Operation-Based, Fine-Grained Version Control Model for Tree-Based Representation

Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, and Tien N. Nguyen

Electrical and Computer Engineering Department, Iowa State University, USA

Abstract. Existing version control systems are often based on text line-oriented models for change representation, which do not facilitate software developers in understanding code evolution. Other advanced change representation models that encompass more program semantics and structures are still not quite practical due to their high computational complexity. This paper presents OperV, a novel operation-based version control model that is able to support both coarse and fine levels of granularity in program source code. In OperV, a software system is represented by a project tree whose nodes represent all program entities, such as packages, classes, methods, etc. The changes of the system are represented via edit operations on the tree. OperV also provides the algorithms to differ, store, and retrieve the versions of such entities. These algorithms are based on the mapping of the nodes between versions of the project tree. This mapping technique uses 1) divide-and-conquer technique to map coarse- and fine-grained entities separately, 2) unchanged text regions to map unchanged leaf nodes, and 3) structure-based similarity of the sub-trees to map their root nodes bottom-up and then top-down. The empirical evaluation of OperV has shown that it is scalable, efficient, and could be useful in understanding program evolution.

1 Introduction

Software development is a dynamic process in which the software artifacts constantly evolve. Understanding code evolution is crucial for developers in the development process. However, most of the existing version control tools are text line-oriented which report the changes in term of added, deleted, or modified text regions. Those line-oriented models for changes disregard the logical structure of a program file. However, programmers, as well as compilers and program analysis tools, generally view a program as a structure of program elements, especially the fine-grained entities such as classes, methods, statements, expressions, etc. When a programmer modifies a program, (s)he always maintains in his/her mind the intention to change such elements. Therefore, reporting the changes in term of lines in existing version control tools would be less useful in analysis and understanding the program's evolution. A versioning tool that can support *change operations* on *fine-grained* program entities is desirable.

Recognizing that importance, researchers have proposed several models and tools for version control of fine-grained program entities [11–14]. Those

D.S. Rosenblum and G. Taentzer (Eds.): FASE 2010, LNCS 6013, pp. 74–90, 2010.

approaches, however, do not scale well because they generally rely on *total versioning* [16], which requires to store all versions of any entity for its own version history. In total versioning, a version of a compound entity refers to the versions of its constituent entities. Thus, as a new version is created for an entity, the corresponding new versions for all of its ancestors need to be created as well. Because fine-grained entities usually have many ancestors and are modified frequently, the versioning tools might need to create a huge number of versions of their ancestors. For example, when a variable is renamed, the enclosing statement(s), block(s), method, class, and package(s) are also considered as changed, and the new versions for all of them need to be created. This problem, called *version proliferation*, though might not involve physical copying, creates cognitive overhead and makes fine-grained versioning complicated. Consequently, it terribly affects the scalability of those fine-grained versioning tools [16].

Fortunately, version proliferation could be avoided using *project versioning* approach [2, 5, 16]. In project versioning, instead of storing the history of individual entity, the versioning tool stores only the history of the entire project. Any change committed to the repository will create a new version of the entire project. Then, the project's history is used to re-construct the histories of only requested entities. This improves the scalability over total versioning since the tool does not waste time and space for versions of non-involved entities.

In this paper, we propose OperV, a novel *operation-based* version control model that is able to support both *coarse* and *fine levels of granularity* in program source code. The key idea is the combination of *project versioning* and *operation-based* approaches. In OperV, a system is represented by a *tree-based hierarchy* (called *project tree*) of logical entities, such as packages, classes, methods, expressions, statements, etc. Changes to the system are represented via *tree edit operations* on the project tree. Only the project tree and its changes are stored. Versions and changes of the finer entities are re-constructed on demand.

To do those tasks, OperV provides the algorithms for identifying, differencing, storing, and retrieving the versions of program entities. All of those algorithms are based on the basis task: mapping the nodes of two versions of the project tree. Since a project tree is too large to be efficient for general tree edit script algorithms, we designed a mapping algorithm using several *divide-and-conquer* techniques specialized toward source code. The first technique is to separate between processing coarse- and fine-grained entities. The coarse-grained entities are processed first. Then, each coarse entity is considered as an independent subtree and is used for mapping of the finer-grained entities. The second technique is the mapping process with bottom-up and top-down phases, in which two nodes are considered as the candidates for mapping only if they have mapped descendants or ancestors. Candidate nodes are compared based on the structural similarity to find the mapped ones. The third technique is the localization of the mappings of leaf nodes based on their texts. The observation is that the leaf nodes belonging to an unchanged text region is considered as unchanged. Therefore, OperV maps a leaf node to the corresponding leaf node (in the other tree) if they belong to the corresponding changed or un-changed text regions.

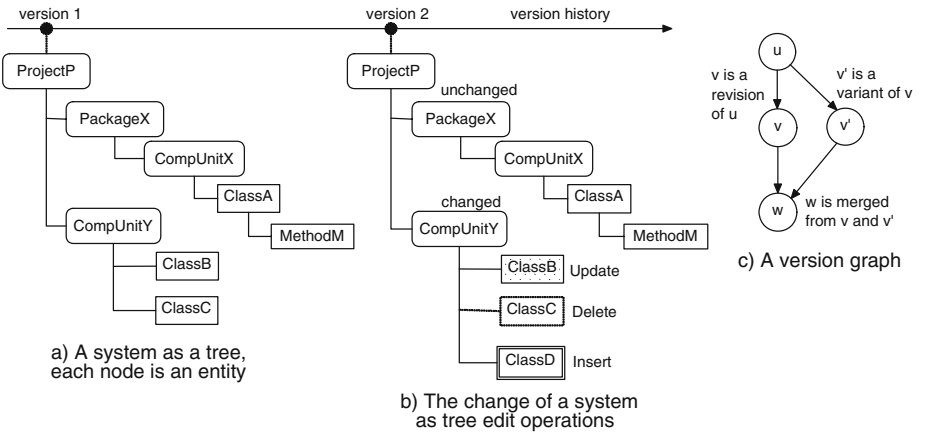

Fig. 1. OperV Concepts

The key contributions of our paper include

1. A scalable version control model that combines project versioning and operation-based approaches to support version control for both fine-grained and coarse-grained entities in a software system,

2. The associated algorithms for identifying, differencing, storing, and retrieving the versions of the system and the entities, and

3. An empirical evaluation to show the quality of OperV and its algorithms.

Section 2 describes the formulation of our model. Sections 3 and 4 present the details of the algorithms in OperV. Sections 5 discusses the evaluation. Section 6 presents related work and conclusions appear last.

2 Formulation

2.1 Concepts

Entity and Project Tree. In OperV, a software system is modeled as a *project*. A project is a hierarchical tree-based structure of *packages* and *compilation units*. A package could consist of other packages and/or compilation units. Each compilation unit is also a hierarchical structure of classes, methods, and other program units as in an abstract syntax tree (AST). The entities within a compilation unit are called *fine-grained*, and the other entities are *coarse-grained*. Thus, a system is represented by an *ordered, attributed tree*, called **project tree**, in which each node represents an **entity** and the parent-children relation of the nodes represents the containment relation between them. Figure 1a) shows an example of the project tree, the coarse-grained and fine-grained entities are illustrated with round and normal rectangles, respectively.

The properties of the entities and the structure of the project tree are represented by the *attributes* of the nodes. Among the attributes, `Id` is a unique and persistent identification of the corresponding entity. `Parent`, `Right`, and `Children`

Table 1. Tree Edit Operations

Operation	Description
Insert(u, v, k)	Insert u as the k^{th} child of v
Move(u, v, k)	Move u as the k^{th} child of v
Delete(u)	Delete u and insert its children as the new children of its parent node
Update(u, a, x)	Change the attribute a of u with new value x

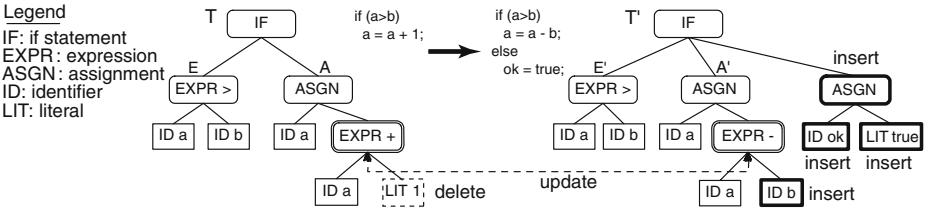

Fig. 2. Edit Operations on AST

respectively refer to the parent node, the right sibling node, and the ordered set of children nodes of the entity. `Type` is the type of the entity (e.g. *Class*, *Method*, etc). `Content` is the attribute representing the associated content of the entity. For example, if the entity is a *variable*, its `Content` contains the variable's name.

Change, Operation, and Version. The system and its entities evolve over time. In OperV, the **change** to a system is represented as tree edit **operations** on the project tree. Table 1 shows the list of operations including inserting, deleting, moving, and updating an attribute of a node (e.g. `Content`). Figures 1a) and b) illustrate such a change with corresponding operations. Figure 2 illustrates the similar operations on fine-grained entities in a program.

An entity is considered to be *changed* if and only if that entity or one of its descendants is affected by the operations of the changes. In other words, if any node of the sub-tree is affected by an operation, the entity corresponding to its root node is considered to be changed. A change of an entity is represented as the operations on the nodes of the corresponding sub-tree. The states of an entity between changes are called its **versions** and the changes are called **deltas**. In OperV, a delta is represented as a sequence of operations, i.e. an editing **script**.

Version Relation and Version Graph. The versions of an entity (including entire project tree) have the following relations:

1) *revision-of*: v is a revision of u if v is modified from u as in a sequential line of evolution.

2) *variant-of (branch-of)*: v' is a variant of v if v' is interchangeable and differ with v in some aspects of function, design, or implementation.

3) *merge-from*: w is a merge from v if w is created by merging v with another version v'. The merge of versions is used in the situation that different changes made to the same version need to be combined.

The versions of an entity and their relations are represented via its **version graph**, in which the nodes represent versions and the edges represent the changes between the versions. Figure 1c) shows an example of a version graph.

2.2 Problems

OperV needs to provide the algorithms for the following problems:

1. **Storage:** storing the changes to a system, i.e. all states and changes to the project tree and entities.
2. **Retrieval:** retrieving a version of a project and that of any entity.
3. **Differencing:** reporting the changes between any two versions of an entity.
4. **Merging:** merging two versions of an entity into a single one.

Because the operation-based program merging techniques have been well studied in the literature [27–29], we will describe only the algorithmic solutions for the first three problems in the next sections.

3 Mapping Entities on Project Tree

Because OperV is based on project versioning, any change(s) to one or multiple entities committed to the repository will create a new version of entire project. Only the changes and states (i.e. the version graph) of the project, however, are stored. The version graph of other entities will be re-constructed and processed on demand, using the stored version graph of the project tree. Therefore, the basis task to solve the aforementioned problems is to compute the changes of the project tree. This is in fact the *tree editing problem*, i.e. the problem of mapping the nodes of two project trees representing two consecutive versions of the system and deriving an edit script that transforms one tree into the other.

Since the project tree is huge, optimal tree editing algorithms are too computationally expensive. In [19], we developed Treed, a tree editing algorithm to compute an editing script for two ASTs representing for any two cloned fragments. In this paper, we generalize it to work for two project trees. In general, the algorithm has two steps. First, it determines a one-to-one mapping between the nodes of two project trees, in which any two mapped nodes represent two versions of the same entity and unmapped nodes represent deleted/inserted entities. Then, it derives the editing operations based on the recovered mapping.

This section presents the mapping step. Section 4 discusses how edit scripts are derived and used to solve the problems of storage, retrieval, and differencing.

3.1 Mapping Strategy

In Treed, the mapping step is designed based on the following observations:

O1) Leaf nodes of ASTs belonging to unchanged text regions are unchanged.

O2) If two nodes u and v are mapped, their ancestors and descendants are likely to be mapped.

O3) Two versions of a compound entity generally have similar structures. Thus, if the sub-tree rooted at u is highly similar to the sub-tree rooted at v, then u and v are likely to be mapped.

To measure the structural similarity of the (sub)trees, Treed uses Exas [1] characteristic vectors, which are shown in our previous work to be efficient and accurate in capturing the structure of trees and graphs. Using Exas, each tree is assigned an occurrence-counting vector of its structural features, such as the label sequences of its paths. For any two trees with two vectors x and y, their structural similarity is defined as $1 - \frac{2\|x-y\|}{\|x\|+\|y\|}$. That is, the smaller their vector distance is, the more similar they are. Larger trees (i.e. having large vectors) are allowed to have a larger distance within the same level of similarity. More details on Exas could be found in [1].

We customize Treed to work on the project tree with the following divide-and-conquer principle. That is, instead of processing the whole project tree, we divide it into two levels: coarse-grained and fine-grained levels. The mapping will be first applied for the parts of project trees containing only coarse-grained entities. Then, each pair of mapped coarse-grained entities is then processed independently to map their finer-grained entities. In practice, the coarse-grained entities, i.e. compilation units and packages, are generally stored as source files and directories. Therefore, this strategy would facilitate the storage of both coarse-grained and fine-grained entities in a conventional file system.

Nevertheless, two mapping processes for coarse-grained and fine-grained entities have the same phases: 1) initial mapping, 2) bottom-up mapping, and 3) top-down mapping. First, OperV initially maps as many tree nodes at the lowest level as possible (file/directory nodes at the coarse granularity level and leaf nodes at the fine granularity level). Then using such initial mapping, OperV maps the nodes at higher levels in a bottom-up manner, using the above observations to find the candidates and to choose mapped nodes based on their structural similarity. After going up to the root node, the algorithm goes top-down, trying to map the unmapped descendants of the mapped nodes, based on the similarity of their structure or contents.

3.2 Initial Mapping

Initial mapping for coarse-grained entities is based on name and location. Two coarse-grained entities (e.g. file, directory) having the same name and the same location in the tree will be mapped, i.e. be considered as the two versions of the same entity. For finer-grained entities, the initial mapping is much more complex because OperV does not directly store them (due to the huge size of the project tree). To map the finer-grained entities in two versions of a compilation unit, the corresponding source files are first parsed into ASTs. Then, the initial mapping is applied for the leaf nodes of those ASTs based on observation O1 in Section 3.1. This observation suggests the use of text-based differencing to divide the text un-parsed from an AST into the changed and unchanged text segments and to map leaf nodes into only the nodes in the corresponding segments. Therefore, the initial mapping for ASTs works in the two following steps:

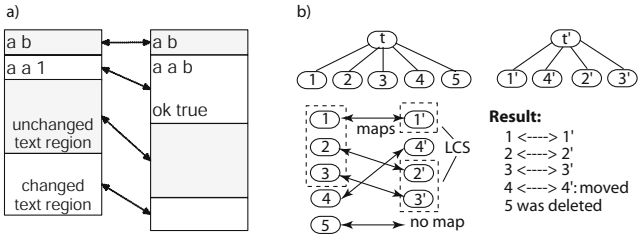

Fig. 3. Initial Mapping and Top-Down Mapping

Map Leaf Nodes of Unchanged Text Segments. First, two ASTs are unparsed into text-based versions. (Treed uses the unparsed text instead of the original text to discard the differences in formatting and other cosmetic changes). Then, text-based differencing, with the longest common subsequence (LCS) algorithm, is applied to compare two sequences of text lines in order to detect and align the unchanged text lines. The alignment of unchanged text lines will partition the text lines into (un)changed segments as in Figure 3a). If `Content` of a leaf node belongs to more than one segment, those segments will be joined. The joined segment will be considered as changed if it is joined from a changed segment. With this process, each leaf node will belong to only one segment.

Then, Treed traverses two trees, exports their sequences of leaf nodes, and marks the leaf nodes belonging to unchanged segments as "unchanged". Such unchanged leaf nodes in two sequences are mapped one-by-one in order. In Figure 2, two sequences of leaf nodes are $[a, b, a, a, 1]$ and $[a, b, a, a, b, ok, true]$. Because the first two nodes of those sequences belong to an unchanged text line (`if (a > b)`), they are mapped one to one: $a \rightarrow a, b \rightarrow b$.

Map Leaf Nodes of Changed Text Segments. Next, Treed maps the leaf nodes belonging to the changed text segments. Such segments contain *all changed leaf nodes* and might contain *unchanged leaf nodes*. For example, in Figure 3, two sequences of nodes $[a, a, 1]$ and $[a, a, b, ok, true]$ correspond to changed text segments. However, the first node a is unchanged in the statement `a = a + 1`.

To find the mapped nodes, for each pair of the aligned segments of changed lines in the two versions, Treed finds the LCS between the corresponding sequences of leaf nodes. Two nodes are considered as matched if they have the same `Type` and `Content`. The matched nodes of resulting subsequences are mapped to each other as unchanged leaf nodes. In the above sequences, Treed finds the common subsequences $[a, a]$ and maps the corresponding nodes $[a, a]$.

3.3 Bottom-Up Mapping

Treed maps the inner nodes in the bottom-up manner as follows. If an inner node u has a descendant u_1 mapped to v_1, u will be compared to any *ancestor*

of v_1. Then, u will be mapped to a candidate node v if the subtrees rooted at u and v are sufficiently similar in structure and type (measured via Exas method, see Section 3.1). If no such v exists, u is considered as unmapped. For example, In Figure 2, both E' and T' contain the mapped nodes to the nodes in E. However, because E is identical to E', they are mapped to each other. Similarly, A is mapped to A', although they are not identical in structure.

3.4 Top-Down Mapping

The bottom-up process might be unable to map some nodes, such as the relocated nodes and renamed identifier ones. Thus, after bottom-up mapping, Treed examines the two trees in the top-down fashion and maps those not yet mapped nodes based on the already mapped nodes. Given a pair of mapped nodes u and v from the bottom-up pass, Treed proceeds the mapping between their descendant nodes by the following mechanism:

Firstly, Treed performs a greedy algorithm to find additional mapped nodes between the children nodes of u and v. The already mapped nodes are kept. If unmapped children nodes are inner ones, their descendants are compared based on Exas structural similarity as in the previous step. If they are unmapped leaf nodes, Treed computes their similarity based on their attributes. For AST leaf nodes, since their `Contents` are generally identifiers and literals, such attributes are first separated as the sequences of words, using well-known naming conventions such as Hungarian or Camel. For example, "getFileName" is separated into "get", "file", and "name". The similarity of two words is computed via the Levenshtein distance, a string-based similarity measure.

After the children nodes are mapped, the LCS algorithm is run on those two sequences of mapped nodes. The mapped nodes not belonging to the resulting LCS are considered to be moved, such as node 4 in Figure 3.

3.5 Complexity Analysis

Given two trees T_1 and T_2, let $|T|$ be the number of nodes and $n = max(|T_1|,|T_2|)$. The above mapping algorithm contains the following sequential steps. First, building the text lines from the leaves of the trees takes $O(|T_1| + |T_2|)$, which is $O(n)$. Then, the unchanged lines are located using the longest common subsequence algorithm between two lists of lines. Thus the complexity of this step is $O(m_1 m_2) < O(n^2)$, where m_i is the number of lines built from tree T_i, respectively. Mapping the leaf nodes on those (un)changed lines takes at most $O(n^2)$ because of the use of the longest common subsequence algorithm between two lists of nodes. Computing characteristic vectors for all AST nodes is done incrementally from the leaf nodes to the root node, therefore it takes $O(n)$ [1]. Note that the accessing and comparing time of those vectors do not depend on the size of the tree since the features are extracted using n-paths in Exas, in which the lengths of vectors depend only on the number of AST nodes' types and the maximum size of the n-paths [1].

In the bottom-up mapping, each node v in one tree has at most $l_v h$ candidate nodes to map in the other tree, where l_v is the number of children of v and h is the maximum height of T_1 and T_2. Therefore, the complexity of this step is $O(\sum_{v_1 \in T_1} l_{v_1} h + \sum_{v_2 \in T_2} l_{v_2} h) = O(h \sum_{v_1 \in T_1} l_{v_1} + h \sum_{v_2 \in T_2} l_{v_2}) = O(h|T_1| + h|T_2|) = O(hn)$. In the top-down mapping, for each pair of mapped nodes v_1 and v_2, the matching between the two lists of their children can be done with $O(l_{v_1} l_{v_2})$. Since a node in one tree is mapped to only one node in the other tree, the complexity of this step is $O(\sum_{v_1 \in T_1} l_{v_1} l_{v_2}) = O(\sum_{v_1 \in T_1} l_{v_1} \sum_{v_2 \in T_2} l_{v_2}) = O(|T_1||T_2|) = O(n^2)$. Finally, traversing the trees to derive the editing script is in $O(n)$. In total, the complexity of the mapping algorithm is $O(n^2)$.

4 Differencing, Storage, and Retrieval

4.1 Deriving Editing Script from Mapping

After having the mapping for all nodes of the project tree, Treed could derive the edit operations for each node. For example, if a node is not mapped into any node in the new (old) version, it is *deleted* (*inserted*). If it is mapped but has different location with the mapped node, it is considered to be *moved*. If its attributes change, it is considered to be *updated*. Then, the editing script for the project tree could be generated via a top-down traversal. At each node, OperV exports the `Update`, `Move`, `Delete`, and `Insert` operations on its children, and then, derives *recursively* the editing script for its mapped children.

Treed is able to find the changes not only to fine-grained entities but also to coarse-grained ones. For example, by top-down mapping from the root node of the project tree, Treed could find two files having different names but identical structures, and determine them to be two versions of a renamed file. Note that due to the divide-and-conquer technique which separates between processing coarse-grained and finer-grained entities, Treed does not detect the moving operations across coarse-grained entities.

4.2 Storage

OperV uses the *backward delta* mechanism [16] to store the version graph of the system. That is, only the latest versions of all the branches in the version graph (i.e. sink nodes), together with all editing scripts (i.e. edges of the version graph) are stored. Note that, the individual changes of other entities are not stored. Because of the backward delta storage, an editing script is the script that transforms the later version of the project tree into an earlier one.

As a new version v_{n+1} of the project is *committed* to the repository, OperV first maps and computes the difference of v_{n+1} to the current version v_n. It stores the computed script, and then replaces v_n by v_{n+1} as the latest version. Along with the editing script, it stores also the set of changed, coarse-grained entities (called a *change set*) for efficiency of future retrieval tasks.

4.3 Retrieval

The retrieval requests could 1) check out a version of the entire project; 2) retrieve a version of an entity; or 3) retrieve the history (i.e. version graph) of an entity.

Type 1: From the stored (latest) version of the project, OperV will gradually apply the backward deltas as the editing scripts on the current project tree to re-construct the previous versions. In other words, OperV re-constructs and traverses the version graph of the project from its sink nodes (i.e. latest versions) and edges (i.e. deltas) to the desired version.

Types 2 and 3: OperV re-constructs the version graph of a requested entity in a similar way. The process is as follows. First, OperV extracts the latest version v of the requested entity e from the project tree. Then, OperV gradually processes the backward deltas until a requested version. For each delta d, OperV checks whether e was changed, i.e. affected by d. If e is a coarse-grained entity, OperV checks the corresponding change set (i.e. the set of changed, coarse-grained entities), which was stored along with d. If e is a fine-grained entity, OperV checks whether d contains any operation on e or a descendant of e.

Then, if e is unchanged, OperV continues processing the next delta. Otherwise, the operations of d affecting on e and its descendants will be applied on v. If such operation involves another entity u, e.g. an operation moving a variable node from a statement v to another statement u, the current version of u will be requested to be extracted. After all operations are applied on v, OperV has the previous version of e, and the next delta will be processed using that version. Note that, when extracting or editing a version of an entity, OperV builds only the part of the project tree related to that entity. For example, if the entity is a method, OperV parses only the source file containing that method, rather than parsing all files to build the entire project tree. This on-demand scheme enables an efficient version retrieval for any fine-grained entities.

4.4 Tool Implementation

We have implemented a prototype of OperV with all aforementioned algorithms. In the implementation, logical entities are mapped to physical files in a file system: projects and packages are mapped to directories, classes are mapped to files. Finer-grained entities (corresponding to the AST nodes of a class) are parsed from source files. The tool provides the following functions:

1. Check-out the entire system at a version (this involves version retrieval),
2. Commit a new version of the entire system (this task involves version identification, differencing, and storage),
3. Display a version of a fine-grained entity (this involves version retrieval),
4. Display the version log of an entity in term of its version graphs, and
5. Compute and display the change of any two versions of an entity (i.e. differencing). If the entity is coarse-grained, the change is displayed in term of the file/directory operations. For a fine-grained entity in an AST, the change is

shown as AST operations. This function is done via mapping and differencing algorithms on the sub-trees corresponding to two versions of that entity.

5 Empirical Evaluation

5.1 Scalability and Efficiency in Differencing

To evaluate our tree editing algorithm, we compared it to one of the state-of-the-art optimal algorithms proposed by Zhang and Shasha [21, 23]. Two algorithms were run on a set of changed source files in a range of revisions of subject systems (Table 2). Then, the lengths of the output edit scripts and running time of two algorithms were compared. All experiments were carried out on a Windows XP computer with Intel Core 2 Duo 2Ghz CPU, 3GB RAM, and 80GB HDD.

Table 2 shows the results. Column *Revision* is the range of revisions of subject systems in our experiments. Column *File* is the number of modified files under processing (the added and deleted files were not processed since they are of trivial cases). Columns *Size* and *Size** show their average sizes in term of AST nodes before and after the changes, respectively. In this experiment, we set the limit for the file size of less than 2,000 nodes due to the scalability problem in the optimal algorithm. Columns L_{OperV} and L_{opt} show the average lengths of the output scripts of our algorithm and the optimal one, respectively. The columns labelled with Δ show the percentages of the cases in which the differences in the lengths of scripts are zero, between 1 and 3, and more than 3 operations, respectively. Columns T_{OperV} and T_{opt} show the average running time for each pair of files in two algorithms.

The result implies that, our algorithm outputs the scripts with the optimal lengths in 70-85% cases. In half of the remaining cases, the scripts are a bit longer with a few operations. Such little sacrifice in accuracy brings large saving on time efficiency. It could be seen that, the saving ratios are from 50 - 500 times.

5.2 Scalability in Storing

Table 3 shows the result of our experiment on the scalability of OperV the storing task. In this experiment, since only OperV was operated, we did not set the limit for file sizes, and we ran on wider ranges of revisions. For each subject system, we stored the initial version, continuously committed the successive versions and measured the total processing time (including parsing, differencing, and storing)

Table 2. Comparison to Optimal Algorithm on Running Time and Script Length

Project	Revision	File	Size	Size*	L_{OperV}	L_{opt}	Δ=0	Δ=1-3	Δ >3	$T_{OperV}(s)$	$T_{opt}(s)$
ArgoUML	5000-5100	135	422	417	21	20	84%	6%	10%	0.19	13.1
Columba	200-300	208	513	516	45	43	73%	14%	13%	0.11	15.3
GEclipse	1800-2000	138	624	647	53	50	69%	10%	21%	0.13	25.3
jEdit	4000-4100	158	921	931	39	37	70%	13%	17%	0.13	50.3
ZK	5000-5400	284	272	277	16	15	85%	8%	7%	0.13	20.7

Table 3. Time and Space Costs

Project	Revision	File	MaxSize	Time (s)	Project(MB)	Delta(MB)	Delta/version
ArgoUML	5000-6000	1297	19044	0.7	14.7	8.1	0.06%
Columba	200-300	1549	4081	0.4	12.1	1.2	0.10%
GEclipse	1000-2000	779	10563	0.2	9.2	12.6	0.14%
jEdit	4000-5000	394	24660	1.7	7.8	16.1	0.21%
ZK	5000-6000	991	8287	0.3	8.3	2.9	0.03%

[Columba v252-v253] LocalCalendarStoreTest.java

Event model = new Event();	*String uuid = new UUIDGenerator().newUUID();*
model.setSummary("summary");	**Event model = new Event(uuid);**
model.setDescription("description");	model.setSummary("summary");
String uuid = new UUIDGenerator().newUUID();	model.setDescription("description");
storage.add(model);	storage.add(model);

Fig. 4. Example of Change in Object Usage

[Columba v213-v214] TittleBar.java

	if (active) {
	...
g2.setPaint(firstHalf);	g2.setPaint(firstHalf);
g2.fillRect(0, 0, w, h);	g2.fillRect(0, 0, w, h);
	} else {
	g2.setColor(fillColor);
	g2.fillRect(0,0,w,h);

Fig. 5. Example of Change in Control Flow

and storage space. In the table, Column *Time* shows the average processing time for each revision. Columns *Project* and *Delta* show the total space for the initial version and all deltas, respectively. Column *MaxSize* shows the maximum file size in term of AST nodes on each system.

It could be seen that OperV is highly scalable. Processing time of each version generally costs less than 1 second on large-scale systems which could contain thousands of files with the sizes up to tens of thousands of nodes. For `jEdit`, although the number of files is less than in the other projects, the sizes of its files are generally larger. This makes processing time for `jEdit` a bit longer because OperV's differencing algorithm has a quadratic complexity on file size.

The storage cost is reasonable. For example, at revision 4000, `jEdit` has size of 7.8MB. After checking-in 1,000 versions, OperV needs extra 16.1MB to store 1,000 deltas. Thus, on average, each delta costs about 0.21% of the project size.

5.3 Anecdotal Evidence of Usefulness

Figure 4 shows an interesting example that OperV reported when running on Columba v252-v253. In this case, a developer moved the initialization statement for `uuid`, and used `uuid` as a parameter of a constructor call for an `Event` object. This example shows that OperV is able to detect the change in the usage of

objects, which is one kind of semantic change. Thus, it provides more useful information for developers to understand code evolution than text-based changes.

OperV is also able to detect changes in a control flow. For example, in Figure 5, it was able to report the addition of branching to a block of code. Similarly, it found another case in which the statement `if (services.length > 1)` was changed to `if ((services != null) && (services.length > 1))`. It was able to report the insertion of that new control condition.

In another example in `WorkflowJobEditPart.java` of GEclipse v1803-v1804, OperV reported the renaming of two variables: `workflowJobNameFigure0` to `workflowJobNameFigure` and `workflowJobDescriptionFigure1` to `workflowJobDescriptionFigure` several times. This result suggests that OperV could be used as a foundation for the "renaming" refactoring recovery tools.

6 Related Work

The source code versioning and differencing approaches could be classified based on the abstraction level in the program representations of their tools. Generally, they could be classified into lexical (text-based), structural (often tree-based), syntactical (AST-based), or semantic approaches.

Traditional version control tools (CVS [3], RCS [4], SVN [5]) use the **text-based** differencing approaches (such as *text diff* [6]), which consider a program as a set of text lines and often use the longest common subsequence technique to calculate the inserted, deleted, and modified lines. Ldiff [7] was built upon text *diff* to track the changes to lines including line addition, removal, or moving. This type of approaches does not work well when developers care more about the changes in code structure. Moreover, the reported changes in term of text lines do not fit well with automated program analysis tools for software evolution.

Structural versioning and differencing tools assume that software documents are stored in a fine-grained and structural manner (e.g. XML). Algorithms for such structural differencing include [8–10]. POEM [11] provides tree-based version control for functions and classes [16]. The principles of the tree-based COOP/Orm framework [12] include the sharing of unchanged tree nodes among versions and change propagation. Unified Extensional Versioning Model [13] supports fine-grained versioning for a tree-structured document by composite/atomic nodes and links. Each atomic node is represented by a text file. In Coven [14], a versioned fragment could be entire method for C++ and Java programs, or text paragraph in LaTeX documents. In Ohst's model [15], fine-grained changes are managed within contexts of UML tools and design transactions. Maletic and Collard [39] also used text `diff` to locate changed entities in an XML representation of code. However, their goal was not to derive edit scripts.

In general, those approaches heavily rely on the total versioning scheme with the technical drawback of the version proliferation problem [16] as described earlier. In contrast, the generic tree-based differencing algorithms [20–22] are too computationally expensive for source code evolution because they consider two arbitrary trees as the inputs, rather than two revisions of the same program [17].

OperV uses project versioning to avoid version proliferation and Treed makes it efficient due to its specialization toward source code with the divide-and-conquer strategy. SVN [5] also uses project versioning but with a text-based approach.

Our tree edit algorithm, Treed, belongs to the **syntactical** category. Treed is similar in nature to ChangeDistiller [17], a tree-based differencing and code change extracting tool. However, the first key departure point is the *divide-and-conquer* approach that helps Treed reduce the complexity of mapping the corresponding nodes in two versions. It takes into account the fact that the alignment of unchanged text regions between two revisions will partition the texts of two revisions into smaller and corresponding segments. Thus, it could apply its mapping procedure on smaller segments. ChangeDistiller does not use the alignment of un-changed texts in two versions for divide-and-conquer. Another key difference is the use of Exas [1], a vector-based structure similarity measure for trees. Exas [1] has been shown to perform better than the subtree similarity measure used in Chawathe *et al.* [18], which was also used in ChangeDistiller. Exas' characteristic features capture the structure via the paths in a tree, while Chawathe's approach relies only on the number of matched nodes in two trees.

Other syntactical, tree-based differencing approaches were also used in program differencing and merging tools [24, 25]. Their goals were not to detect editing operations. The tree matching techniques for AST were used in tree-based clone detection tools as well [19, 30–32]. Those algorithms include suffix tree [30], maximum common subtree isomorphism [31], or dice coefficient [32].

Our model is the first to combine the strength of both project versioning and operation-based approaches for fine-grained versioning. Operation-based model represents the changes between versions as explicit operations. However, similar to other syntactical approaches, OperV requires software artifacts to be parseable. Robbes and Lanza [26] advocate for an approach to analyze software evolution using semantic changes. Our work is similar to their analysis tool with the use of tree edit operations for change representation. They define semantic changes as well. The key difference is that OperV is a complete version model that combines versioning for entire project with the operation-based approach. Their evolution analysis approach will benefit from versioning tools built from OperV. Operation-based approach had also been used in software merging [27–29]. Ekman and Asklund [29] presented a refactoring-aware versioning system.

Semantic-based program versioning and differencing approaches often rely on graph-based techniques. They represent the behavior of a program via a graph (e.g. program dependence graph, program slice as in [33], control flow graph as in [34], semantic-graph as in [35]), and detect the changed entities in the graph. Those approaches provide more semantic information than OperV, however, they are more computationally expensive.

There exists version control models that are change-based ([36–38]). That is, in change-based models, the changes are first-class entities with unique identifiers that are used to compose the system at different time. Comparing to change-based models, OperV is still state-based because the revisions have identifiers and the changes (as operations) are computed via our tree edit script algorithm.

7 Conclusions

This paper presents OperV, a novel operation-based, fine-grained version control model and tool for source code. The key idea is the combination between project versioning and operation-based approaches. In OperV, a software system is represented by a project tree whose nodes represent all program entities at both coarse-grained and fine-grained levels. The changes of the system are represented via edit operations on the tree. OperV also provides the algorithms for storing, retrieving, and differencing between the versions of such entities. These algorithms are designed based on several heuristics to improve scalability and efficiency. The empirical evaluation of the model showed that OperV is scalable, efficient, and could be useful for developers in understanding of code evolution.

Acknowledgment. We thank reviewers for their very insightful feedbacks. The first author was partially funded by a grant from Vietnam Education Foundation.

References

1. Nguyen, H.A., Nguyen, T.T., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N.: Accurate and efficient structural characteristic feature extraction for clone detection. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 440–455. Springer, Heidelberg (2009)
2. Nguyen, T.N., Munson, E.V., Boyland, J.T., Thao, C.: An Infrastructure for Development of Multi-level, Object-Oriented Config. Management Services. In: ICSE 2005, pp. 215–224. ACM, New York (2005)
3. Morse, T.: CVS. Linux Journal 1996(21es), 3 (1996)
4. Tichy, W.: RCS - A System for Version Control. Software - Practice and Experience 15(7), 637–654 (1985)
5. Subversion.tigris.org., `http://subversion.tigris.org/`
6. Hunt, J.W., Szymanski, T.G.: A fast algorithm for computing longest common subsequences. Communication of ACM 20(5), 350–353 (1977)
7. Canfora, G., Cerulo, L., Di Penta, M.: Ldiff: An enhanced line differencing tool. In: ICSE 2009, pp. 595–598. IEEE CS, Los Alamitos (2009)
8. Myers, E.W.: An O(nd) Difference Algorithm and Its Variations. Algorithmica 1, 251–266 (1986)
9. Tichy, W.F.: The string-to-string correction problem with block moves. ACM Trans. Comput. Syst. 2(4), 309–321 (1984)
10. Wang, Y., DeWitt, D.J., Cai, J.Y.: X-Diff: An Effective Change Detection Algorithm for XML Documents. In: ICDE 2003, pp. 519–530. IEEE CS, Los Alamitos (2003)
11. Lin, Y., Reiss, S.: Configuration management with logical structures. In: ICSE 1996, pp. 298–307 (1996)
12. Magnusson, B., Asklund, U.: Fine-grained revision control of Configurations in COOP/Orm. In: 6th Software Configuration Management Workshop (SCM-6), pp. 31–47. Springer, Heidelberg (1996)
13. Asklund, U., Bendix, L., Christensen, H., Magnusson, B.: The Unified Extensional Versioning Model. In: 9th Software Configuration Management Workshop (SCM-9). Springer, Heidelberg (1999)

14. Chu-Carroll, M.C., Wright, J., Shields, D.: Supporting aggregation in fine grained software configuration management. In: FSE 2002, pp. 99–108. ACM Press, New York (2002)
15. Ohst, D., Kelter, U.: A fine-grained version and configuration model in analysis and design. In: ICSM 2002. IEEE CS, Los Alamitos (2002)
16. Conradi, R., Westfechtel, B.: Version models for software configuration management. ACM Computing Surveys (CSUR) 30(2), 232–282 (1998)
17. Fluri, B., Wuersch, M., Pinzger, M., Gall, H.: Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Transactions on Software Engineering 33(11), 725–743 (2007)
18. Chawathe, S.S., Rajaraman, A., Garcia-Molina, H., Widom, J.: Change detection in hierarchically structured information. In: SIGMOD 1996, pp. 493–504. ACM, New York (1996)
19. Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N.: Clone-aware Configuration Management. In: The 24th ACM/IEEE International Conference on Automated Software Engineering (ASE 2009). IEEE CS Press, Los Alamitos (2009)
20. Bille, P.: A survey on tree edit distance and related problems. Theor. Comput. Sci. 337(1-3), 217–239 (2005)
21. Zhang, K.: Algorithms for the constrained editing distance between ordered labeled trees and related problems. Pattern Recognition 28, 463–474 (1995)
22. Selkow, S.: The tree-to-tree editing problem. Info. Processing Letters 6(6), 184–186 (1977)
23. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. SIAM Journal of Computing 18, 1245–1262 (1989)
24. Asklund, U.: Identifying conflicts during structural merge. In: Proceedings of the Nordic Workshop on Programming Environment Research (1994)
25. Westfechtel, B.: Structure-oriented merging of revisions of software documents. In: Proceedings of Workshop on Software Configuration Management, pp. 68–79. ACM, New York (1991)
26. Robbes, R., Lanza, M., Lungu, M.: An Approach to Software Evolution Based on Semantic Change. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 27–41. Springer, Heidelberg (2007)
27. Lippe, E., van Oosterom, N.: Operation-based merging. In: SDE-5, pp. 78–87. ACM Press, New York (1992)
28. Edwards, W.: Flexible Conflict Detection and Management in Collaborative Applications. In: Proceedings of UIST (1997)
29. Ekman, T., Asklund, U.: Refactoring-aware versioning in Eclipse. Electronic Notes in Theoretical Computer Science 107, 57–69 (2004)
30. Gode, N., Koschke, R.: Incremental Clone Detection. In: CSMR 2009, pp. 219–228. IEEE CS, Los Alamitos (2009)
31. Sager, T., Bernstein, A., Pinzger, M., Kiefer, C.: Detecting similar Java classes using tree algorithms. In: MSR 2006, pp. 65–71. ACM, New York (2006)
32. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: ICSM 1998, pp. 368–377. IEEE CS, Los Alamitos (1998)
33. Horwitz, S.: Identifying the semantic and textual differences between two versions of a program. In: PLDI 1990, pp. 234–245. ACM, New York (1990)
34. Apiwattanapong, T., Orso, A., Harrold, M.J.: Jdiff: A differencing technique and tool for object-oriented programs. Automated Software Engg. 14(1), 3–36 (2007)

35. Raghavan, S., Rohana, R., Leon, D., Podgurski, A., Augustine, V.: Dex: A semantic-graph differencing tool for studying changes in large code bases. In: ICSM 2004, pp. 188–197. IEEE, Los Alamitos (2004)
36. Lie, A., Conradi, R., Didriksen, T., Karlsson, E., Hallsteinsen, S., Holager, P.: Change oriented versioning. In: Proceedings of the Second European Software Engineering Conference (1989)
37. Cronk, R.: Tributaries and deltas. BYTE, pp. 177–186 (January 1992)
38. Zelller, A., Snelting, G.: Unified versioning through feature logic. ACM Transaction on Software Engineering and Methodology 6, 397–440 (1997)
39. Maletic, J., Collard, M.: Supporting Source Code Difference Analysis. In: ICSM 2004. IEEE CS, Los Alamitos (2004)