PAPER

Development of Program Difference Tool Based on Tree Mapping

Lin LIAN†, Minoru AIZAWA††, *Nonmembers*, Katsuro INOUE†, *and* Koji TORII†††, *Members*

SUMMARY In the program development process, it is often necessary for programmers to know the differences between two programs, or two different versions of a program. Since programs have structures such as iteration statement and selection statement, applying text-based tools such as UNIX `diff` to identify the differences may produce unsatisfactory results. In this paper, we exploit a tree as the internal representation of a program, obtain the mapping between two trees and display the program differences visually based on the mapping and pretty-printing technique so that the structural differences can be identified immediately.

key words: *program differences, tree mapping, pretty-printing, software tool*

1. Introduction

Program development is a complicated process rather than a one-time activity. Both program developers and maintainers frequently face the need to identify the differences between two programs, or two different versions of a program. For developers, it is often necessary to know what parts of a program have been changed for a fixed period; and for maintainers, it is essential to identify how related versions of a program differ. This kind of task is especially difficult when the programs are old, or are written by other programmers. A tool that accurately identifies differences between program versions, therefore, helps maintainers understand the programs and makes the maintenance task easy. For example, if the current version of a program is resulted from a bug fixing of an earlier version, we might only retest the changed components when the remainder of the current version is equivalent to that of the earlier one. Although the concept of 'difference' may change depending on the user's purpose, a tool that can find the structural differences is useful since a program is not simply a collection of text lines.

Existing tools such as the UNIX `diff` [1] are designed to compare text files rather than programs. These text-based comparison tools often produce unsatisfactory comparisons for programs, since they treat one text line as an element of the alphabet of the two programs to compare and output the lines that do not match by the Longest Common Sequence (LCS) [2]–[4] as the differences. Besides, the exact differences are hard to locate by looking at the resulting output. In addition, they may produce irrelevant differences, e.g., they may consider two statements to be different due to the placement of lines breaks, which are usually considered to be identical by programmers. Since programs have structures rather than a sequentially collected lines, these irrelevant details can be filtered out by taking the structures into account.

We develop a tool that identifies differences between programs by exploiting hierarchical structures [5]. Programs to be compared are first transformed into a tree by a parser. The tree, however, differs from the parse tree of compilers. The basic unit of the compilers is individual tokens while our unit of comparison is a statement or a declaration. That is, a node of a tree represents a statement or a declaration and the node's label is assigned to be the content of the statement or declaration. An algorithm is then used to find out a set of pairs of same-label nodes, one from each tree. The results of program comparison are finally printed out by a pretty-printer that prints the two programs line-by-line synchronously and the different parts are highlighted or colored. The readability of the resulting comparisons is therefore lifted.

In the next section, we show examples of program differences by the tool and by `diff`. In Sect. 3 we discuss the internal tree representation of a program and the difference between trees with respect to mapping. In Sect. 4 we present the algorithm that generates the mapping between trees. In Sect. 5 we describe the representation technique of the resulting program differences. We conclude and illustrate future work in Sect. 6.

2. An Example of Program Differences

Consider a simple example of two program versions shown in Fig. 1. There are minor distinction in them, such that the second line in *version1*,

```
char s[ ]; int maxs;
```

is divided into two lines, and the bool expression of `while` statement in function *getstr* is modified along

Manuscript received August 10, 1994.

†The authors are with the Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University, Toyonaka-shi, 560 Japan.

††The author is with the Software Laboratory, TOSHIBA CORPORATION, Kawasaki-shi, 210 Japan.

†††The author is with the Graduate School of Information Science, Nara Advanced Institute of Science and Technology, Ikoma-shi, 630–01 Japan.

```
int getstr(s,maxs)
    char s[ ]; int maxs;
{
    int i,c;
    i=0;
    c=getchar( );
    while (i<maxs)
      {
          s[i]=c;
          i=i+1;
          c=getchar( );
          s[i]='\0';
          return(i);
      }
}

int do_reverse(s1,s2)
    char s1[ ], s2[ ];
{
    int i,j;
    i=0;
    for (j=strlen(s1);j>=0;j--)
      {
          s2[i]=s1[j];
          i=i+1;
      }
    s2[i]='\0';
}

main ( )
{
    char prase[71]; char rex_phrase[71];
    printf("Type a phrase: \n");
    getstr(phrase,70);
    do_reverse(phrase,rev_phrase);
    printf("reverse = %s", rev_phrase);
}
```

Version1

```
int getstr(s,maxs)
    char s[ ];
    int maxs;
{
    int i,c;
    i=0;
    c=getchar( );
    while (i<maxs && c!='\n' && c!=EOF)
      {
          s[i]=c;
          i=i+1;
          c=getchar( );
      }
          s[i]='\0';
          return(i);
}
int do_reverse(s1,s2)
    char s1[ ], s2[ ];
{
    int i,j;
    i=0;
    j=strlen(s1)-1;
    while (j>=0)
      {
          s2[i]=s1[j];
          i=i+1;
          j=j-1;
      }
    s2[i]='\0';
}
main ( )
{
    char prase[81];
    char rex_phrase[81];
    printf("Type a phrase: \n");
    getstr(phrase,80);
    do_reverse(phrase,rev_phrase);
    if (strcmp(phrase, rev_phrase)==0)
      printf(" ** palindrome ** ");
    else printf("reverse = %s", rev_phrase);
}
```

Version2

Fig. 1 Two program versions to be compared.

with moving a few statements out of the loop in *Version2*. There are also some changes in functions *do_reverse* and *main*.

The result of applying `diff` utility to the two program versions is shown in Fig. 2, where the structural change is not easily understood. For example, `diff` deals with

```
char s[ ]; int maxs;
```

and

```
char s[ ];
int maxs;
```

as different. Also, `diff` can not point straightforwardly that the changes of statements `s[i]='\0';` and `return(i);` , which are in the inner body of the `while` loop in *Version1* and out of the loop in *Version2*. `Diff` only shows the change of }'s position in two programs and we hardly know the practical changes immediately by looking at what it shows.

Since we wanted to know the structural change of two versions in intuitive manner, we have constructed a tool for it. Figure 3 shows the execution result of our tool. The two versions are located side-by-side on the screen based on the comparison result. The underlined statements in *Version1* are to be deleted from *Version1*, and those in *Version2* are to be inserted into the remaining of deletion to make up *Version2*. (These deleted and inserted lines are represented by distinct colors on the color display.) Also, unchanged statements

```
2c2,3
<     char s[ ]; int maxs;
---
>     char s[ ];
>     int maxs;
7c8
<     while (i<maxs)
>     while (i<maxs && c!='\n' && c!=EOF)
12c13
<
---
>       }
16d16
< }
23c23,24
<     for (j=strlen(s1);j>=0;j--)
---
>     j=strlen(s1)-1;
>     while (j>=0)
26a28
>         j=j-1;
33c35,36
<     char prase[71]; char rex_phrase[71];
---
>     char prase[81];
>     char rex_phrase[81];
36c39
<     getstr(phrase,70);
---
>     getstr(phrase,80);
39c42,44
<     printf("reverse = %s", rev_phrase);
---
>     if (strcmp(phrase, rev_phrase)==0)
>       printf(" ** plindrome ** ");
>     else printf("reverse = %s", rev_phrase);
```

Fig. 2 Result of executing `diff`.

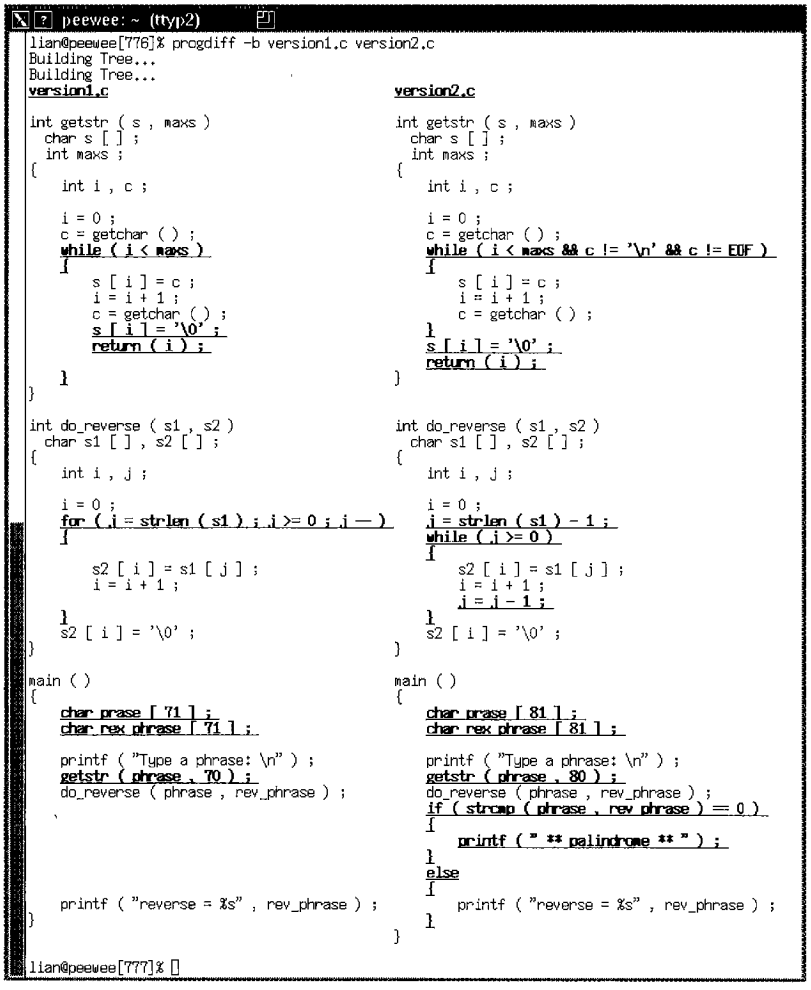


Fig. 3 The result produced by the tool.

are located side-by-side and pretty-printed properly. We would understand fairly easily the structural change of two versions.

The tool we have developed is based on algorithms for finding minimum distance of two trees [6]–[8], instead of those for two strings used in `diff`. We will describe the algorithm used by our tool in the next two sections.

3. Tree Representation of Program

3.1 Tree Representation

Since declarations and statements constitute a program, and also structures like iteration statements and selection statements are hierarchically included in a program, we use a tree as internal representation of a program. The tree is similar to but different from the parse tree generally used by many compilers, such that a node in the parse tree denotes either a token such as a variable name or a non-terminal that represents a substructure, such as a part of an expression. If such parse trees were used for comparison, not only much memory space would be needed but also long time would be spent in comparison.

A node of the tree exploited in the tool we have built, however, denotes a declaration, an expression, or a statement, so that the tree becomes much smaller than the corresponding compiler-like parse tree.

Our internal structure would have similarity to graphical representation methods for program structures, such as PAD (Problem Analysis Diagram) [9], where arranging overall program structure in easily understandable way is one of main concerns. Since our method mainly aims to differentiate programs efficiently and exploit structural changes naturally, it employs limited and simple features.

Nodes and edges of the tree are constructed as follows:

- The root of the tree has no label. (To make handling easy, assume all roots to have the same label.)
- A leaf denotes preprocessing line, or a variable declaration, or a simple statement such as assignment

statement or print statement. The label of the node is the content of the declaration or statement.

- An internal node denotes one of a function definition, a structure definition, a selection statement or an iteration statement. The label is the head of the statement or the structure.

- If node X is in the body of node Y statement, then connect an edge from X to Y.

An example of such a tree is shown in Fig. 4. Each number in a circle represents the node's number in the tree in the preorder; The text outside the circle represents the label of the node.

Therefore, all trees discussed in this paper are rooted, ordered and labeled and the preorder traversing result of nodes of the tree returns the original program. We list some notations with respect to trees used in this paper:

$|T|$: the number of nodes of T;

v_i: the ith node of T according to preorder traversal;

l_i: the label of node v_i;

$T(i)$: the subtree of T with node v_i as root.

When we want to distinguish two trees T and T', we would use notations $|T'|$, v'_i, l'_i, and $T'(i)$ for tree T'. Also, the expression $x < y$ for nodes x and y means $i < j$, where $v_i = x$ and $v_j = y$.

3.2 Difference with Respect to Mapping

We consider mapping $M = \{(i,j) \mid l_i = l'_j,\ 1 \leqq i \leqq |T|,\ 1 \leqq j \leqq |T'|\}$ from tree T to tree T', and define the difference between two trees T and T' with respect to M to be the sequence of the following two operations:

Deleting nodes of $T - \{v_i \mid v_i \in T,\ i \in Domain(M)\}$;

Inserting nodes of $T' - \{v'_j \mid v'_j \in T',\ j \in Range(M)\}$;

where $Domain(M) = \{i \mid (i,j) \in M\}$ and $Range(M) = \{j \mid (i,j) \in M\}$.

For T and T', if M is given, we can easily obtain the difference between them.

Certainly a mapping must satisfy some practical constraints, otherwise no meaningful difference between two trees could be obtained. In Fig. 5 (a), e.g., all nodes of T and all nodes of T' are covered by mapping $\{(1,1),(2,2),(3,3)\}$, therefore no difference with respect to the mapping would be generated, though the two trees are obviously different. Tai proposed constraints on M as follows [6]:

For any two pairs $(i_1,j_1),(i_2,j_2) \in M$,

a) $i_1 = i_2$ iff $j_1 = j_2$;

b) $i_1 < i_2$ iff $j_1 < j_2$;

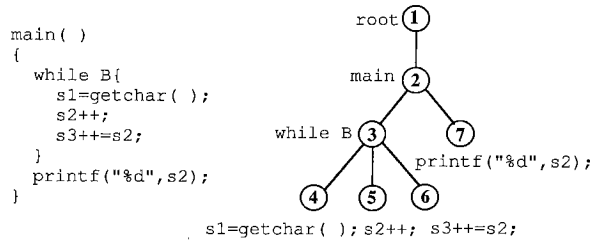


Fig. 4 An example of program and its tree representation.

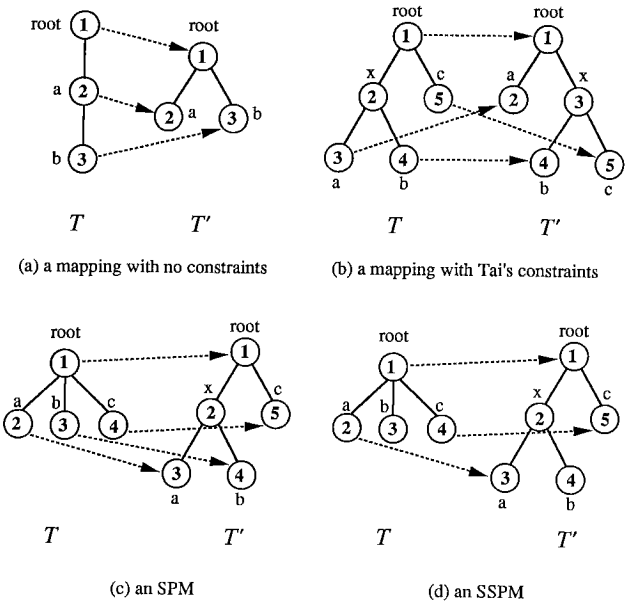


Fig. 5 Mappings from T to T'.

c) v_{i_1} is an ancestor (descendant) of v_{i_2} iff v'_{j_1} is an ancestor (descendant) of v'_{j_2}.

Figure 5 (b) shows an example of Tai's mapping, and the corresponding operation sequence becomes "deleting v_2 from T followed by inserting v'_3 into T'". This sequence, however, apparently does not reflect the practical tailoring process from T to T', since it could destroy existing sibling relations and create new ones. Adding the following constraint d) to Tai's definition, we get a more strongly constrained mapping. It was proposed by Tanaka et al., and is called Structure Preserving Mapping (*SPM*) [7], [10].

d) For arbitrary nodes v_1 and v_2 of T, when R_{v_1} and R_{v_2} are determined,

$$el(v_1) < v_2 \text{ iff } el(R_{v_1}) < R_{v_2}.$$

For a given node s in T, and a mapping M from T to T', $el(s)$ expresses the largest leaf under preordering in subtree $T(s)$. R_s is the root of the minimum subtree in T' that covers all images of the nodes of $T(s)$ under M but including no images of any nodes other than $T(s)$. It is formally defined below:

$R_s = max\{\cap_{h\in\Phi(s)} An(h)\}$,
where
$\Phi(s) = \{v'_j | (i,j) \in M$ and v_i is a node of subtree $T(s)\}$,
$An(h) = \{a | a$ is an ancestor of h including h itself$\}$.
In Fig. 5 (b), for example,
$el(v'_1) = v'_5$,
and
$R_{v_2} = max\{\cap_{h\in\{v'_2,v'_4\}} An(h)\} = v'_1$.
From the mapping $M = \{(1,1),(2,3),(3,4),(4,5)\}$ in Fig. 5 (c), an operation sequence "v'_2 with label x is inserted to T'" is generated, which does not reflect the practical tailoring process either since it inserts nodes into some subtree structure.

In order not to allow such operations, we consider a mapping where both M and M^{-1} (the inverse of M) are *SPM*s. This kind of mapping, called Strongly Structure Preserving Mapping (*SSPM*) was also proposed by Tanaka [8]. Figure 5 (d) shows such a mapping, which generates "v_3 is deleted from T followed by v'_2's insertion to T' followed by v'_4's insertion as v'_2's child to T'." This is a reasonable and practical sequence that transfers T to T'. In case of Fig. 5 (c), we have $M^{-1} = \{(1,1),(3,2),(4,3),(5,4)\}$ and it is not an *SPM*†.

The above discussion shows that *SSPM* would suit to the mapping for the difference between two trees. In the next section, we present an algorithm to calculate the maximum *SSPM*.

4. Algorithm of Calculating Maximum Mapping

From the definition of difference between two trees above, it is obviously that the difference becomes small when $|M|$ becomes large. So calculating the maximum *SSPM* constitutes the basis for obtaining the minimum difference between trees T and T'. Tanaka proposed an algorithm to compute the distance between two trees under *SSPM* [8], which can be used for our purpose but includes many unnecessary parameters and steps. Thus, we propose a simple and straight-forward algorithm to obtain maximum *SSPM*.

Suppose the roots of T and T' be v_1 and v'_1 respectively, and φ indicate the *SSPM* that makes $|M|$ maximum. φ is collected by the recursive procedure *Map* as follows:

$\varphi := \phi$;
$Map(1,1)$;

procedure Map(r,r');
begin
  if $l_r = l'_{r'}$ **then** $\varphi := \varphi \cup \{(r,r')\}$; **endif**
  case $A_{r,r'}$ **of**
  "$match$": **for** each $(v_i, v'_j) \in X_{r,r'}$ **do** $Map(i,j)$;
  "$scan\_v_{r_p}\_v'_{r'}$": $Map(r_p, r')$;
  "$scan\_v_r\_v'_{r'_q}$": $Map(r, r'_q)$;
  endcase
end;

We need to calculate $X_{i,j}$ and $A_{i,j}$ for each node v_i of T and each node v'_j of T', where $X_{i,j}$ is a set of node pairs, indicating the most weighted non-cross matching of the bipartite graph $B(I,J,E_{i,j})$, where $I = \{v_{i_1},\ldots,v_{i_m}\}$ and $J = \{v'_{j_1},\ldots,v'_{j_n}\}$ are the set of v_i's children and the set of v'_j's children respectively, and $E_{i,j} = \{(v_{i_s}, v'_{j_t}) |\ v_{i_s} \in I, v'_{j_t} \in J$ and $H_{i_s,j_t} \geqq 1\}$. $A_{i,j}$ is a string, indicating the next direction of the recursion. H_{i_s,j_t} is the maximum number of same-label pairs (x,y) under *SSPM* in which $x \in T(i_s)$, $y \in T'(j_t)$, and it is also called the edge's weight.

Let $T = (V,E)$, $T' = (V',E')$, $L = \{v | v \in V$ and v is a leaf$\}$, $L' = \{v' | v' \in V'$ and v' is a leaf$\}$. The calculation of the values of H, X and A proceeds bottom-up from pairs of leaf nodes to the pair of roots and is divided into three main steps as described informally below††. A more formal one is shown in Appendix. Note that it is obvious that $X_{i,j}$ becomes empty when v_i or v'_j or both are a leaf node.

Step 1: For arbitrary $(v_i, v'_j) \in L \times L'$, $H_{i,j}$ is set to 1 if $l_i = l'_j$ and 0 otherwise. Also, $A_{i,j}$ is always set to "$match$" and $X_{i,j}$ is always set to empty here.

Step 2: For arbitrary pair of a leaf and a non-leaf node, e.g., $(v_i, v'_j) \in (V - L) \times L'$, then we set $H_{i,j} = 1$ and $A_{i,j} =$" $match$" if $l_i = l'_j$. Otherwise, if there exists v_{i_p} such that $H_{i_p,j} = 1$†††, then we set $H_{i,j} = 1$ and $A_{i,j} =$" $scan\_v_{i_p}\_v'_j$" so that recursion proceeds to the pair (v_{i_p}, v'_j). In any other cases, we set $H_{i,j} = 0$ and $A_{i,j} =$" $match$". $X_{i,j}$ is set to empty for all cases.

Step 3: For arbitrary $(v_i, v'_j) \in (V - L) \times (V' - L')$, $X_{i,j}$ is first computed by using such an algorithm proposed in [11]. Then W, the weight of $X_{i,j}$ is compared with $max\{H_{i_1,j},\ldots, H_{i_m,j}\}$ and $max\{H_{i,j_1},\ldots,H_{i,j_n}\}$†††† and the maximum one is assigned to be $H_{i,j}$ with a possible increment. $A_{i,j}$ is determined depending on the comparison of these values.

The complexity of the algorithm is analyzed as fol-

†Suppose it were *SPM*, it would satisfy the above constraint d). However, $el(v'_2) = v'_4$ is less than v'_5 in preorder, but $R_{v'_2} = v_1$,thus $el(R_{v'_2}) = el(v_1) = v_4 = R_{v'_5}$, and this is a contradiction.

††We have designed an algorithm to obtain maximum *SSPM* for general trees. However, in the tree that represents a program internally, no two nodes on a path from the root to a leaf would have the same label. So we take advantage of this feature in this paper for easy understanding.

†††Note that since v_{i_p} is a child node of v_i and the calculation is bottom-up, $H_{i_p,j}$ has been computed already.

††††$H_{i_s,j}$ $(1 \leqq s \leqq m)$ and H_{i,j_t} $(1 \leqq t \leqq n)$ have been known already.

lows:

In the first step, $O(|L| \times |L'|)$ operations are required. In the second step, $O((|V| - |L'|) * |L| + |L| * (|V'| - |L'|))$ operations are required. In the third step, $\sum_{i=1}^{|V|} \sum_{j=1}^{|V'|} O(n_i \times n'_j) = O(|V| \times |V'|)$ operations are required, where n_i and n'_j are the number of child nodes of v_i and the number of child nodes of v'_j respectively. Thus, the time complexity of the algorithm is $O(|V| \times |V'|)$. Obviously, the space complexity is also $O(|V| \times |V'|)$.

The actual execution time of this algorithm on SPARCstation ELC with SunOS 4.1.1 is, e.g., 0.02 Seconds for the programs shown in Fig. 1, together with 0.08 Seconds for parsing two versions of C programs into the internal tree structures. Also, the execution times for a pair of C programs of about 100 lines including three functions are 0.2 Seconds and 1.1 Seconds respectively. The overall execution times of our tool is dominated by fairly slow parser, remaining room for improvement. The execution time of this mapping algorithm actually does not grow in the square order of program size, since our tool applys the algorithm to each C function having the same names in two versions. The sizes of the C functions would not grow in proportion to the total program sizes.

5. Program Difference Tool

5.1 Display Method

As we know, it is not easy to locate the structural differences between two programs by looking at the output produced by `diff`. In the design of our tool, we exploit pretty-printing technique to generate 'pretty' output. In pretty-printing, an internal representation (in the case of our tool, the tree) is traversed, and tokens of the nodes are printed when the nodes are visited. The addition of spaces and blank lines at appropriate places is also necessary.

In order to produce visual output of program comparison, the text lines where changes occur are printed at a color that is different from the color of unchanged text if a color display is available. On white/black display, underlines or highlights are used to distinguish changed text from normal text. Also, two programs are pretty-printed simultaneously to color the different parts. The two trees, which represent the two programs to be compared, are traversed in pre-order. And the traversals are arranged in such a way that corresponding nodes will be visited at the same time.

When a node v of tree $T1$ that has no corresponding node in the other tree $T2$ is visited, the label of node v is printed and colored by color $C1$ on the output for $T1$. At the same time, a blank line is printed out for $T2$. The traversal of $T1$ advances to the next node while the traversal of $T2$ remains at the same node.

When a pair of corresponding nodes is visited, if their labels are different, those labels are printed and colored by color $C1$ and $C2$ respectively on the outputs for $T1$ and $T2$. If both have the same label, those labels are printed by regular text color on the outputs for $T1$ and $T2$ respectively. The traversals of both $T1$ and $T2$ advance to the next nodes.

When the traversals reach a point where neither node has a corresponding node in the other tree, the traversal of one of the trees proceeds by itself until a node that has a corresponding node is reached. And then the traversal of the other tree proceeds.

The resulting pretty-printed programs are displayed in a window side by side, as shown in Fig. 3. Therefore, the differences can be easily viewed.

5.2 Other Features

A program may be composed of many procedures and functions (in case of C functions only). In many development activities, e.g., fixing bugs to get a new program, what usually occurs is that only few functions of the program change while other ones keep unchanged. So displaying the whole programs may be a waste of time and space for some programmers who concern with changed parts only. In the design of our tool, we provide an optional functionality that displays whole functions only if they change and displays only declaration parts of those functions without changes.

We also provide optional functionalities for programmers who concern the old program (the program before modification) or the new one (the one after modification) only. This feature is implemented by displaying the pretty-printed old (or new) program only in a window.

6. Conclusion

We have developed a tool that identifies differences between two programs. Since programs have nested structure such as loops and conditional statements, a tree similar to but different from parse tree is exploited as the internal representation of a program in this paper. The difference between two trees with respect to mapping has been defined and the fitness of *SSPM* to tree differences has been discussed. A straight-forward algorithm that obtains the maximum *SSPM* between trees has been proposed. Based on the maximum *SSPM*, and by exploiting pretty-printing technique, we produce visual program differences that can be identified immediately. In order to make the tool useful for displaying differences of relatively large programs that may be composed of many functions, we provide a feature that displays changed functions only. The tool has been implemented for C language programs and is being lifted to new versions.

The future work on this topic includes the representation method of the generated differences for various purposes. Especially when the tool is applied to collect the updating history of program texts for software reliability growth model [12], the representation would become one of the key factors. The difference tool for general documents, e.g., design documents, would be also an important research issue.

Acknowledgement

The authors thank Takeshi Ogihara and Hajimu Iida for useful discussions on this paper and the anonymous referees for many important comments on the earlier version of this paper.

References

[1] SunOS 4.1, On-line manual, section 1, Oct. 1989.
[2] D.S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," Comm. ACM, vol.18, no.6, pp.341–343, 1975.
[3] J.W. Hunt and T.G. Szymanski, "A fast algorithm for computing longest common subsequences," Comm. ACM, vol.20, no.5, pp.350–353, 1977.
[4] D.S. Hirschberg, "Algorithms for the longest common subsequence problem," J.ACM, vol.24, no.4, pp.664–675, 1977.
[5] L. Lian, K. Inoue, and K. Torii, "An experiment of program comparison tool based on the mapping between parsing trees," IEICE Technical Reports, SS 93-18, July 1993.
[6] K.C. Tai, "The tree-to-tree correction problem," J.ACM, vol.26, no.3, pp.422–433, July 1979.
[7] E. Tanaka and K. Tanaka, "A metric on trees and its computing method," IEICE Trans., vol.J65-D no.5, pp.511–518, May 1982.
[8] E. Tanaka, "The metric between trees on the strongly structure preserving mapping and its computing method," IEICE Trans., vol.J67-D no.6, pp.722–723, June 1984.
[9] Y. Futamura, T. Kawai, H. Horikoshi, and M. Tsutsumi, "Development of computer programs by problem analysis diagram (PAD)," Proc. of 5th International Conf. on Software Eng., pp.325–332, San Diego, California, 1981.
[10] E. Tanaka, "A correction of paper: the metric on trees and its computing method'," IEICE Trans., vol.J76-D-I, no.11, p.635, Nov. 1993.
[11] I. Mori, S. Masuda, and E. Tanaka, "An algorithm for finding a maximum weight noncross matching of a cyclic bipartite graph," IEICE Technical Reports, COMP 92-23, July 1992.
[12] S. Kusumoto, K. Matsumoto, T. Kikuno, and K. Torii, "On a measurement environment for controlling software development activities," IEICE Trans., vol.E74-D, no.5, pp.1051–1054, May 1991.

Appendix: The Algorithm to Calculate H, X and A for each v_i and v'_j.

Step 0. Set all node pairs $(v_i, v'_j) \in V \times V'$ to be unmarked.

Step 1. For each unmarked $(v_i, v'_j) \in L \times L'$,

if $l_i = l'_j$
  then $H_{i,j} := 1$
  else $H_{i,j} := 0$;
endif
$X_{i,j} := \phi$;
$A_{i,j} :=$" $match$";
Put a mark to the pair (v_i, v'_j).

Step 2. For each unmarked $(v_i, v'_j) \in (V - L) \times L'$ such that (v_{i_k}, v'_j) has been marked for arbitrary child node v_{i_k} of v_i.

if $l_i = l'_j$ **or** there exists v_{i_p} such that $H_{i_p,j} = 1$
  then $H_{i,j} := 1$
  else $H_{i,j} := 0$;
endif
$X_{i,j} := \phi$;
if $l_i \neq l'_j$ and $H_{i_p,j} = 1$
  then $A_{i,j} :=$" $scan\_v_{i_p}\_v'_j$"
  else $A_{i,j} :=$" $match$";
endif
Put a mark to the pair (v_i, v'_j).

H, X and A can be similarly computed for each unmarked $(v_i, v'_j) \in L \times (V' - L')$ such that (v_i, v'_{j_k}) has been marked for arbitrary child node v'_{j_k} of v'_j.

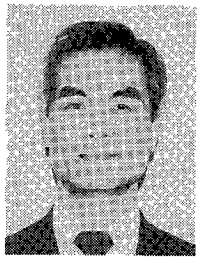
Step 3. For each unmarked $(v_i, v'_j) \in (V - L) \times (V' - L')$ such that (v_{i_s}, v'_{j_t}), (v_{i_s}, v'_j) and (v_i, v'_{j_t}) have been marked for arbitrary child node v_{i_s} of v_i $(1 \leq s \leq m)$ and arbitrary child node v'_{j_t} of v'_j $(1 \leq t \leq n)$, compute $X_{i,j}$ using an algorithm to find maximum-weighted non-crossing matching [11]. Let $W = weight(X_{i,j})$, v_{i_p} be the node such that $H_{i_p,j} = max\{H_{i_1,j}, \ldots, H_{i_m,j}\}$, v'_{j_q} be the node such that $H_{i,j_q} = max\{H_{i,j_1}, \ldots, H_{i,j_n}\}$. $H_{i,j}$ and $A_{i,j}$ are calculated as follows:

if $H_{i_p,j} < W$ and $H_{i,j_q} < W$
  then $H_{i,j} := W$; $A_{i,j} :=$" $match$"
  else if $H_{i_p,j} \geq W$ and $H_{i_p,j} \geq H_{i,j_q}$
    then $H_{i,j} := H_{i_p,j}$;
      $A_{i,j} :=$" $scan\_v_{i_p}\_v'_j$"
    else $H_{i,j} := H_{i,j_q}$;
      $A_{i,j} :=$" $scan\_v_i\_v'_{j_q}$";
  endif
endif
if $l_i = l'_j$
  then $H_{i,j} := H_{i,j} + 1$;
endif
Put a mark to the pair (v_i, v'_j).

Lin Lian received B.S. and M.E. degrees in computer science from Sichuan University, Chengdu, China, in 1984 and 1987 respectively. He is now a Ph.D. candidate in the Department of Information and Computer Sciences, Osaka University. From 1987 to 1991, he joined the Department of Computer Science, Sichuan University. His research interest includes program difference tool, program slicing and software test process.

Minoru Aizawa received the B.E. and M.E. degrees in information and computer sciences from Osaka University in 1993 and 1995, respectively. He joined the Toshiba Corporation in 1995. He studied on program difference tool and program development history analysis as his master thesis at Osaka University.

Katsuro Inoue received the B.E., M.E. and Dr.Eng. degrees in information and computer sciences from Osaka University in 1979, 1981, and 1984, respectively. He joined the Department of Information and Computer Sciences, University of Hawaii at Manoa, as an assistant professor in 1984. He worked as a research associate at Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University from 1986. He is currently an associate professor. His research interest includes software development environment, software reuse, and software process. He is a member of IEEE, ACM, IPSJ, and JSST.



Koji Torii received the B.E. and M.E. degrees in communication engineering in 1962 and 1964 respectively, and the Ph.D. degree in electronic engineering in 1967 from Osaka University. In 1967 he joined the Electrotechnical Laboratory. From 1984 to 1995 he was a Professor of Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University. Since 1992 he has been a Professor of Graduate School of Information Science, Nara Advanced Institute of Science and Technology. His research interest includes software engineering, especially software metrics, software development environments, software education and GO-game computer programs. He has been a member of Editorial Board of IEEE Transactions on Software Engineering and of IEEE Software. He was a program co-chair of the 13th International Conference on Software Engineering. He is a member of IEEE, ACM, and IPSJ.