Revision Control System Using Delta Script of Syntax Tree

Yasuhiro Hayase[1], Makoto Matsushita[1], and Katsuro Inoue[1]

Graduate School of Information Science and Technology, Osaka University, 1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
{y-hayase, matusita, inoue}@osaka-u.ac.jp

Abstract. In an opensource development process developers work together using a revision control system. While getting multi-developers working products together into a single form, merge feature of revision control systems is used. Nowadays, merge operations in existing systems are commonly implemented with a line-by-line approach that can fail if two changes to the same line of code happen at the same time.
In this paper, we propose a two-way merge algorithm for source code that exploit the tree structure of modern programming language grammar: the source code is transformed in an intermediate XML representation and the merge operation is conducted on the transformed version.
We give an implementation of the algorithm for the Java language for the subversion revision control system.
Experiments shown that the proposed algorithm gives more accurate merge result than the existing line-by-line algorithms.

1 Introduction

The recent explosion in popularity of the Internet has attracted increasing attention on the open-source development process. In an opensource development effort, the developers distributed all over the world cooperate with each other in parallel, share information by email and mailing list, and manage the products with a centralized revision control system.

In such environment, each developer works side by side. And the results are stored independently in the repository of revision control system. At this time, the developer may have to merge his/her own modifications with the other developers' modifications. The merging operation can be done by humans, or done by the revision control system; usually the preference is given to the revision control system for the purposes of correctness and certainty.

The unit of merging of many existing revision control system is line of code. It may raise some problems when a merge is performed:

Illegal output Under the hypothesis that two developers are working at the same time on the same source file, if one developer deletes a variable and a statements that uses that variable, and the other developer adds a statement using the same variable, then the two modifications shouldn't be merged. However, "line of code" merging can't detect even if two changes conflict, and it outputs an illegal source code.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SCM 2005, September 5-6, 2005, Lisbon, Portugal. Copyright © 2005 ACM

Fail in feasible merging If one developer changes a statement and the other developer adds a comment to the line containing the previous statement, then these modifications can be merged. However, existing systems judge that these modifications cannot be merged, simply because they share the same line.

There is a method which recognizes syntax of the programming language and merges source codes more accurately as one of the methods of solving these problems. In [6] a syntax-based method for accurate merging is proposed. However, due the large number of programming languages it is hard to define an accurate algorithm for each language.

In this paper, we propose a merging algorithm independent from any specific programming language. The algorithm first builds an intermediate representation of the syntax tree of the merging source code before actual merging.

We use an intermediate language based on XML. Matchings of nodes and the deltas between trees are obtained by applying delta calculation algorithm based on FMES algorithm [3]. The delta is the sequence of operations required to modify tree. Merging is accomplished applying the sequence of operations to the tree. The language dependent procedure is separated from the merging procedure.

We implemented this algorithm on the actual revision control system subversion [1]. The system targets Java source codes, and correctly deals with deletion of variable and moving of fragment of source code.

Moreover, we compared our merging algorithm with the algorithm implemented in other system. Concretely, we compared the output of our system and existing system using sample source codes and source codes extracted from development history of actual opensource development. As a result, our system merge achieve a more accurate merging than existing system.

The paper consist as follows: Section 2 represents common revision control system and shows problem of existing revision control system. Section 3 explain storing the intermediate tree representation of the source code in repository. Section 4 shows merging system which recognizes syntax of source code as a solution of the problem. Section 5 explains the implementation of the system. Section 6 shows experimentation and consideration. Section 7 compares our system and related works. Finally concluding remarks and future work are given in Section 8.

2 Revision Control System

First, we represent revision control system used in opensource software development commonly. And next, we explain by using the example of the problems when the results of the works concurrently done are merged.

2.1 Software Development using Revision Control System

Revision control system (e.g. CVS) manages modification history of files such as source codes and documents.

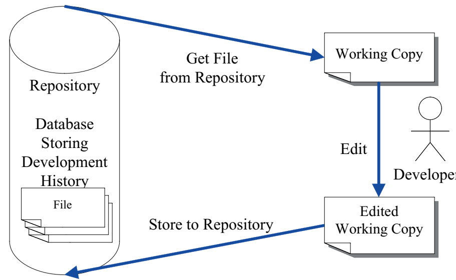


Fig. 1. Revision Control System

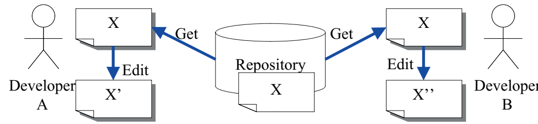


Fig. 2. Parallel development by multi developers

Figure 1 shows a simple workflow of software development using revision control system. Revision control system has a database called **repository** which stores all of developed files. Developer must get copy of a file called **working copy** from repository before modifying the file. And after the developer finish modifying, the developer stores the file in the repository. Software development using revision control system is achieved repeating this workflow.

Also, many of revision control systems support the software development by multiple developers. Figure 2 shows that developer can get copy of a file even if other developers hold copy of the file, and that each developer can modify working copy at will.

When developers modify a file in parallel in this way, modification except for last stored one is lost if each developer stores result of own modification only. To prevent losing other developers' modifications, the file is needed which includes own and other developers' modifications (Figure 3). The procedure getting such file is called **merging**. Merging is done by revision control system or by hand.

2.2 Problems on merging

Existing revision control systems merge line-by-line, and detect conflict if and only if the same line is modified. It has problem of detecting unnecessary collisions or outputs illegal source codes. Following sections show each problem and ideal solution.

Unnecessary collision Suppose that developers A and B get working copies of same file and modifying it. The file had following line.

```
int refs;
```

Developer A modified the line to initialize variable `refs` as follows and stored

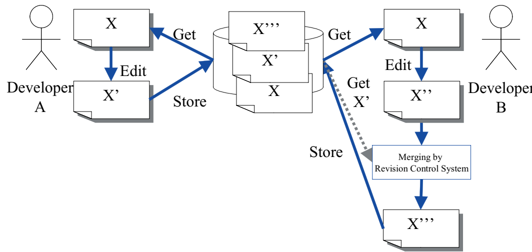


Fig. 3. Commit merged result

to repository.

```
int refs=0;
```

Developer B added comment about variable `refs` in same line as follows before knowing the modification of A and attempted to store.

```
int refs; /* reference count */
```

Developers A and B modified same line, so the revision control system judge that these modifications conflict. In this case, developer B must resolve the conflict because B stores the file after A.

If the system recognizes modifications right way, the system should output following code having both initializing and comment.

```
int refs=0; /* reference count */
```

Illegal merging output Please assume developers A and B get working copies of same file and modifying it. The file had following line which declares three variables.

```
int num, sum, avg;
```

Developer A considered that variable `avg` is not necessary, and deleted it as follows.

```
int num, sum;
```

Developer B added a statement using variable `avg` as follows before knowing the modification of A and attempted to store.

```
int num, sum, avg;
:
avg = num/sum;
```

Developer B have to merge own modification and Developer A's modification before storing. In this case, the revision control system outputs source code as follows because the modifications are not on same lines.

```
int num, sum;
:
avg = num/sum;
```

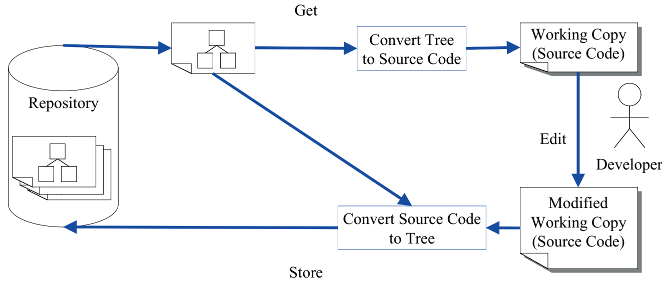


Fig. 4. Data flow on storing modified working copy

However, undeclared variable `avg` is used in this source code. This result contradicts the intention of B. If a variable having same name is declared at outer scope, the developer might not sense the problem because no error is detected on compiling this source code.

If the system recognizes modifications right way, the system should detect collision.

3 Storing source code to repository

In order to resolve problems described in Section 2.2, we propose a revision control system which stores in the repository not the source codes but the intermediate tree representation of the source code. The tree is a syntax tree with white space strings and comments. Specifically, the tree is an ordered tree with no limitations in the number of child nodes. All nodes of the tree have unique IDs. Details about IDs are described in Section 3.2. Data structure of the tree is independent from programming language, so the system does not depend on specific programming language.

3.1 Analyze source code to syntax tree

When developer adds a new source code, the system automatically build a new tree by parsing the source code, and add new unique IDs to all nodes of the tree. When a file is checked out, the system fetch the matching tree from the repository and converts it to source code (Figure 4). The source code is obtained from the tree by doing a depth-priority-search visit and printing the text stored in the node. Developers can then modify the source code and commit the changes to the repository. When the system stores the source code, it parses the source code again, makes a tree without IDs. And it searches for any pairs of similar nodes between the modified source tree and the original source code tree. Any node in the modified source code tree that belongs to the pair has assigned the same ID of the other node in the pair. Any node that doesn't belong to some pair has assigned and unique ID by the system.

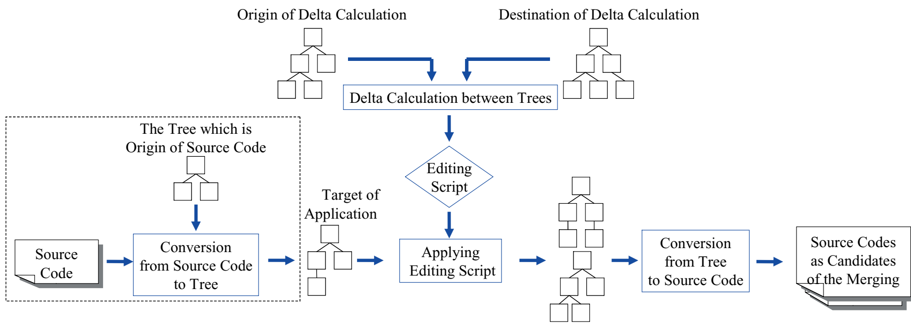


Fig. 5. Data flow on merging

3.2 Adding IDs to nodes

To identify the same node in different trees, the system put IDs to nodes. The procedure to put IDs to the nodes is as follows. The tree that is the origin of the source code is stored in the repository of the system, and the nodes of the original tree have IDs already. First, the system searches for the pairs of two nodes by comparing A and B. This procedure is called **matching calculation** at the following. The result of the calculation is a bijective map f between nodes in A' and B' which are the subsets of nodes in A and B respectively. The map f associates nodes which are judged identical on the basis of the data in the nodes or similarity of parent or brothers. Nodes in A' get the same IDs as nodes which correspond in f respectively. Nodes not in A' but in A get new unique IDs respectively. The FMES algorithm[3] is used for matching calculation. Then, the system makes links from the nodes which use a variable or a function to the nodes which declare them.

4 Merging algorithm recognizing syntax of source code

In this section, we explain the merging algorithm that exploits the syntax of the source code. This algorithm targets the tree described in section 3.

Figure 5 shows data flow in merging procedure. The inputs of the procedure are (1) the modified source code tree, (2) the original source code tree, and (3)the destination source code tree.

At first, the difference between (2) and (3) tree are computed. The difference is expressed as a sequence of tree editing operations called **editing script**. Multiple trees are output by applying the editing script to the first tree.

4.1 Delta calculation between trees

The delta between trees is expressed as an **editing script**. An editing script is a sequence of **tree editing operations** of four kinds:

1. `insert`: Add a node to the tree. Its parameters are the ID of the node, the data in the node, ID of the parent node, and a number which represent position in the brothers.
2. `delete`: Delete a node from the tree. Its parameter is the ID of the node.
3. `update`: Update the data of the node. Its parameters are the ID of the node, and the new data in the node.
4. `move`: Move a subtree around the tree. Its parameters are the ID of root node of the subtree, ID of the parent node in destination, and a number which represent position in the brothers.

The system edits a tree by applying sequentially the operations described in the editing script.

Generally, for two arbitrary trees A and B, there are an infinite number of scripts that may convert A into B. We define the cost of the editing script as the summed cost of each single operation in the script, and we adopt the script with lowest cost. The cost of operation is defined in delta calculation algorithm. We use FMES algorithm [3] extended by analysis of renaming as follows.

Renaming analysis FMES is a delta calculation algorithm for generic tree based on text similarity. Some problems can arise if it is applied unchanged to source code. There is a problem on identifiers matching. For example, it is typical that variable having similar role are named in a similar way, for example a common prefix and number or short name suffix. (ex. `buffer1, buffer2...`) These names are similar enough as text, so FMES may judge `buffer1` and `buffer2` are matched. When the variable's name is changed, text-based matching algorithm can't match renamed nodes.

Thus, a different algorithm is used for matching of nodes representing identifiers. First, matching of all the nodes but the identifiers' ones is calculated with the FMES algorithm. Next, the algorithm makes two sequences of the identifier nodes of each tree, and calculates the longest common subsequence (LCS) of nodes. The identifier nodes match if the nodes have exactly the same data or the ancestor nodes of the identifier nodes match at more than the certain rate.

The pairs of nodes in the LCS are added to the matching set. Matchings of the other identifier nodes are calculated by brute force method.

4.2 Applying editing script

This section describes the algorithm which applying the editing script from tree A to tree B to yet another tree C. The inputs of the algorithm are the editing script S, the tree C as target of editing, and the tree A as the origin of S.

The editing operations in the script use the node's ID to identify nodes. However, when applying S to C, for a given ID, there may be no matching node in C. In this case, similar nodes substitute the target node of the operation. (Figure 6)

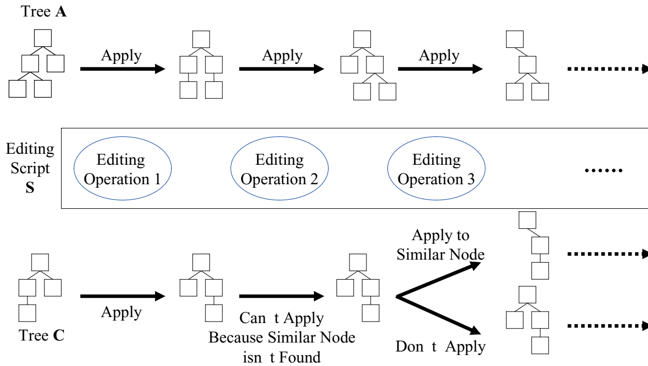


Fig. 6. Applying editing script to source code tree

Searching similar nodes and positions Similar nodes are searched in two ways. The first method is based on the similarity of sibling nodes (figure 7(a)). Consider the set made of all the pairs of nodes on both sides of the target node, and the subset of it whose elements satisfy the following conditions:

– Two nodes which have the IDs of both of pair exist in tree C.
– The two nodes in tree C have same parent node.

Only the pairs with the shortest distance between its nodes are picked up from the subset. The candidate nodes are searched between nodes of tree C corresponding to both of the pair. The candidate nodes must have similar text data, and the node having same ID as the candidate node must not exist in tree A.

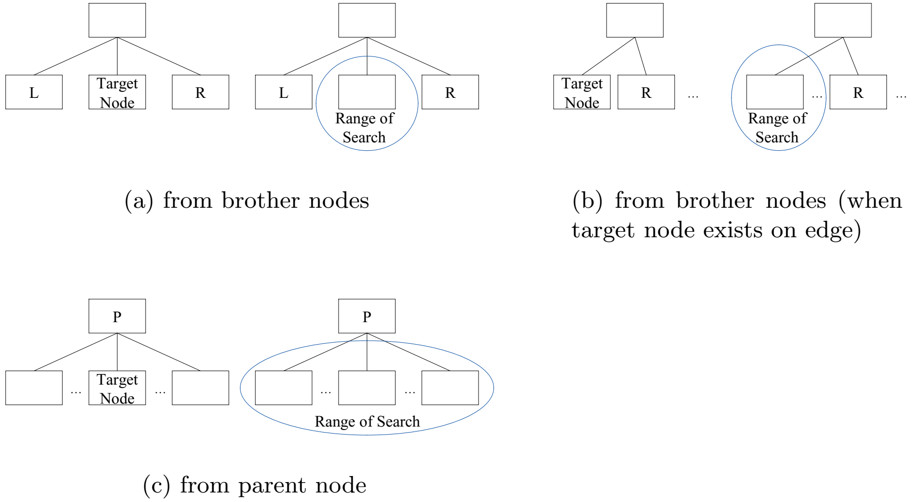
If target node exists on edge like figure 7(b), then the candidate nodes are searched using brother nodes in one side as a clue.

The second method uses parent node. Like figure 7(c), the candidate nodes are searched from children of the node having same ID as the parent node of the target node. The candidate nodes must have similar text data, and the node having same ID as the candidate node must not exist in tree A. This method is used only when the ID of parent of target node was not used in the first method.

The candidate nodes founded by these methods have a score. If a candidate node has same ID as target node, then the candidate node has 3 point. Else, the candidate node gets 1 point if each of left, right, or parent of the candidate node has same ID as corresponding node of target node.

The position as parameter of editing operation described as the ID of the parent node and the rank order in the brothers. Accordingly, every time when an editing operation is applied to tree C, it is necessary to search the similar position. The candidate positions are searched and scored based on left, right, and parent node of the target position as well as the candidate nodes.

Applying editing script to original tree It is necessary to apply the same part of the script S to the tree A in order to search similar nodes and positions. Thus the script S is applied to tree C and tree A simultaneously.



(a) from brother nodes

(b) from brother nodes (when target node exists on edge)

(c) from parent node

Fig. 7. Finding similar nodes

Applying editing script When applying the edit operations, if multiple candidate nodes or positions are found, then a tree is built for each possible combination of nodes.

When applying a `delete` operation whose target node has child nodes, two trees are built: one with the removed subtree, and one copy of the original tree.

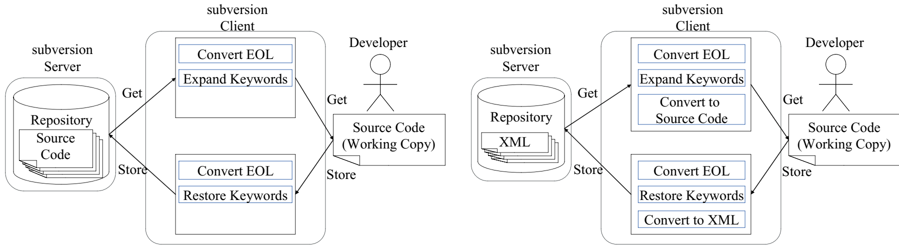
The algorithm records score of trees. Before applying the script, the score of tree is 0. When the algorithm applies each operation, The value calculated from how operation was applied is added to the score. The value is a score of the node or the position to which the operation is actually applied. In the case of `move` operation, the value is average of scores of the node and the position.

5 The implementation

This section describes the implementation of the system. We implemented the system by extending the Subversion revision control system. The implementation of the checkout and commit operation is described in section 5.1, and the implementation of the merging operation in Section 5.2.

5.1 Checkout and commit

Figure 8(a) shows the original checkout and commit operations as implemented in Subversion. Subversion is a client-server system, whose server manages the repository and the client manages the working copy. The client hash function converts the EOF character and replaces certain keywords when creating a working copy by checkout or committing to the repository.



(a) Checkout and commit of original subversion

(b) Checkout and commit recognizing tree structure of source code

Fig. 8. Checkout and commit of existing system and our system

We added the following functions to subversion client.

1. Conversion from tree to source code
2. Conversion from source code to tree

The tree is expressed in special intermediate language based on XML. The subversion server has not been modified. Figure 8(b) shows the outline of the extended subversion system. If the file which checked out or committed has no special meta data `svn:conversion-handler`, extended subversion behaves same as original subversion.

If the file has meta data `svn:conversion-handler`, the XML file is stored in the repository. When the extended client checks out the file, the client converts the XML file to source code and uses the source code as working copy. When the extended client commits the file, the client converts the source code to an XML file.

The following sections describe the implementation of conversion between source code and XML file.

Conversion from source code to XML file The system and make syntax trees keeping white space strings and the comments by analyzing the source code. The system creates an XML file whose elements all correspond to all nodes of the syntax trees. The white space strings and the comments are also included in the XML file.

And the system makes links from an identifier element using a variable or a method to identifier declaring the variable or the method. The following methods are used to express the links. The system generates a temporary ID, and put it to attribute `id` of the destination element of the link, and put the same temporary ID to attribute `ref` of the source element of the link.

Next, the system put unique ID to all elements of the XML file. When new file is added to repository, the system put new ID to all elements of the XML file.

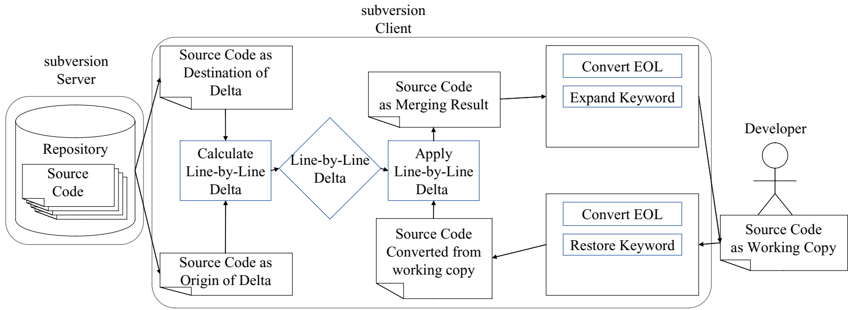


Fig. 9. Merging on original subversion

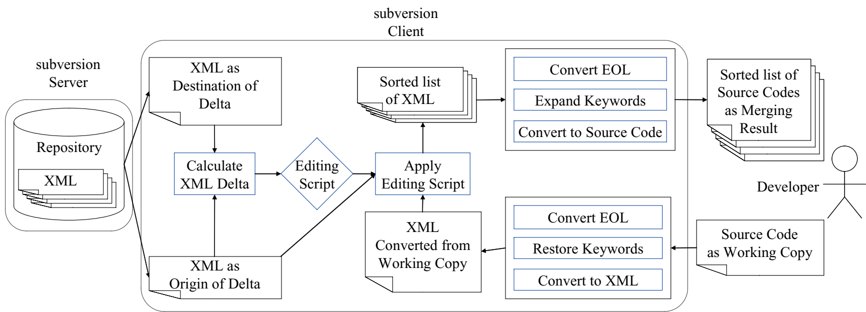


Fig. 10. Merging recognizing tree structure of source code

When file is not new one, the XML file before editing is stored in the repository, and elements of the XML file have already unique IDs.

Then, the system calculates the matching of nodes between the XML file to be committed and the XML file before editing. This calculation uses FMES algorithm and analysis of renaming described in Section 4.1. If the element in the new XML file corresponds to an element of old XML file, the system puts same ID as the element in old XML file to the element in the new XML file. If the element in the new XML file does not correspond to any node, the system puts new ID to the element.

The Universally Unique Identifier (UUID) is used as new ID. When the system puts ID to the element having temporary ID, the system also changes the ref attribute of elements which link to the element.

Conversion from XML file to source code The XML file is converted into the source code only by removing all tag from the XML file, because the XML document keeps all texts of the source code as a text element in order of appearing.

5.2 Merging

Figure 9 shows data flow when original subversion system merges files. The original subversion client merges files line-by-line.

The inputs of merging procedure are a working copy and the two files in repository. First, end-of-line character and keywords in working copy are restored. And, calculate the line-by-line delta of two files stored in the repository. Next, the client applies the difference to the restored working copy. Finally, the system expands the keyword and converts end-of-line characters of the application result. The working copy is overwritten using it.

We added to the subversion client the following functions.

1. Conversion from tree to source code
2. Conversion from source code to tree
3. Delta calculation between trees.
4. Delta application to tree.

Function (1) and (2) are described in section 5.1. Only the client is extended, and the server isn't changed. Figure 10 shows the outline of the merging function of our extended subversion. If the file has no meta data `svn:conversion-handler`, our system processes same as the existing subversion client.

If the file has meta data `svn:conversion-handler`, the system uses function of merging tree. In this case, as described in section 5.1, an XML file is stored in the repository and working copy is a source code.

First, the working copy edited by developer is converted into XML file as well as storing it. And, calculate the tree delta of two XML files stored in the repository. The delta is expressed as an editing script. The merge results are obtained by applying the delta to the XML file made from working copy.

The merging result has two cases. One is the single XML file is obtained. And another is the multiple XML files are obtained. If single XML file is obtained, the system converts the XML file to source code and overwrites the working copy using it. If multiple XML files are obtained, the XML files are sorted by the score described in Section 4.2. The developer must select one XML file. The system convert selected XML file to source code and overwrites the working copy using it.

Applying of delta When the system applies the delta to the tree, for the purpose of searching similar nodes or positions, the same delta is applied to tree which is the calculation origin of the delta at the same time. The system searches neither similar nodes nor the positions when the delta is applied to the tree which is the calculation origin of the delta.

Similar elements are searched when the element having target ID of operation does not exist or when the operation to the text element is applied. The `insert` operation and the `move` operation take a position in the parameters. So, similar positions must be searched when these operations are applied. The similar nodes and positions are searched by using the method described in Section 4.2.

When multiple similar nodes or positions are found or when any nodes or position are not similar enough, the tree is duplicated. The next operation in the editing script is applied to the duplicated trees as well as duplication origin.

The system verifies whether the trees to which all operations were applied fulfill the following requirements.

- The destination of link from identifier node does exist. If the destination doesn't exist, the system warns to developer.
- The names of the identifier are equal between the link source and link destination. If the names are different, the identifier has been renamed. The system changes the name of the link source to the name of the link destination.

6 Evaluation

In this section, we show the two experimentation results of our system.

6.1 Application to sample source codes

We checked whether our system can correctly merge the source codes made for the test.

At first, we made a source code. It is called the Original. And we made the three source codes that were made by modifying the Original in different way. Each source code is called the Variant 1, 2, 3. Each modification is described below.

Variant 1 The variable `x` declared in Original was deleted.
Variant 2 The method using variable `x` was added.
Variant 3 The variable `x` declared in Original was renamed.

The tree deltas from Original to Variant 1, 2, 3 were calculated. Each delta is called the Delta 1, 2, 3. We applied these deltas to the Variant 1, 2, 3 and check the result. Moreover, we did same experiment using line-by-line merging and compared the results. Table 1 and Table 2 show the experiment result.

The line-by-line merging system failed all combinations. It output an illegal source code or detected wrong conflict. (Table 1(a))

On the other hand, our system correctly merged excluding one. It output right source code or detected correct conflict. In one case that calculation failed, the calculation did not finish because it failed in the search for a similar position, and it had generated a huge amount of trees.

We recognized that we should improve the precision of the search for the similar nodes and the positions from this experiment result.

	Delta 1	**Delta 2**	**Delta 3**
Variant 1		Illegal Output	Conflict
Variant 2	Illegal Output		Illegal Output
Variant 3	Conflict	Illegal Output	

(a) Result of line-by-line merging

	Delta 1	**Delta 2**	**Delta 3**
Variant 1		Failure	Success
Variant 2	Dead link detection		Success
Variant 3	Success	Success	

(b) Result of tree merging

Table 1. Result of merging

6.2 Pseudo application to development of actual softwares

Target of the experiment We extracted the revision guessed to have used the merging function from the warehouse of actual opensource software. And we tried to reproduce the files before it was merged in those revisions. It is as follows concretely.

Please assume developers X and Y committed the same file F at short interval. Developer X committed at revision n and Developer Y committed at revision n+1. When the range of the change in revision n+1 is included within the range of the change in revision n, it is considered that developer Y checked out revision n, and edited it. When it is not so, it is considered that developer Y checked out revision n-1, edited it, merge the modification of developer X to working copy, and committed it.

The file before developer X change is merged is not stored in the repository. However, this file can be reproduced if it correctly merges it with the opposite direction. Then, we did the experiment that tried to reproduce this file using the merging function.

84 data in which committing happened within ten minutes was extracted from the repository of the Eclipse project (22606 files and 162683 revisions) and Jakarta projects (19420 files and 103358 revisions), and the data is used to experiment.

line-by-line merging	**count**	**tree merging**	**count**
Success	71	Success	71
Failed	13	Success	9
		Failed	4

Table 2. The result of merging actual softwares

cause of failure of line-by-line merging	**count**	**tree merging**	**count**
Addition or Deletion of White Space to the Same Line	4	Success	4
Semantic Change and Reform	1	Success	1
EOL Code Change	1	Success	1
Overlapped Semantic Change	2	Success	2
Overwriting Prior Change	2	Success	1
		Failure	1
Semantic Conflict	2	Failure	2
Broken Source Code	1	Failure	1

Table 3. details when line-by-line merging fails

Result of the Experiment Table 2 shows the result of the experiment. The tree merging has succeeded when succeeding in line-by-line merging. In addition, tree merging has succeeded in 9/13 in which line-by-line merging failed.

Next, details when line-by-line merging fails are shown in Table 3.

The reason for two cases of four cases in which "tree merging" failed is that it had tried to merge two edits conflicting semantically. A huge amount of candidates was generated when the editing operation was applied, and it failed in merging though edits are not conflict semantically.

7 Related Works

Wuu Yang [11] proposed the algorithm identifying syntactic differences between two versions programs.

T. Mens [8] classifies technologies of software merging by various criteria. Our algorithm is basically classified in three-way, syntactic, and operation-based merging.

Bernhard Westfechtel [10] proposed the syntactic 3-way merging algorithm. The algorithm is consist of language-dependent part and language-independent part, and detects conflict using the information of identifier. Our system is different from the algorithm of Westfechtel in two points. First, our system allows user to use arbitrary editors to edit the source code. Second, our system may output two or more candidates of merging result.

David Binkley et al. [2] proposed the semantic merging algorithm targeting the simple language having procedure call.

Marc Shapiro[9] and A. Kermarrec[7] propose the method of the synthesis of editing scripts by calculating the dependence between the operations in the each editing script, and re-ordering the operations in the scripts.

XmlDiff, DeltaXML, XML TreeDiff, diffmk, xydiff, etc. are enumerated as a system that calculates the difference of a general XML document. Because this research puts ID to the elements of the XML files, and simplifies the application of delta. This research distinguished from the above systems.

XCoP and Oracle XML DB are enumerated as a system that stores the XML document in the database and manages its version. These systems use the special operation for the editing of XML document. In our system, the developer can use arbitrary editor for editing source codes.

Shu-Yao Chien [5] [4] propose the method of making the time cost when an old version is acquired united to the amount of the disk use when the history XML document is stored.

8 Conclusion

This paper shows the problem of the merging function of existing revision control systems, and proposed the merging function according to the syntax of the source code as one solution to the problem, and described the design of the system. This system decreases the conflict on merging, and lightens the problem caused by merging.

The work in the future are covering the languages other than Java, improving precision of the matching, merging more efficiently, and spanning link between different files in the repository. It is also important to cover the document forms other than the source code.

Extension to Subversion done in this research is distributed as free software at following URL.
http://sel.ist.osaka-u.ac.jp/~y-hayase/svn-xml/

References

1. subversion. http://subversion.tigris.org/.
2. D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol.*, 4(1):3–35, 1995.
3. S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, 1996.
4. S.-Y. Chien, V. J. Tsotras, and C. Zaniolo. Version management of XML documents. *Lecture Notes in Computer Science*, 1997:184–189, 2001.
5. S.-Y. Chien, V. J. Tsotras, and C. Zaniolo. XML document versioning. *SIGMOD Record*, 30(3):46–53, 2001.
6. S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, 1989.

7. A. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Twentieth ACM Symposium on Principles of Distributed Computing (PODC)*, August 2001.
8. T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, 2002.
9. M. Shapiro, A. Rowstron, and A.-M. Kermarrec. Application-independent reconciliation for nomadic applications. In *Proc. SIGOPS European Workshop: "Beyond the PC: New Challenges for the Operating System"*, Kolding (Denmark), 2000.
10. B. Westfechtel. Structure-oriented merging of revisions of software documents. In *Proceedings of the 3rd international workshop on Software configuration management*, pages 68–79, New York, NY, USA, 1991. ACM Press.
11. W. Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21(7):739–755, 1991.