Then, MDIR pops (B, b) from the work list, adds it to the match list, and adds to the work list the successor set



(D, d) = 0	(D, e) = 1	(D, f) = 2	(D, g) =3	(D, b) =…	(D, a) =…
(E, d) = 1	**(E, e) = 0**	(E, f) = 1	(E, g) = 2	(E, b) =…	(E, a) =…
…	…	…	…	…	…
(B, d) = 12	(B, e) =…	(B, f) =…	(B, g) =…	(B, b) = 0	(B, a) = ?

Fig. 8 COST(B,a) = COST(successor set mapping of (B, a)) + COST(editing the label of B to a) + COST(deleting b, f and g)



Step	Work List	Match List
1	(A,a)	
2	(B,b)(F,f)(G,g)	(A,a)
3	(F,f)(G,g)(D,d)(E,e)	(A,a)(B,b)
4	(G,g)(D,d)(E,e)	(A,a)(B,b)(F,f)
5	(D,d)(E,e)	(A,a)(B,b)(F,f)(G,g)

Fig. 9 Computing the match list

(B, b), namely, (D, d) and (E, e). Next, MDIR pops (F, f) from the work list, adds it to the match list, and proceeds similarly.

3.4 Forcing and preventing matches

Manual overrides are not a standard operation in most tree-to-tree correction algorithms. MDIR has 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 enough to first prevent the match of either of these two nodes with any node other than the required one, and second, make the cost of deletion and insertion of these nodes very high. This would be the case if the algorithm did not have the additional constraint concerning the distance to the subtree root. Because of this constraint, it is often necessary to delete entire subtrees at a time, when no match can be found for any node close enough to the subtree root.

So, we have to avoid deleting one of the nodes involved in the forced match, during 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 sub-optimal edit script, because it is possible that a few ancestors were deleted, while the forcibly matched node was not 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. 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. The base case of the recursive BACKTRACK procedure enforces this constraint. 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. Instead, BACKTRACK 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 the latter manageable.

3.5 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's worst case running time is $O((2 * F(p))!N^2)$.

The average case is considerably faster than the worst case, in our implementation, due to the following strategies. We prune the search tree by using both the tree structure and any semantic information, such as optional type information. We also limit the running time by returning a possibly suboptimal solution.

In practice, the observed runtime is $O(KN^2)$, where K is a large constant. In comparison, THP has a worst case running time of $O(d^3N^2)$, where d is the maximum degree of a tree and $d << N$ (Torsello et al. 2005). Regarding memory requirements, both THP and MDIR could be implemented in $O(N^2)$ space, at the expense of additional complexity. Our current THP implementation requires $O(dN^2)$, and MDIR requires $O(bN^2)$, where b is a large constant factor.

4 Empirical evaluation

An empirical evaluation of the accuracy of the MDIR algorithm is necessary because bounds B and R remove the guarantee of optimality. We generated the test data as follows: 1) generate a random tree with random labels taken from a pool of 10 possible names, so as they are non-unique; 2) copy the tree; 3) delete a random number

Table 1 Evaluation of MDIR ($R = 100K$)

Case	#	Ideal Ops	THP		MDIR	
	Nodes		Ops	Secs	Ops	Secs
Renames	640	569	770	2	569	64
	1280	857	1509	7	963	442
Deletes	640	492	701	2	492	50
	1280	1113	1397	5	1114	169
Internal Deletes	640	441	1076	3	1093	215
	1280	652	2407	9	735	471
Node Degree	640	288	712	2	288	65
	1280	576	1194	10	576	248

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. We ran MDIR once with bound $R = 100K$, and another time with bound $R = 5K$. We left bound B unchanged from its default value in all runs.

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. All numbers were measured on an Intel Pentium 4 CPU 3 GHz with 1.5 GB of RAM.

On average, THP produced edit scripts that are sub-optimal by about 120%, whereas MDIR produced edit scripts that are 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%. The accuracy deteriorated significantly for both MDIR and THP, when using nodes of large degree, 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. So MDIR can recover the mapping from tree structure alone.

The improved match quality comes at a heavy runtime cost. MDIR was about 60 times slower than THP on average, with bound $R = 100K$. As predicted, setting bound R to $5K$ produced slightly sub-optimal edit scripts, but noticeably reduced the running time. On a tree of 1,280 nodes with an optimal edit script of 396 edits, THP produced an edit script of size 1,775 in 7 seconds. MDIR with $R = 100K$ produced an edit script of size 459 in 6 minutes, whereas MDIR with $R = 5K$ produced an edit script of size 479 in 4 minutes. Empirical data with those two different values of R is shown in Figures 10, 11, 12 and 13. Varying the bound R did not have much effect

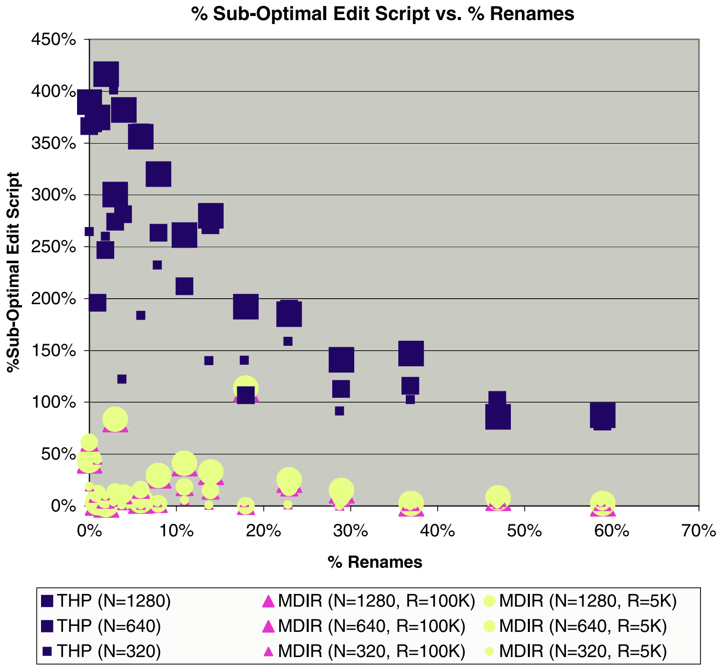


Fig. 10 A comparison of THP and MDIR ($R = 100K$, and $R = 5K$) showing the sub-optimality of the edit script vs. the percentage of renames

on MDIR's precision. Note that all the tree pairs used in those figures have internally deleted nodes, even if this is not the varying parameter.

Figure 10 shows the sub-optimality of the edit script when varying the percentage of renames, for both THP and MDIR. The worst edit script for MDIR was suboptimal by around 100%, whereas THP was off by over 400%. This figure may mislead the reader into thinking that the accuracy of THP increases with the percentage of renames. Of course, it does not. THP detects renames but not internal deletes, so when the percentage of renames in the optimal edit script increases — compared to the other operations, THP's precision seems to improve.

Figure 11 shows the sub-optimality of the edit script when varying the percentage of deletes, for both THP and MDIR. THP generated one edit script that was suboptimal by over 50%, whereas MDIR generated fully optimal edit scripts.

Figure 12 shows the sub-optimality of the edit script when varying the percentage of inserts, for both THP and MDIR. THP generated edit scripts that were suboptimal by around 110% on average, whereas MDIR generated edit scripts that were off by 20% on average.

Figure 13 shows the sub-optimality of the edit script when varying the node degree, for both THP and MDIR. THP generated edit scripts that were suboptimal by over 110% on average, whereas MDIR generated edit scripts that were off by 16% on average.

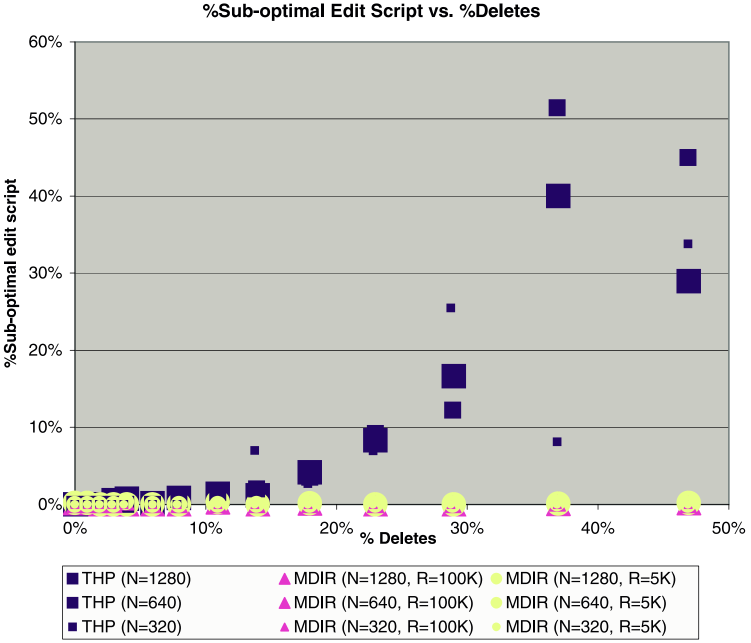


Fig. 11 A comparison of THP and MDIR ($R = 100K$, and $R = 5K$) showing the sub-optimality of the edit script vs. the percentage of deletes

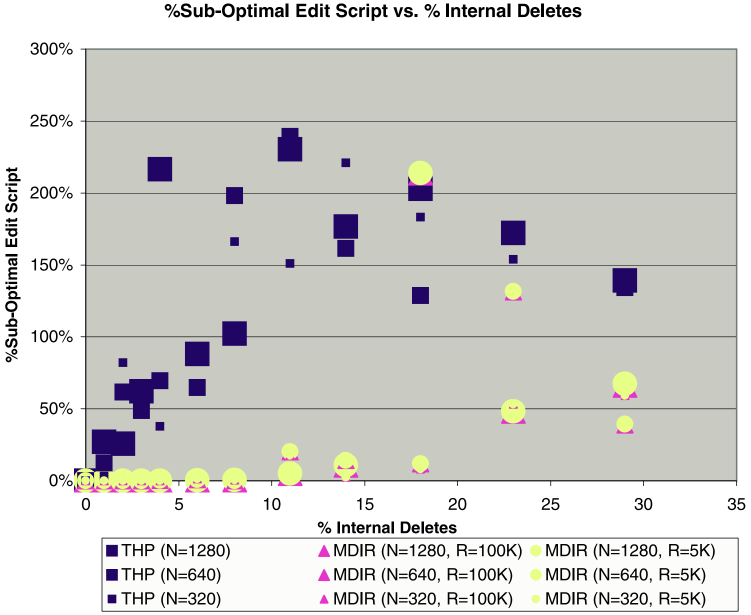


Fig. 12 A comparison of THP and MDIR ($R = 100K$, and $R = 5K$) showing the sub-optimality of the edit script vs. the percentage of internal deletes

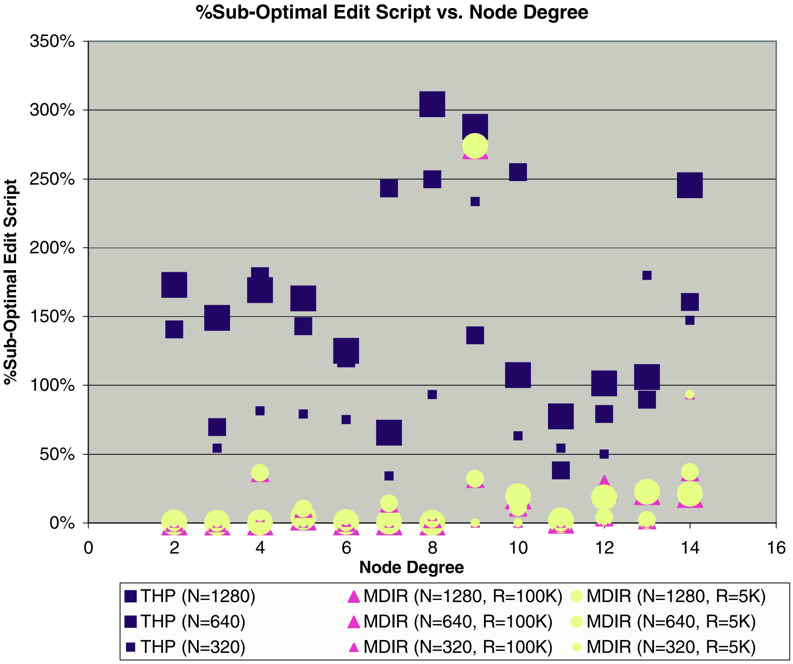


Fig. 13 A comparison of THP and MDIR ($R = 100K$, and $R = 5K$) showing the sub-optimality of the edit script vs. the node degree

Table 2 Random trees generated for the performance evaluation of MDIR

Case	Deletes	Inserts	Internal deletes	Internal inserts	Renames
MDIR_1	20%	20%	10%	10%	40%
MDIR_2	10%	10%	10%	10%	60%
MDIR_3	10%	10%	30%	30%	20%
MDIR_4	5%	5%	15%	5%	15%

The previous data showed mainly the improvement in the precision of the matching. We also wanted some indication on the algorithm's performance and scalability. Ideally, one would generate graphs with edits that follow the probability distribution of typical graph edits. Since we did not have such a distribution, we generated random trees instead, as before, but with various combinations of deletions, insertions, internal deletes, internal inserts, and renames. Table 2 lists some of the different trees we generated.

Although the performance numbers were sensitive to the percentages of the different kinds of edits, the trends were very similar. Figure 14 shows a plot of the performance of the MDIR algorithm, with $R = 5K$, when varying the tree sizes. For comparison, the times for THP are shown in the corresponding THP series. Our empirical results clearly confirm our earlier theoretical complexity analysis, i.e., the

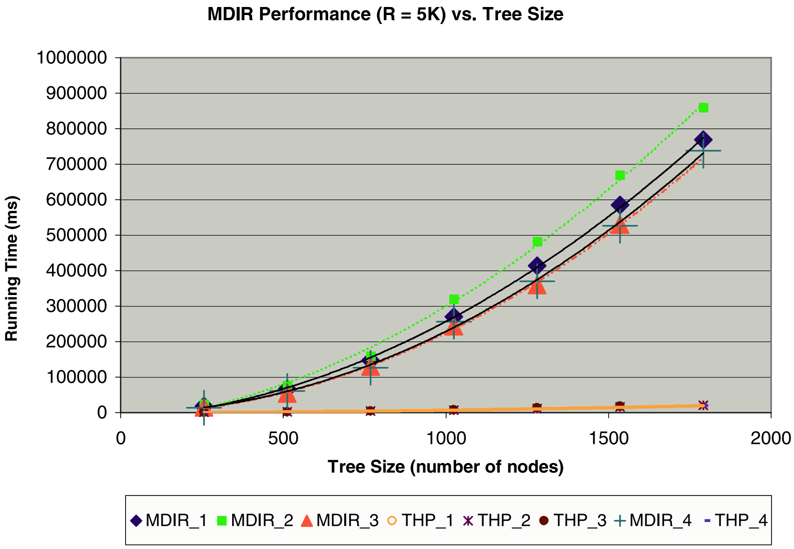


Fig. 14 The performance of MDIR ($R = 5K$) vs. the tree size

growth is quadratic. Although Figure 14 shows that MDIR performs much slower than THP, it does not give any indication on the accuracy of MDIR compared to THP, which we previously demonstrated.

We have avoided prematurely optimizing our current implementation to allow for easier debugging, but we think that we can improve the running time in several ways. Heuristics, such as simulated annealing or genetic algorithms, could significantly improve the SEARCH procedure by obtaining a better initial solution, and thus a better starting cost. This optimization could yield a significant performance gain, and allow the implementation to handle larger values of B, as well as trees of larger degree than is currently possible. Another optimization would be to reduce the number of nodes in the trees under comparison, e.g., by adding some nodes as attributes on their parents.

In summary, MDIR has a dramatically improved accuracy over THP and an acceptable non-interactive performance for many usage scenarios. Unlike exact graph matching algorithms, it can scale to thousands of nodes and can handle realistic architectural views, as the extended examples in Section 6 will demonstrate.

5 Architectural view synchronization

In this section, we use tree-to-tree correction algorithms to synchronize hierarchical graphs corresponding to C&C views.

5.1 General approach

We represent the structural information in a C&C view as a cross-linked tree structure that mirrors the hierarchical decomposition of a system. The tree also includes

Fig. 15 Figure 15(a) indicates a *match*; Figure 15(b) indicates a *rename*; Figure 15(c) indicates an *insertion*; and Figure 15(d) indicates a *deletion*

some redundant information to improve the accuracy of the structural comparison. For instance, the subtree of a node corresponding to a port includes additional nodes for all the port's involvements, i.e., all the components and their ports reachable from that port. Each node is decorated with properties, such as type information. The type information, if provided, populates a matrix of incompatible nodes that may not be matched. That matrix also includes optional user-specified constraints to force or prevent matches.

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 graph to appear in several subtrees. We consider one of these nodes the *defining occurrence*, and add a *cross-link* from each repeated node back to its defining occurrence. These redundant nodes, while they significantly increase the tree sizes, greatly improve the tree-to-tree correction accuracy. 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 corresponding to the system decomposition, i.e., components, ports and representations. During the second pass, we synchronize the edges in the architectural graph. The post-processing step is simple at that point, since it has the mapping between the nodes in the two graphs.

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. The final step is optional because the architect may decline the edit operations for various reasons, or may be interested only in a change impact analysis (Krikhaar et al. 1999). 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 produces semantic information that speeds up the comparison. This optional information can also reduce the amount of data entry for assigning types to the elements that the edit script will create.

In Step 3, matching instances proceeds as follows: a) build tree-structured data from the two C&C views to be compared; b) use tree-to-tree correction to identify matches and structural differences; and c) obtain an edit script to merge the two views.

The tool shows the structural differences by overlaying icons on the affected elements in each tree (See Figure 15). 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. The tool shows in bold a node if it detects differences in its subtree. The tool shows in italics ports that are inherited from the component type.

Various features can restrict the size of the trees and help reduce the comparison time:

- **Start at Component:** the user can select any component to be the tree root, and can reduce the tree sizes by selecting subtrees;
- **Restrict Tree Depth:** the user can exclude from comparison any nodes beyond a certain tree depth;
- **Elide Elements:** the user can exclude selected nodes and their entire subtrees from comparison. Elision is temporary and does not generate any edit actions.

The tool gives the user manual control using the following features:

- **Forced matches:** the user can manually force a match between two elements that may not match structurally;
- **Manual overrides:** the user can override any edit action suggested by the structural comparison.

Step 4 produces from the edit script a common supertree, that previews the merged view after the edit actions are applied. In this step, the user can assign types to elements to be created, change the types of existing elements, or override types that were automatically inferred based on the type matching in Step 2. The tool also checks the edit script for errors, such as illegal element names. The user can also rename any architectural element that the edit script will create. Finally, the user can cancel any unwanted edit actions.

5.2 Specialized tools

This approach supports building architectural view differencing and merging tools for a wide range of Architecture Description Languages (ADLs). To evaluate our approach, we represented the C&C views in the Acme general purpose ADL (Garlan et al. 2000).

We also developed a tool to extract as-built C&C views from ArchJava (Aldrich et al. 2002). Similar tools can extract as-built views from an implementation-constraining ADL with code generation capabilities, or an implementation-independent ADL with an implementation framework, such as C2 (Medvidovic and Taylor 2000).

We intended our synchronization tools to be lightweight enough that they can fit into a single dialog in an integrated development environment, such as Eclipse (Object Technology International 2003), and not require a specialized environment for architectural recovery (Telea et al. 2002). Both AcmeStudio a domain-neutral architecture modeling environment for Acme (Schmerl and Garlan 2004), and ArchJava's development environment, are Eclipse plugins, which reduced the tool integration effort.

One synchronization tool can make an as-designed architectural view, expressed in Acme, incrementally consistent with an as-built architectural view, extracted from an ArchJava implementation. We still need to change the ArchJava infrastructure to support making incremental changes to an existing ArchJava implementation based on changes to the as-designed view.

Another synchronization tool, based on the same approach, more generally takes any two C&C views represented in Acme. One view could correspond to a documented architecture. The second view could correspond to a C&C view recovered using any architectural recovery technique, e.g., (Schmerl et al. 2006). The second view could also be another C&C view, either retrieved from a configuration management system, or corresponding to a variant in a product line.

Synchronizing an as-designed C&C view with an as-built C&C view must often address expressiveness gaps between architectural information at different levels of abstraction. Although we use Acme and ArchJava to illustrate some of these differences that must be bridged, synchronizing any pair of as-designed and as-built C&C views may encounter similar challenges.

Structural differences. There will always be name differences of the same structural information between Acme and ArchJava. For instance, an ArchJava port can be named `in`, a reserved keyword in Acme. Even if code generation automatically produces a skeleton implementation from the architectural model, connector names and role names are lost, since ArchJava does not even name those elements. Finally, in Acme, port names are critical for typechecking. But in ArchJava, port names are unimportant and obey the standard programming language notions of binding and scope.

Hierarchy. Acme treats hierarchy as design-time composition, where a component at one level in the hierarchy is just a transparent view of a more detailed decomposition specified by the representation of that component. Multiple representations for a given component or connector could correspond to alternative ways of decomposing an element. On the other hand, ArchJava views hierarchy in terms of integration of existing components, along with glue code, into a higher level component. Due to the glue, a higher-level component is semantically more than the sum of its parts. These differing views of hierarchy create additional challenges for architectural synchronization. For example, if multiple representations are present at the design level, there must be a way to specify which of these representations was actually implemented.

Matching instances. Obtaining the tree-structured data from Acme simply converts the Acme architectural graph into the cross-linked tree structure discussed earlier. Acme does not have first-class constructs for required and provided methods. In keeping with Acme's model for extensible properties, the tool adds properties on a port to represent its provided and required methods, as well as other salient properties, e.g., the port's visibility.

To obtain the tree-structured data from an ArchJava implementation, the tool traverses the compilation units, ignores classes that are not component types, and fields that are not of component type. Different modeling choices are possible in this case. First, ArchJava does not name connectors or connector roles. The tool generates synthetic names from the components and ports that a connector connects to. Second, the ArchJava top-level component can have ports, whereas the top-level component in Acme, i.e., the Acme system, cannot. One option is to create a top-level component in Acme to correspond to ArchJava's top-level component. Another is to create a synthetic component to hold these ports. Third, ArchJava ports can be private, whereas all Acme ports are public. One option is to represent ArchJava private ports as Acme ports on an internal component instance; another is to simply ignore private ports.

Matching types. Assigning architectural styles and types to an Acme view enforces the architectural intent using constraints (Monroe 2001). For instance, a constraint on a component type may specify that all instances of that type must have exactly two ports. Similarly, setting architectural styles on the overall system — and on each sub-system representation if applicable, enforces any constraints associated with the style. In Acme, the Pipe-and-Filter style prohibits cycles, a constraint that a general purpose implementation language, such as ArchJava, does not directly enforce.

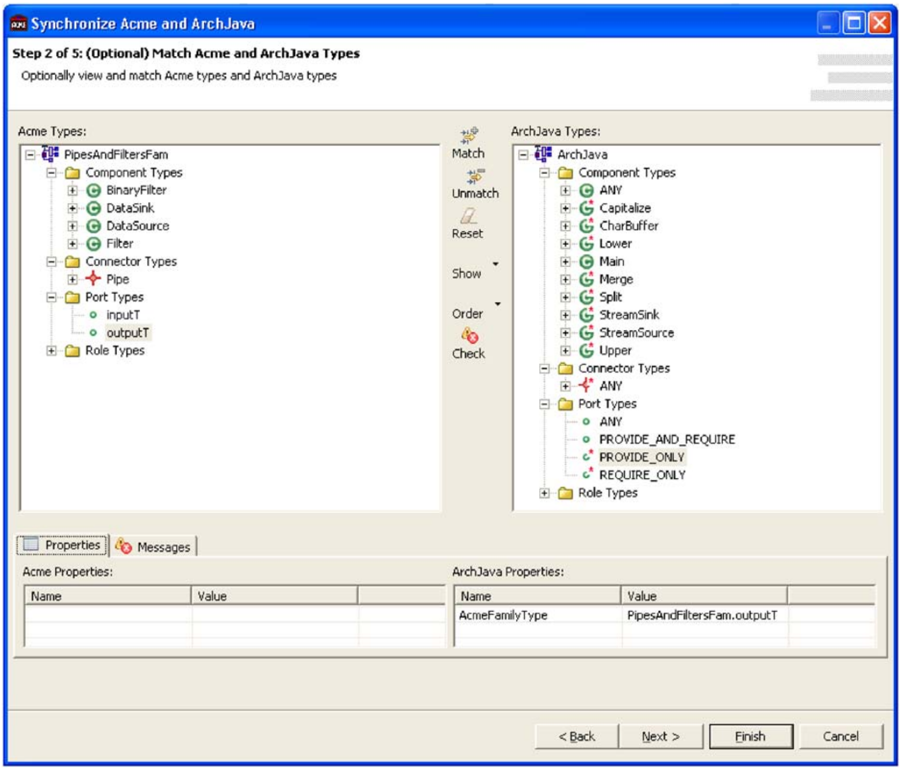
In many design languages, types are arbitrary logical predicates. An element is an instance of any type whose properties and rules it satisfies. And one type is a subtype of another, if the predicate of the first type implies the predicate of the second type. Such a type system is highly desirable at design time, because it allows designers to combine type specifications in flexible ways. Acme embodies this approach, but is hardly unique; for instance, PVS (Rushby et al. 1998) takes a similar approach. As an example of using a predicate-based type system, consider an architecture that is a hybrid of the Pipe-and-Filter and Shared-Data architectural styles. In this example, a `Filter` component type has at least one `input` and one `output` port, while a `Client` component in the Shared-Data style has at least one port to communicate with the repository. A component in this architecture might inherit both the `Filter` and the `Client` specifications, yielding a component that has at least three ports — two for communicating with other filters and one for communicating with the `Repository`.

However, implementation-level type systems, such as the ones provided by C2SADL (Medvidovic et al. 1996) or ArchJava, cannot express the example above. A specification that a component has a port implies a requirement that the environment will match that port up with some other component. Therefore, conventional type systems require a component type to list all of the ports it might possibly have — or at least all those ports that are expected to be connected at runtime. There is no way to express that a `Filter` component has "at least two ports" — instead, one must say that the `Filter` has "at most" or "exactly" two ports. Therefore, in the implementation, one cannot combine the `Filter` type with a `Repository` component type — which defines a third port that is prohibited by the filter specification.

So a design-level predicate-based type system is fundamentally incompatible with a type system for a programming language. As a result, the matching algorithm may not rely on exactly matching typing information as in UMLDiff (Xing and Stroulia 2005). In our approach, the user specifies arbitrary matches between the type hierarchies from Acme and ArchJava, flattened and shown side-by-side.

Consider synchronizing the Acme model of a simple system following the Pipe-and-Filter style with its ArchJava implementation. In Figure 16, the user matches the types as follows. The user selects the `Capitalize`, `CharBuffer`, `Lower`, `Merge`, `Split`, `Upper` component types in ArchJava and matches them with `Filter` Acme type. All the component instances of these ArchJava types will be assigned the `Filter` Acme type during synchronization. Using a limited form of wildcards, the user assigns the Acme type `Pipe` to the ArchJava connector type `ANY`. So any Acme connector created for an implicit ArchJava connector instance will have that type.

Since ArchJava ports are not typed, the user can individually assign to an ArchJava port a set of Acme port types. To reduce the manual work, the user use another



Synchronize Acme and ArchJava

Step 2 of 5: (Optional) Match Acme and ArchJava Types

Optionally view and match Acme types and ArchJava types

Acme Properties:

Name	Value	

ArchJava Properties:

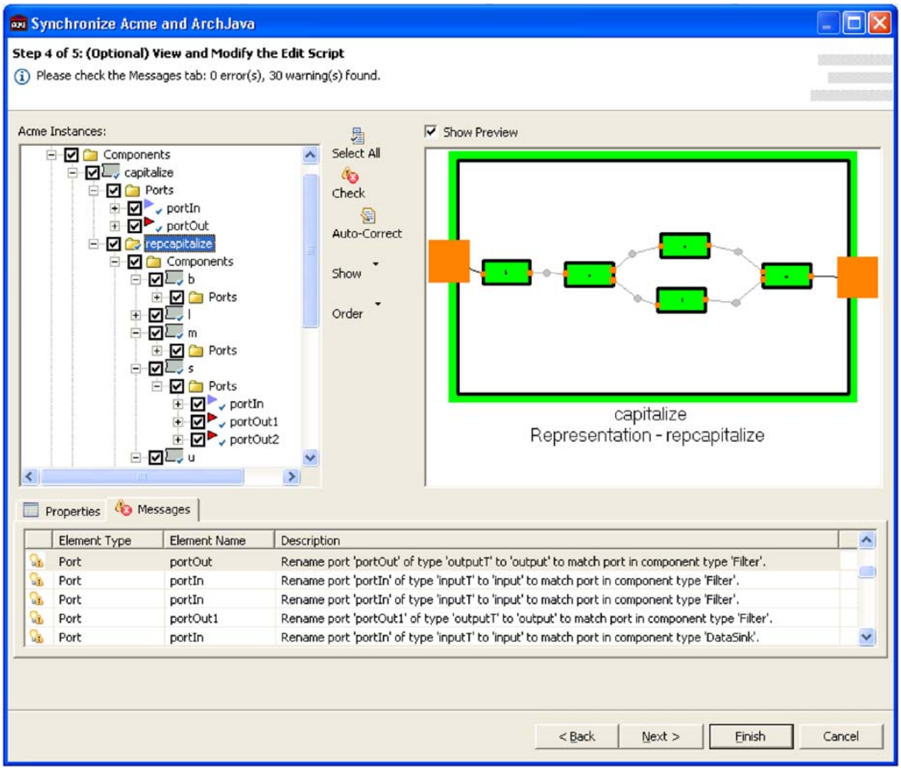
Name	Value	
AcmeFamilyType	PipesAndFiltersFam.outputT	

Fig. 16 Matching types between the as designed Acme model of a simple system following the Pipe-and-Filter style with its as-built ArchJava implementation

form of wildcards. He can assign an Acme type, e.g., `outputT`, to any ArchJava port that only provides methods. Similarly, he can assign the `inputT` Acme type to any ArchJava port that only requires methods. In addition, AcmeStudio defines connection patterns for most architectural styles. Based on these patterns, the tool can infer the Acme role types, once the user assigns types to components, ports and connectors. For instance, the tool infers the role type `sourceT`, based on the source component type `Filter`, source port type `inputT`, and connector type `Pipe`.

In this case, the synchronization produces the edit script in Figure 17. Since the user mapped the types, the edit script elements already have types. Each view element that already has a type is displayed using the same type- and style- dependent visualization that it would have in AcmeStudio. If the user does not specify architectural types and styles, the elements that the edit script will create will be untyped. Of course, the user can set the types on the newly inserted view elements at a later point in AcmeStudio. Although assigning types during synchronization seems to duplicate functionality, it may affect the edit script and the view merging as explained below.

For instance, when a component instance is assigned the `Filter` component type, it inherits any ports declared on that type, e.g., ports `input` and `output`, of types `inputT` and `outputT`. So the tool need not create additional ports of these



Synchronize Acme and ArchJava

Step 4 of 5: (Optional) View and Modify the Edit Script

Please check the Messages tab: 0 error(s), 30 warning(s) found.

Acme Instances:

☑ Show Preview

Element Type	Element Name	Description
Port	portOut	Rename port 'portOut' of type 'outputT' to 'output' to match port in component type 'Filter'.
Port	portIn	Rename port 'portIn' of type 'inputT' to 'input' to match port in component type 'Filter'.
Port	portIn	Rename port 'portIn' of type 'inputT' to 'input' to match port in component type 'Filter'.
Port	portOut1	Rename port 'portOut1' of type 'outputT' to 'output' to match port in component type 'Filter'.
Port	portIn	Rename port 'portIn' of type 'inputT' to 'input' to match port in component type 'DataSink'.

Fig. 17 Validating the edit script can involve renaming some ports to match the names declared in the Acme type

types on the component instance. Based on the user's selection in Figure 16, the tool matches the ArchJava port `portOut` — since it only provides methods, with the Acme type `outputT`. The tool suggests renaming the port `portOut` of type `outputT`, to match the `output` port on the `Filter` type.

The user can accept the corrective actions suggested by the tool using the Auto-Correct button in Figure 17. In that case, the tool automatically renames `portOut` port to `output`, and updates all the cross-references in the edit script. The user can also change the assigned or inferred types before pushing the changes to the Acme model.

6 Extended examples

In this section, we evaluate the tools for C&C view synchronization in several extended examples on real architectural views.

6.1 Extended example: Aphyds

In this example, we synchronize an as-designed C&C view with an as-built C&C view retrieved from an implementation. This example mainly highlights the ability of the underlying MDIR algorithm to detect inserts, deletes and renames.

The subject system is Aphyds, a pedagogical circuit layout application. To evaluate ArchJava's expressiveness to specify the architecture in code, Aphyds was re-engineered into an ArchJava implementation starting with 8,000 source lines of Java code — not counting the libraries used (Aldrich et al. 2002). We chose Aphyds since it had a documented as-designed architecture, and the ArchJava implementation enables extracting the as-built C&C view using a tool.

Figure 18 shows an informal drawing by the developer of the original Java program of the as-designed Aphyds architecture. In the following discussion, the architect performing the synchronization is a third party, with no prior experience with the original Java program, or with re-engineering the Java program into the ArchJava implementation.

As-designed architecture. The architect created an Acme model based on the informal architecture in Figure 19(a). He represented the `circuitModel` as a single component, and added all the computational components to a representation of `circuitModel` in Figure 19(b). In the original diagram (Figure 18), the thin arrows represent control flow, and the thick arrows represent data flow, but the architect did not make that distinction and showed all communication as Acme connectors.

Matching types. The architect chose an Acme Model-View-Controller style, `MVCFam`. Since the architect was interested in the control flow, he assigned the `provideT` Acme port type defined in `MVCFam` to any ArchJava port that only provides methods. Similarly, he assigned the `useT` Acme port type to any ArchJava port that only requires methods, and the `provreqT` Acme port type to any ArchJava port that both provides and requires methods. He also assigned the generic `TierNodeT` Acme type to all components and the `CallReturnT` Acme type to all the implicit ArchJava connectors (See Figure 20).

Matching instances. The architect used the synchronization tool to compare the two views. As he was the least sure about how he represented the `circuitModel` component in Acme, he decided to focus on that 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 21). The architect was particularly intrigued that the Acme representation for `circuitModel` had more connectors than the ArchJava implementation. In Figure 21, the tool only matched the `starConnector` which connects components `circuit`, `partitioner`, `floorPlanner`, `place`, `route` and `channel` in Figure 19. The architect investigated this further and confirmed that the Acme connectors corresponding to the thick data flow arrows in the informal diagram in Figure 18 are not in the implementation. There was no data flow in the implementation. Aphyds was written for academic study, not for industrial application, it is missing some of the data flows that would be present in a real application, i.e., the data flow is simulated rather than real, and thus it does not appear in the extracted architectural diagram. So, the architect accepted the edit actions to delete these extra connectors from the Acme model.

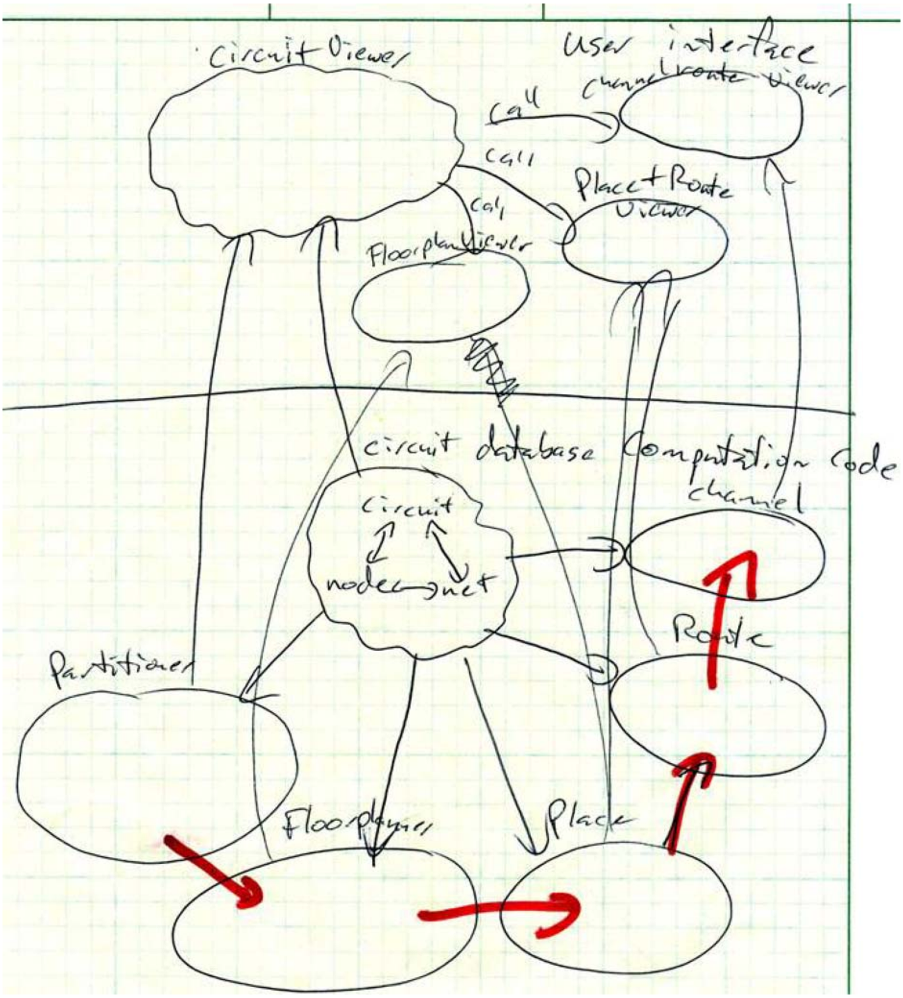
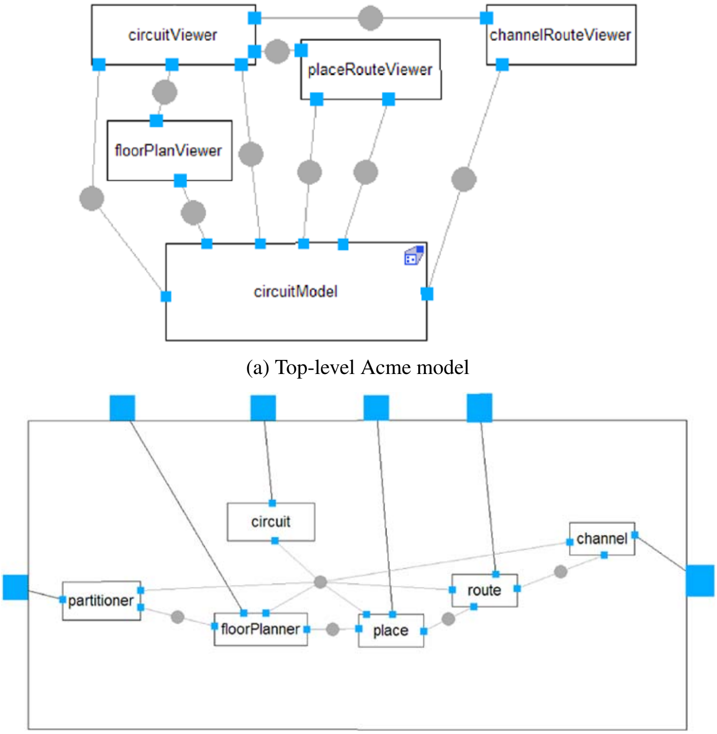


Fig. 18 Informal Aphyds as-designed architecture as drawn by the original developer

Merging instances. The architect next turned his attention to the additional top level component, shown as `privateAphyds` in Figure 21. Based on the synchronization options he selected, he determined that the tool created `privateAphyds` to represent a private `window` port in ArchJava and the corresponding glue. After looking at the control flow, the architect assigned that subsystem the Publish-Subscribe Acme style. He also renamed component `privateAphyds` to `window`, renamed the added connector to `windowBus`, and assigned `windowBus` the `EventBusT` connector type from the style. The architect also decided to use the same component names as the ArchJava implementation to avoid future confusion, so he accepted the renames in the edit script.

Discussion. Figure 22 shows the resulting C&C view after it has been manually laid out in AcmeStudio. Unlike the original architect's view, Figure 22 shows



(a) Top-level Acme model

(b) Acme representation of the `circuitModel` component

Fig. 19 As-designed Aphyds architecture represented in Acme

bi-directional communication taking place between components `placeRouteViewer` and `model`. The architect investigated that unexpected communication, and traced it to a callback. Aphyds is a multi-threaded application with long running operations moved onto worker threads. So 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 for evolving the system.

Performance evaluation. On an Intel Pentium 4 CPU 3 GHz with 1.5 GB of RAM, comparing an Acme tree of around 650 nodes with an ArchJava tree of around 1,150 nodes (Figure 21) with MDIR took under 2 minutes. In comparison, 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. For Aphyds, the edit script consisted of over 300 renames, over 600 inserts and over 100 deletes.

6.2 Extended example: Duke's Bank

In this example, we synchronize two C&C views, where one the as-built view is recovered by instrumenting the running system. This example mainly highlights the ability of the underlying MDIR algorithm to detect moves in addition to renames.

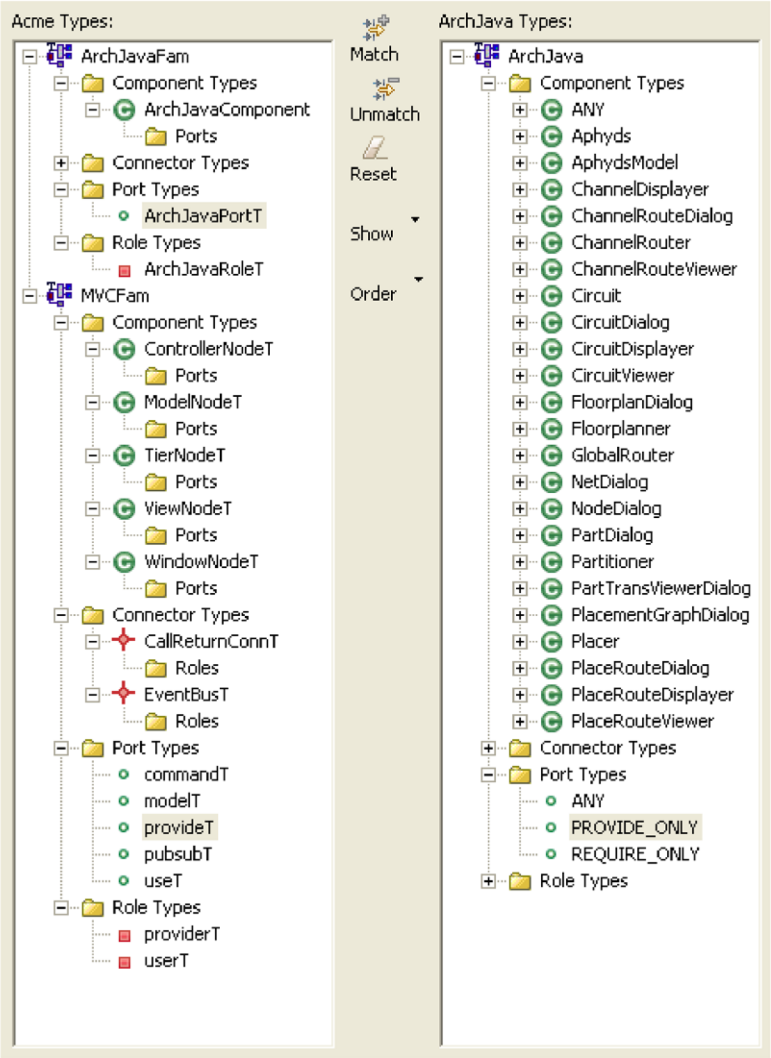
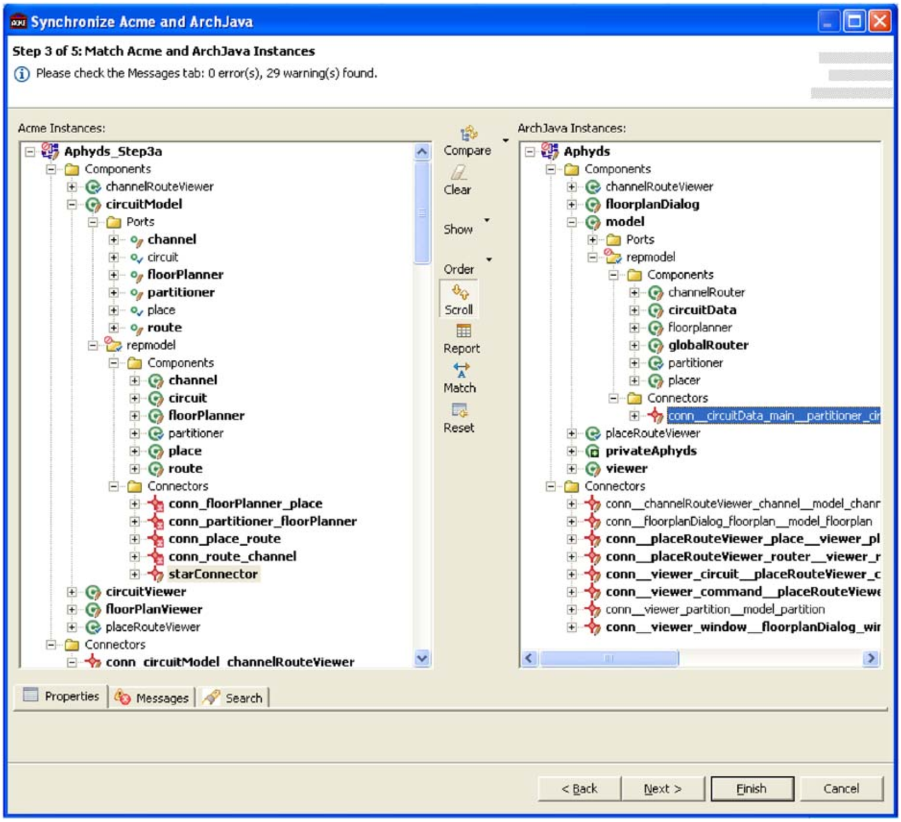


Fig. 20 Matching types between Acme (left) and ArchJava (right)

The subject system is Duke's Bank, a simple Enterprise JavaBeans (EJB) banking application. The architect wanted to compare the documented architecture with the as-built architecture, recovered using an architectural recovery technique other than ArchJava. Duke's Bank is also representative of industrial code that uses middleware, and furthermore, has a documented as-designed architecture.

As-designed architecture. The architect converted an informal diagram (See Figure 23) into an Acme model (See Figure 24).

As-built architecture. The as-built architecture was recovered by a dynamic architecture extraction tool, DiscoTect (Schmerl et al. 2006). DiscoTect currently generates one component instance for each session and entity bean instance created at


Synchronize Acme and ArchJava

Step 3 of 5: Match Acme and ArchJava Instances

Please check the Messages tab: 0 error(s), 29 warning(s) found.

Fig. 21 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

runtime. So the architect post-processed it, and unified such multiple instances into one instance. The goal was to make the recovered C&C view in Figure 25 comparable to a typical C&C view, where each component instance represents any number of runtime components.

Matching types. In this case, the as-built view and the as-designed view use the same architectural style and types, so the architect skipped the optional step of matching types.

Matching instances. The differencing tool correctly detected the moves corresponding to replacing the `container` component in one view with its representation in the other view (See Figure 26). Because a tool generated the names in the recovered view, e.g., `AccountBean_e55d75`, there was a large number of renames in this case. The synchronization tool matched all the elements between the two views, despite the large number of renames.

Discussion. The tool also identified on `Account_Controller_Bean` a port that was attached to a `DbWriter` connector. Figure 24 does not show a connection between the `Account_Controller_Bean` and the `DB` components. In fact, the

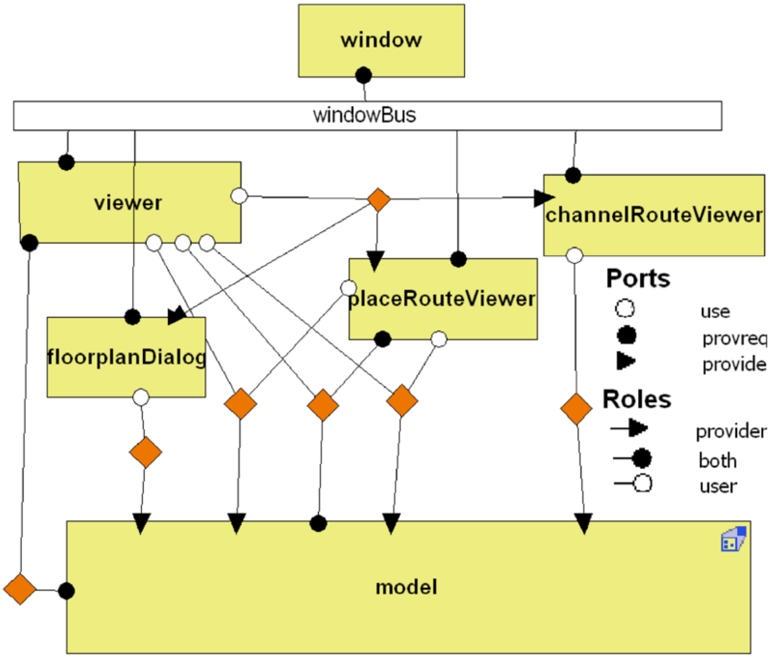


Fig. 22 As-built Aphyds architecture with Acme styles and types

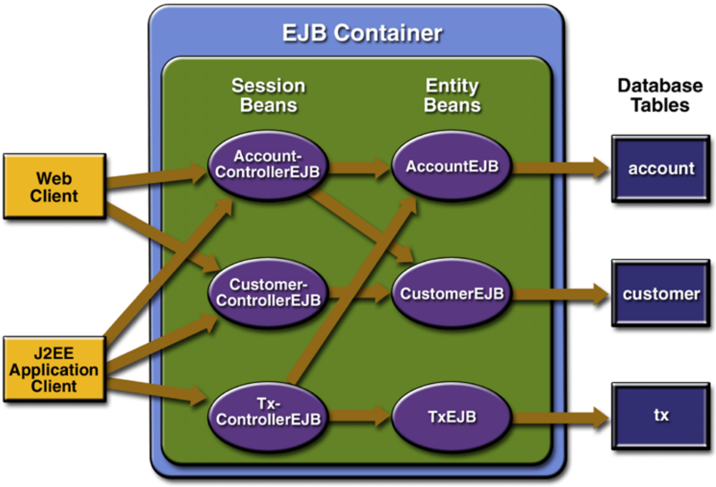


Fig. 23 Informal as-designed architecture for the Duke's Bank application (Sun Microsystems 2006)

EJB specification recommends that all database access goes through entity beans. In this case, the tool found an architectural violation in Sun's own example!

Performance evaluation. On an Intel Pentium 4 CPU 3 GHz with 1.5 GB of RAM, MDIR took around 30 seconds to compare the two Acme trees, one with

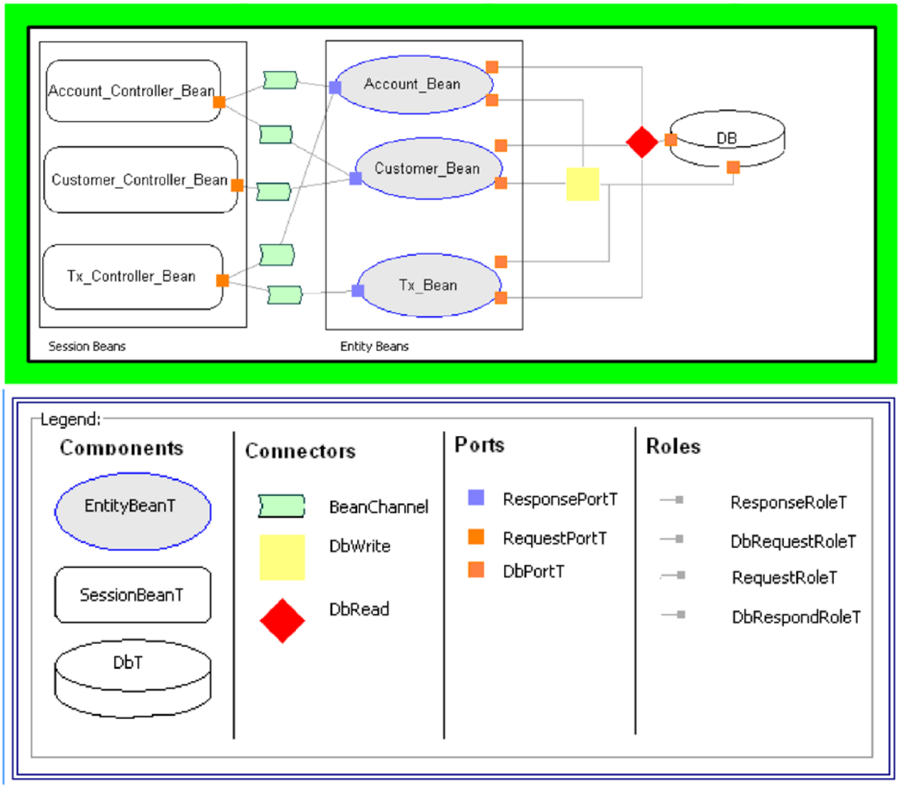


Fig. 24 Duke's Bank documented architecture in Acme; the components were added inside the Acme representation of an EJB container (shown as a thick border). Session and Entity Beans are grouped

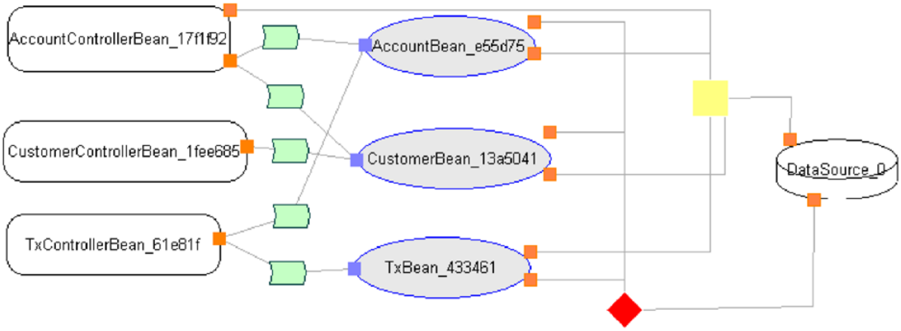
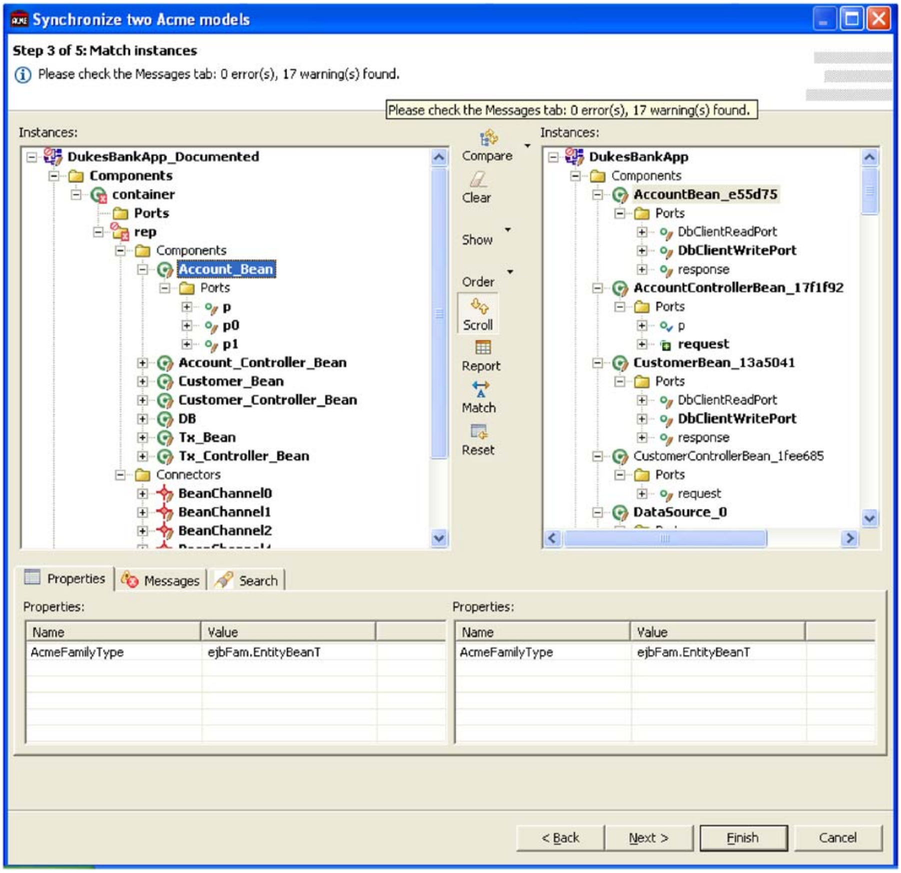


Fig. 25 Duke's Bank recovered architecture in Acme

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 in this case, THP did not correctly identify the moved view elements.



Step 3 of 5: Match instances

Please check the Messages tab: 0 error(s), 17 warning(s) found.

Instances:

- DukesBankApp_Documented
  - Components
    - container
      - Ports
      - rep
        - Components
          - Account_Bean
            - Ports
              - p
              - p0
              - p1
          - Account_Controller_Bean
          - Customer_Bean
          - Customer_Controller_Bean
          - DB
          - Tx_Bean
          - Tx_Controller_Bean
        - Connectors
          - BeanChannel0
          - BeanChannel1
          - BeanChannel2

Instances:

- DukesBankApp
  - Components
    - AccountBean_e55d75
      - Ports
        - DbClientReadPort
        - DbClientWritePort
        - response
    - AccountControllerBean_17f1f92
      - Ports
        - p
        - request
    - CustomerBean_13a5041
      - Ports
        - DbClientReadPort
        - DbClientWritePort
        - response
    - CustomerControllerBean_1fee685
      - Ports
        - request
    - DataSource_0

Properties:

Name	Value	
AcmeFamilyType	ejbFam.EntityBeanT	

Properties:

Name	Value	
AcmeFamilyType	ejbFam.EntityBeanT	

Fig. 26 Comparison of the Duke's Bank documented and recovered architectures

6.3 Extended example: HillClimber

In this example, we evaluate the tool to synchronize two C&C views again, but this time, we allow the user to force matches. All the examples actually use the feature to prevent matches, to avoid matching elements of incompatible types.

The subject system, HillClimber, is part of a collection of Java applications to graphically demonstrate artificial intelligence algorithms, built on the `CIspace` framework. In particular, HillClimber, demonstrates stochastic local search algorithms for constraint satisfaction problems. HillClimber was re-engineered from about 15,000 lines of Java (Abi-Antoun et al. 2007). We chose HillClimber because it uses a framework. A product line architecture often uses a framework as its platform, and one often needs to compare variants in a product line (Chen et al. 2003). The implementation technology, ArchJava, also made it easy to extract the as-built C&C view.

As-designed architecture. The applications that use the `CIspace` framework follow a simple high-level design. An application `window` uses a `canvas` to display

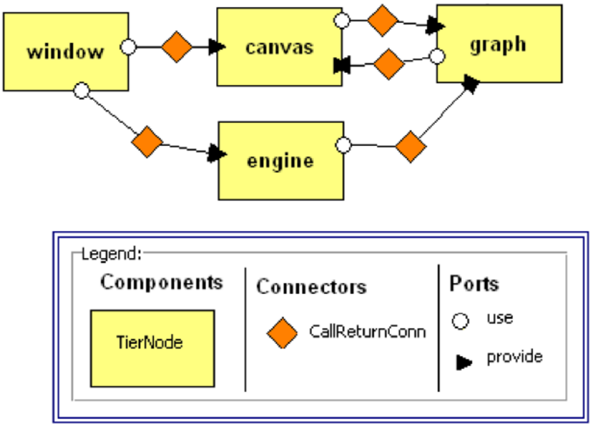


Fig. 27 Base design for a CIspace framework application

nodes and edges (not shown) of a graph in order to demonstrate the algorithms provided by the engine (See Figure 27).

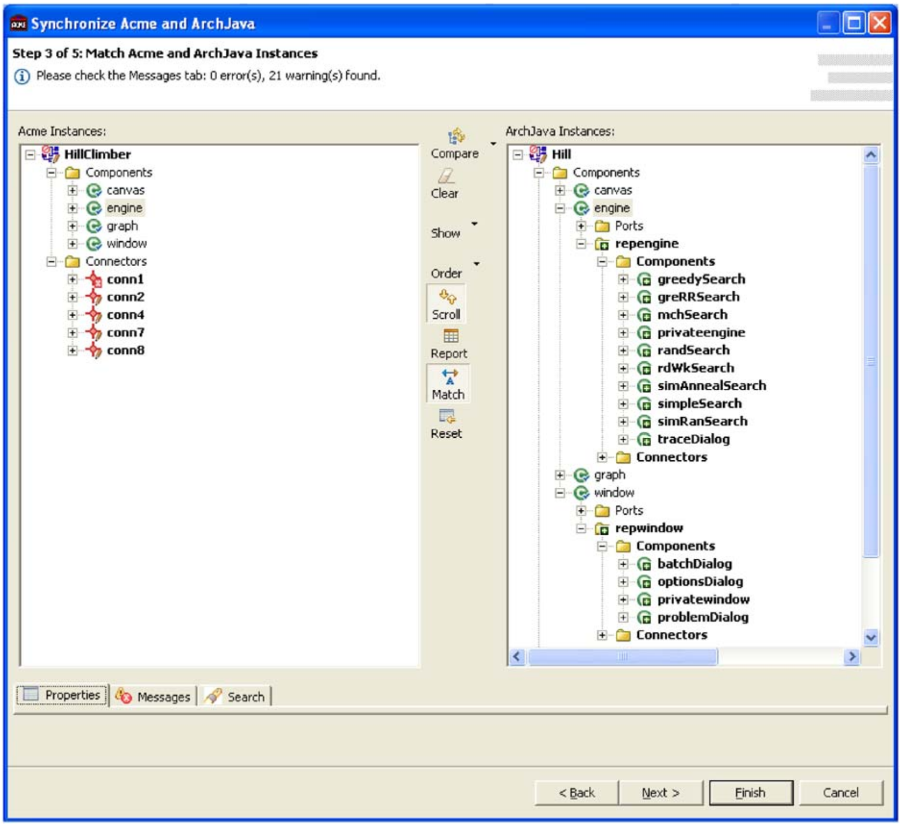
As-built architecture. We first ran the tool to synchronize the as-designed C&C view in Figure 27 with a C&C view retrieved from the HillClimber implementation. In this case, the top-level structure of the as-designed view was not sufficiently detailed, i.e., the various nodes have roughly the same number of ports. In such cases, structural comparison alone can produce inaccurate results. In this case, the MDIR algorithm incorrectly matched the top-level element graph in one view to window in the other view.

So the user manually forced the matches between the top-level nodes in the two views, and re-ran the comparison. This time, the MDIR algorithm took into account these manual overrides when matching the instances. Having correctly matched the top-level elements, the comparison highlighted additional differences between the two views. For instance, Figure 28 shows many missing sub-architectures. But the user decided to merge only the changes for the top-level elements and obtained the as-built architecture in Figure 29.

Discussion. In a product line architecture, each instantiation of a framework often introduces additional runtime dependencies. Indeed, HillClimber added several connections to the documented architecture, and these connections seem mostly justified. For instance, the connection between engine and canvas is needed since one of the sub-components of engine required access to functionality from the canvas.

7 Related work

Landmark-based algorithms. We group several algorithms that have been proposed for differencing hierarchical information under the category of "landmark-based algorithms": they have been proposed in the context of program differencing, e.g., JDiff (Apiwattanapong et al. 2004), Dex (Raghavan et al. 2004), and design differencing, e.g., UMLDiff (Xing and Stroulia 2005). 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 them to recognize the unchanged nodes first and use



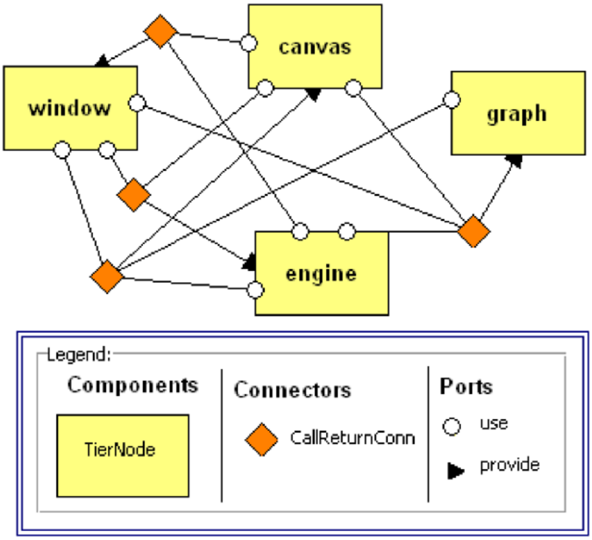
Step 3 of 5: Match Acme and ArchJava Instances

Please check the Messages tab: 0 error(s), 21 warning(s) found.

Fig. 28 Manual overrides improve matching the instances. The user forced a match between the `engine` nodes in the two trees by selecting them both and clicking on the 'Match' button before running the differencing algorithm

them as "landmarks" to efficiently identify the other changes. 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 was tested on instances where more than 80% of the nodes matched exactly (Chawathe and Garcia-Molina 1997). 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 such as 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. 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 the Aphyds and Duke's Bank examples, our inputs had more than half of their nodes renamed. Finally, none of these algorithms offer the ability to manually force or prevent matches. It may be possible to easily

Fig. 29 The as-built HillClimber architecture



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 and can generalize to more than two trees, something not easily done with tree edit distance algorithms (Jiang et al. 1994). 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.

Graph matching approaches. A exhaustive graph matching algorithms, based on variants of the A* algorithm (Messmer 1996), do not scale beyond a few dozen nodes (Hlaoui and Wang 2002). In the context of architectural views, Sartipi proposed an approach for architectural recovery using a variant of the A* graph matching algorithm, but with an optimization that may cause it to miss the optimal solution in some cases (Sartipi and Kontogiannis 2003).

More scalable, heuristic-based approaches, such as spectral methods, perform poorly when the graphs are not nearly isomorphic. Furthermore, these algorithms occasionally miss the optimal solution (Conte et al. 2004). Others, such as the Similarity Flooding Algorithm (SFA), have an accuracy of around 50% (Melnik et al. 2002). 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” (Melnik et al. 2002).

Mandelin et al. proposed probabilistic matching based on label, region, type or position information (Mandelin et al. 2006), but the approach requires training the *evidencers*. Mandelin et al. also mention that a simple greedy search algorithm does not work in many cases.

Model transformation. Graph transformation approaches, surveyed by Mens and van Gorp (Mens and Van Gorp 2005), tackle the same problem, but use a different set of assumptions. First, in many graph grammars, productions do not delete vertices and edges, which effectively prohibits insertions and deletions, one of our requirements. Second, graph transformation approaches do not attempt to find the optimal transformation that would preserve properties of view elements. Finally, these approaches do not yet offer easy to use tools such as the ones illustrated in Section 6.

Conformance checking. The technique we followed in the extended examples is similar in spirit to the Reflexion Models technique (Murphy et al. 2001). Using Reflexion Models to check the conformance of the as-designed and the as-built view is limited to checking the correspondences between the edges. In particular, the comparison assumes that the two views have the same number of nodes and that the nodes are identically named. Furthermore, in Reflexion Models, views are non-hierarchical, which requires generating different views for each level of the hierarchy. Checking the conformance of two hierarchical views using our approach can be considered a generalization of the computation of the Reflexion Model.

Of course, there are many other less automated approaches. For instance, the FOCUS approach checks the conformance of an implementation with respect to an architectural style, but manually relates the as-designed and the as-built views (Medvidovic and Jakobac 2006).

Consistency management. There is significant work in the area of viewpoints, view merging and inconsistency management, e.g. (Easterbrook and Nuseibeh 1996; Egyed 2006). A viewpoint captures data from disparate sources into independent but interrelated units. In view merging, there is also a notion of knowledge order or degree, i.e., a match can be disputed. When synchronizing between an as-built and an as-designed architecture, one may want to model incompleteness and inconsistency as a first class notion. In our approach, we model both views using the same viewtype, arbitrarily bridging the inevitable expressiveness gaps in the process. We also assume that one of the two views is authoritative. Implicitly, when the user decides to commit some edit actions but not others, they are allowing some acceptable differences to remain. In future work, it may be interesting to model this more precisely using ideas from inconsistency management.

8 Conclusions

In this paper, we presented a novel algorithm for differencing and merging tree-structured data that improves an existing algorithm to detect moves, and support forcing and preventing matches. We used the tree-to-tree correction algorithm to compare and merge hierarchical architectural Component-and-Connector (C&C) views. We then presented tools that incorporate the algorithm, and showed how our relaxed assumptions match more closely the problem domain of differencing and merging architectural views. Finally, we illustrated the tools in extended examples and showed how the approach can find interesting differences in real architectural views.

Acknowledgements 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.

References

Abi-Antoun, M., Aldrich, J., Garlan, D., Schmerl, B., Nahas, N., Tseng, T.: Improving system dependability by enforcing architectural intent. In: Proceedings of the Workshop on Architecting Dependable Systems, pp. 1–7 (2005)

Abi-Antoun, M., Aldrich, J., Coelho, W.: A case study in re-engineering to enforce architectural control flow and data sharing. J. Syst. Softw. **80**(2), 240–264 (2007)

Alanen, M., Porres, I.: Difference and union of models. In: Proceedings of 6th International Conference on the Unified Modeling Language, Modeling Languages and Applications, pp. 2–17 (2003)

Aldrich, J., Chambers, C., Notkin, D.: ArchJava: connecting software architecture to implementation. In: Proceedings of the 24th International Conference on Software Engineering, pp. 187–197 (2002)

Ammann, M.M., Cameron, R.D.: Inter-module renaming and reorganizing: examples of program manipulation-in-the-large. In: Proceedings of the International Conference on Software Maintenance, pp. 354–361 (1994)

Apiwattanapong, T., Orso, A., Harrold, M.J.: A differencing algorithm for object-oriented programs. In: Proceedings of the 19th IEEE International Conference on Automated Software Engineering, pp. 2–13 (2004)

Chawathe, S.S., Garcia-Molina, H.: Meaningful change detection in structured data. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 26–37 (1997)

Chen, P.H., Critchlow, M., Garg, A., van der Westhuizen, C., van der Hoek, A.: Differencing and merging within an evolving product line architecture. In: Proceedings of the 5th International Workshop on Software Product-Family Engineering, pp. 269–281 (2003)

Clements, P., Bachman, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architecture: View and Beyond. Addison-Wesley, Cambridge (2003)

Conte, D., Foggia, P., Sansone, C., Vento, M.: Thirty years of graph matching in pattern recognition. Int. J. Pattern Recognit. Artif. Intell. **18**(3), 265–298 (2004)

Dashofy, E.M., van der Hoek, A., Taylor, R.N.: Towards architecture-based self-healing systems. In: Proceedings of the First Workshop on Self-Healing Systems, pp. 21–26 (2002)

Dickinson, P.J., Bunke, H., Dadej, A., Kraetzl, M.: Matching graphs with unique node labels. Pattern Anal. Appl. **7**(3), 243–254 (2004)

Easterbrook, S., Nuseibeh B.: Using ViewPoints for inconsistency management. Softw. Eng. J. **11**(1), 31–43 (1996)

Egyed, A.: Instant consistency checking for the UML. In: Proceeding of the 28th International Conference on Software Engineering, pp. 381–390 (2006)

Eixelsberger, W., Ogris, M., Gall, H., Bellay, B.: Software architecture recovery of a program family. In: Proceedings of the 20th International Conference on Software Engineering, pp. 508–511 (1998)

Erdogmus, H.: Representing architectural evolution. In: Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research, pp. 159–177 (1998)

Garlan, D., Monroe, R.T., Wile, D.: Acme: architectural description of component-based systems. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of Component-Based Systems, pp. 47–68. Cambridge University Press, Cambridge (2000)

Hlaoui, A., Wang, S.: A new algorithm for graph matching with application to content-based image retrieval. In: Proceedings of the Joint IAPR International Workshop on Structural, Syntactic, and Statistical Pattern Recognition, pp. 291–300 (2002)

Jiang, T., Wang, L., Zhang, K.: Alignment of trees—an alternative to tree edit. In: Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching, pp. 75–86 (1994)

Jimenez, A.M.: Change propagation in the MDA: a model merging approach. Master's thesis, University of Queesland (2005)

Krikhaar, R., Postma, A., Sellink, A., Stroucken, M., Verhoef, C.: A two-phase process for software architecture improvement. In: Proceedings of the IEEE International Conference on Software Maintenance, pp. 371–380 (1999)

Mandelin, D., Kimelman, D., Yellin, D.: A Bayesian approach to diagram matching with application to architectural models. In: Proceedings of the 28th International Conference on Software Engineering, pp. 222–231 (2006)

Medvidovic, N., Jakobac, V.: Using software evolution to focus architectural recovery. Autom. Softw. Eng. **13**(2), 225–256 (2006)

Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**(1), 70–93 (2000)

Medvidovic, N., Oreizy, P., Robbins, J.E., Taylor, R.N.: Using object-oriented typing to support architectural design in the C2 style. In: Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 24–32 (1996)

Mehra, A., Grundy, J., Hosking, J.: A generic approach to supporting diagram differencing and merging for collaborative design. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 204–213 (2005)

Melnik, S., Garcia-Molina, H., Rahm, E.: Similarity flooding: a versatile graph matching algorithm and its application to schema matching. In: Proceedings of the 18th International Conference on Data Engineering, pp. 117–128 (2002)

Mens, T., Van Gorp, P.: A taxonomy of model transformation. In: Proceedings of the International Workshop on Graph and Model Transformation (2005)

Messmer, B.: Efficient graph matching algorithms for preprocessed model graphs. PhD thesis, University of Bern (1996)

Monroe, R.: Capturing software architecture design expertise with Armani. Technical Report CMU-CS-98-163R, Carnegie Mellon University School of Computer Science (2001)

Muccini, H., Dias, M.S., Richardson, D.J.: Towards software architecture-based regression testing. In: Proceedings of the Workshop on Architecting Dependable Systems, pp. 1–7 (2005)

Murphy, G.C., Notkin, D., Sullivan, K.J.: Software reflexion models: bridging the gap between design and implementation. IEEE Trans. Softw. Eng. **27**(4), 364–380 (2001)

Object Technology International, Inc.: Eclipse platform technical overview. http://www.eclipse.org/whitepapers/eclipse-overview.pdf (2003)

Ohst, D., Welle, M., Kelter, U.: Differences between versions of UML diagrams. In: Proceedings of the 9th European Software Engineering Conference/11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 227–236 (2003)

Raghavan, S., Rohana, R., Leon, D., Podgurski, A., Augustine, V.: Dex: a semantic-graph differencing tool for studying changes in large code bases. In: Proceedings of the 20th IEEE International Conference on Software Maintenance, pp. 188–197 (2004)

Roshandel, R., van der Hoek, A., Mikic-Rakic, M., Medvidovic, N.: Mae—a system model and environment for managing architectural evolution. ACM Trans. Softw. Eng. Methodol. **13**(2), 240–276 (2004)

Rushby, J., Owre, S., Shankar, N.: Subtypes for specifications: predicate subtyping in PVS. IEEE Trans. Softw. Eng. **24**(9) (1998)

Sartipi, K., Kontogiannis, K.: On modeling software architecture recovery as graph matching. In: Proceedings of the 19th IEEE International Conference on Software Maintenance, pp. 224–234 (2003)

Schmerl, B., Garlan, D.: AcmeStudio: supporting style-centered architecture development. In: Proceedings of the 26th International Conference on Software Engineering, pp. 704–705 (2004)

Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., Yan, H.: Discovering architectures from running systems. IEEE Trans. Softw. Eng. **32**(7), 454–466 (2006)

Shasha, D., Zhang, K.: Approximate tree pattern matching. In: Apostolico, A., Galil, E.Z. (eds.) Pattern Matching Algorithms. Oxford University Press, Oxford (1997)

Shaw, M., Garlan, D.: Software Architectures: Perspectives on an Emerging Discipline. Prentice Hall, New York (1996)

Spitznagel, B., Garlan, D.: Architecture-based performance analysis. In: Proceedings of the Conference on Software Engineering and Knowledge Engineering (1998)

Sun Microsystems: J2EE tutorials. Duke's bank. http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Ebank2.html (2006)

Telea, A., Maccari, A., Riva, C.: An open visualization toolkit for reverse architecting. In: Proceedings of the 10th International Workshop on Program Comprehension, pp. 3–10 (2002)

Torsello, A., Hidovic-Rowe, D., Pelillo, M.: Polynomial-time metrics for attributed trees. IEEE Trans. Pattern Anal. Mach. Intell. **27**(7), 1087–1099 (2005)

van der Westhuizen, C., van der Hoek, A.: Understanding and propagating architectural changes. In: Proceedings of the Working IFIP Conference on Software Architecture, pp. 95–109 (2002)

Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. J. ACM **21**(1), 168–173 (1974)

Wang, Y., DeWitt, D.J., Cai, J.-Y.: X-Diff: an effective change detection algorithm for XML documents. In: Proceedings of the 19th International Conference on Data Engineering, pp. 519–530 (2003)

Xing, Z., Stroulia, E.: UMLDiff: an algorithm for object-oriented design differencing. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 54–65 (2005)

Zhang, K., Jiang, T.: Some MAX SNP-hard results concerning unordered labeled trees. Inf. Process. Lett. **49**(5), 249–254 (1994)