

Clone-aware Configuration Management

Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, Tien N. Nguyen
 Electrical and Computer Engineering Department
 Iowa State University
 {tung,hoan,nampham,jafar,tien}@iastate.edu



Abstract—Recent research results show several benefits of the management of code clones. In this paper, we introduce *Clever*, a novel clone-aware software configuration management (SCM) system. In addition to traditional SCM functionality, *Clever* provides clone management support, including clone detection and update, clone change management, clone consistency validating, clone synchronizing, and clone merging. *Clever* represents source code and clones as (sub)trees in Abstract Syntax Trees (ASTs), measures code similarity based on structural characteristic vectors, and describes code changes as tree editing scripts. The key techniques of *Clever* include the algorithms to compute tree editing scripts; to detect and update code clones and their groups; and to analyze the changes of cloned code to validate their consistency and recommend the relevant synchronization. Our empirical study on many real-world programs shows that *Clever* is highly efficient and accurate in clone detection and updating, and provides useful analysis of clone changes.

1 INTRODUCTION

Code clones are exactly matched or similar portions of code that are often created by the copy-and-paste programming practice. Classical approaches considered code clones to be harmful, thus, emphasized on the detection and removal of clones [6], [37]. However, recent research has shown more benefits of code clone management during software evolution than removing them [11], [22], [23], [32].

During software development, source code is modified. As regular code, cloned code evolves as well. However, existing code clone management approaches are still ad-hoc, limited, and unsatisfactory, especially with changes to clones. Software configuration management (SCM) [13] area provides many well-established tools for managing the changes to source code in software systems with useful version control and collaboration supports. Thus, code clone management in evolving software should be incorporated into an SCM system. In other words, an SCM tool should be *clone-aware*.

The integration of clone management support into an SCM system creates several benefits. Firstly, the management and tracking of changes to clones and clone groups can take advantage of change management supports without requiring a full retrieval of individual versions. Secondly, clone group management would be more time efficient and complete because reported changes from an SCM tool will help in updating the clone results for a new version without complete re-detection. Re-detection is time-consuming for large-scale systems while changes might affect a small set of code clones.

Finally, collaborative and team development supports in SCM will facilitate the maintenance of consistent editing to cloned code from multiple developers. Recent study by Krinke [25] on several open-source systems showed that half of the changes to code clone groups are inconsistent changes.

Unfortunately, current SCM tools are not well-equipped with clone management supports. Text line-based change management approaches in existing SCM systems are *not* suitable for clone management because code clones are not necessarily identical. They often have slight modifications, thus, requiring an approach that can better capture the code semantics than the text-based approach in existing SCM tools. Describing changes to clones and clone groups in term of changed lines is clearly insufficient in supporting code understanding.

1.1 Clone-aware SCM functionality

In this paper, we introduce *Clever*, a novel *clone-aware software configuration management system*, which makes use of SCM functionality and provides additional supports for clone management. *Clever* was developed as an add-on to Subclipse/SVN, an Eclipse plug-in SCM tool. Let us first describe *Clever* from the users' perspective. In addition to traditional SCM functionality [13], *Clever* has the following clone-aware supports:

1. Detecting code clones and grouping them,
2. Updating clones and groups as source code changes,
3. Managing the changes to individual clones and groups,
4. Reporting clones/groups and their changes at any version,
5. Notifying developers on potential inconsistent modifications to the cloned code, and
6. Supporting consistent changes to clones and merging.

Clone Detection and Updating (tasks 1 and 2): First of all, a developer could use Eclipse to work on a software project. At any time, (s)he can start code clone detection on any version of the project. When (s)he checks in the code, if clone detection has never been initiated, *Clever* will perform the detection. It reads source files and extracts important features. The initial detection is launched using our algorithm on those fragments (Section 5). In addition to the normal check-in data, clone-related information is also stored for future updating.

When the developer checks out a version, along with the code, the clone information is also retrieved. At any time, if requested, *Clever* produces a clone report that describes the

clones and clone groups. Each *clone group* is reported as a set of cloned fragments and each cloned fragment is reported with its location. The developer is able to use Eclipse’s editor to make changes to source files. As a new version is checked into the repository, Clever will perform the clone updating process. It uses the reported changes from SVN and builds the sets of deleted, modified, and newly created fragments accordingly. Then, Clever executes its clone updating algorithm (Section 5). It also runs clone updating as requested and clone information of the previous detection is used for the update.

Clone Change Management (tasks 3 and 4): The developer can use the *textual differencing* tool of SVN to compare two versions of a program. (S)he can also use Clever’s *clone differencing* functionality to display the changes in term of editing operations between the versions. This function can be used to show how a clone has been modified from the previous version, which could help him/her understand the changes and consistently modify other cloned code.

At any time, the developer can invoke the change report for clone groups. The report shows the evolution of a clone group in consecutive versions. The newly created, modified, disappearing, and un-changed clone groups will also be shown. As selecting a group, (s)he could see its clone members and which clone fragments have been modified, created, or deleted.

Clone Consistency Validating (task 5): This clone-aware feature is similar in spirit to the *conflicting change detection* in traditional SCM systems when two developers made changes to the *same* file. For code clones, the changes could be made to *two* cloned fragments. For example, A is a clone of a fragment B . Assume that A has been slightly modified into A' by developer 1 to fix a bug. A' is then checked into the repository. In this case, Clever checks if those changes could potentially cause inconsistency. If true, it will bring up the cloned fragments of A including B to the developer’s attention along with editing operations from A to A' for his reference.

Clone Synchronizing (task 6): Those recommended operations could be applied into B one-by-one and the developer can verify the correctness of the result. If (s)he decides that the changes from A to A' do not apply to B , then B can be kept the same. Assume that later, developer 2 checks out and modifies B into B' . When B' is checked in, Clever will perform consistency validation on the changes from B to B' against those from A to A' . If the changes are not consistent, they will be brought to the developer for verification as well. Moreover, another type of consistency validation is provided in Clever when a user simply copies a code fragment and renames identifier(s). Clever is able to recommend consistent renaming operations for the applicable identifiers.

Clone Merging (task 6): This clone-aware feature corresponds to the *merge* function in traditional SCM. For example, if the changes to B are consistent with those to A , Clever will perform a recommendation for automatic incorporation of the changes to A into B' . If inconsistency occurs, the developer would have to manually update B' for consistency based on the reported changes from A to A' . The merging result B'' after human verification will be checked into the repository.

1.2 Approach Overview

Let us give an overview of our approach to build such clone-aware SCM system. In addition to reusing SVN’s text line-based storage representation for code, Clever also views a program as an abstract syntax tree (AST). Any sufficiently large subtree of the program’s AST is called a fragment and considered as a potential clone. Two fragments, i.e. two AST’s subtrees, are considered as clones, and called a clone pair, if they are sufficiently similar in structure. Clever measures their structural similarity by the distance of their structural characteristic vectors, extracted by using Exas method [33]. All detected clone pairs form a clone graph. The connected components in that graph are considered as clone groups.

Since Clever uses tree-based representation for source code and clones, a new *tree edit scripting algorithm*, called **Treed**, is developed to capture and represent their changes. Treed takes two text-based versions of an arbitrary fragment from SVN, parses them into two ASTs, and then computes a sequence of tree editing operations, i.e. a tree editing script, transforming one version to the other. Treed also identifies the matched and different nodes between two trees.

From such tree-based changes of the program, Clever derives the change sets of the fragments and updates the clones and groups accordingly. It also analyzes the changes to the cloned fragments to find potential inconsistent changes and recommends the relevant synchronization and merged results.

The main contributions of this paper include:

1. A novel clone-aware SCM system named Clever,
2. A new tree edit scripting algorithm,
3. An efficient algorithm for clone detection and updating,
4. A novel clone change analysis method that can detect inconsistent changes to the cloned code and recommend the relevant synchronization and merging, and
5. An empirical study that shows the benefits of Clever.

The next section describes Treed. Section 3 is about clone detection and updating. Section 4 discusses our clone change analysis method. Evaluation is given in Section 5. Related work is described in Section 6. Conclusions appear last.

2 TREE EDITING

2.1 Tree-based Code Representation

In Clever, an AST’s subtree T is an ordered, attributed tree and modeled as a set of nodes associated with five functions R , P , C , $type$, and val . $R(T)$ is the root node of the tree. For each node $u \in T$, $P(u)$ is its parent, $C(u, k)$ is its k th child, $type(u)$ is its AST node type, and $val(u)$ is its attribute value. val is generally used for identifiers, literals, and operators. For the AST nodes representing the control structures (e.g. `if`, `while`), their $vals$ are empty. Figure 1 shows an example. Leaf nodes are in rectangles. Inner nodes are in rounded rectangles.

2.2 Editing Operation and Editing Script

When the code changes, its corresponding AST is considered to be edited, i.e. transformed, into the tree representing the new code by an *editing script*, i.e. a sequence of tree editing operations. As the common tree edit approaches, Treed uses the following tree editing operations for an AST:

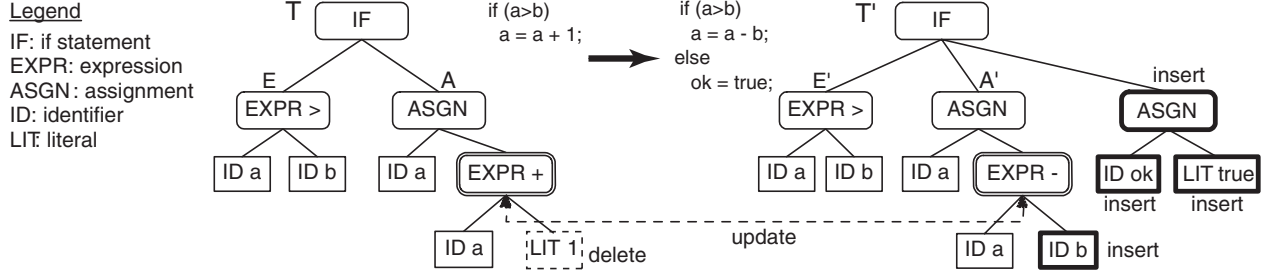


Fig. 1. Tree Editing Example: In the label of each node, its type is in capital font and its *val* (if exists) is in normal font.

$Update(u, x)$ changes the value $val(u)$ of node u into x .
 $Insert(u, v, k)$ inserts node u as the k th child of node v .
 $Delete(u)$ deletes node u , and inserts its children as the new children of its parent node.

$Move(u, v, k)$ moves the subtree rooted at node u to node v in which u becomes the k th child of node v .

In the example in Figure 1, an `if` statement was edited by modifying the `if` branch and adding an `else` branch. The two trees represent the two versions. An editing script consists of one *Delete* (dotted line box), one *Update* (double-line box), and four *Insert* operations (bold boxes). Other nodes (single-line boxes) are either unchanged or moved.

As we can see, an editing script could be used to describe the change to the code. For example, the nodes that are unaffected by the editing operations can be considered as unchanged, and the affected ones are the changed ones.

For any two trees, there will always exist at least one editing script (such as the script that deletes all nodes of the first tree and inserts all the nodes of the second). However, there might exist more than ones. Since such scripts give the same result, we could use the *optimal* script with the *minimum* number of operations to model developers' rational editing.

Definition 1 (Tree Editing Script Problem): Given two AST's (sub)trees T and T' , find the optimal editing script that transforms T into T' .

The number of operations of the optimal editing script for T and T' is generally referred to as their *editing distance*. This could be used to measure their similarity. That is, the shorter the editing distance is, the more similar the trees are [18]. Thus, finding the optimal editing script is helpful in both detecting cloned code and analyzing their changes.

However, finding the optimal editing script could be inefficient in large-scale projects since the numbers and the sizes of the trees needed to be processed are huge¹. Therefore, instead of developing such an exact algorithm, we alternatively develop a heuristic algorithm, called Treed (Tree Edit) to efficiently find one editing script.

2.3 Treed Algorithm

The task of Treed is to take two text-based versions of an arbitrary portion of code from an SCM tool, parse them into AST's subtrees, and to compute a tree editing script, as short as possible, that transforms the old version to the new one.

1. The subject systems in our experiments usually have several thousands of code fragments with the minimum size of 50 nodes.

2.3.1 Map Relation versus Editing Script

The key insight of Treed is that the nodes in T that are unchanged, updated (i.e. *vals* are changed), moved, or have no/few changes in its subtrees should be kept as many as possible, rather than deleting them and inserting new nodes, because the latter case creates a longer script. If a node u of T is kept, it must correspond to a node u' of T' . We consider them to be *mapped* to each other. In Figure 1, nodes E , A , and $[EXPR +]$ are mapped to E' , A' , and $[EXPR -]$, respectively.

Given an editing script Δ for T and T' , one could always determine all mapped nodes between them based on the (un)changed, moved, and updated nodes caused by Δ . That is, one could derive a *map relation* between the nodes of T and T' . The map relation is unique.

In the other direction, given a map relation \mathfrak{R} , one could always derive an editing script Δ because one could imply from the map \mathfrak{R} the editing operations that were applied on each node. For example, if a node $u \in T$ is mapped to a node $u' \in T'$ and their *val* attributes are different, then u is updated. If u or u' are unmapped, i.e. they are not mapped to any node in the other tree, they are deleted or inserted. The derived script Δ might not be unique or optimal. The more nodes can be mapped, the more concise the derived script is.

Based on that knowledge, Treed works in two steps. The first step is to determine the map relation \mathfrak{R} between the nodes of T and T' , with as many nodes being mapped as possible. Then, the next step is to derive the editing script Δ from \mathfrak{R} .

2.3.2 Find the Map Relation

Treed finds the map relation with the following observations:

1. If a leaf node belongs to (i.e. its *val* text sits in) an unchanged line of code (LOC), it is considered as unchanged. In general, the values, i.e. identifiers, literals, and operators, usually lie completely in a line. For example, Java syntax does not allow an identifier or a number to span across multiple lines. Extremely long strings lying in multiple lines are unusual. In Figure 1, the first line (containing the expression `if (a > b)`) is unchanged, thus, all leaf nodes of the expression E are unchanged.

2. Two inner nodes $t \in T$ and $t' \in T'$ should not be mapped if they do not have any two mapped descendant nodes in corresponding subtrees. Since there might exist more than one mapping candidates for each node, it should be mapped only to another node such that two respective subtrees are sufficiently similar.

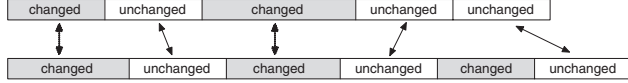


Fig. 2. Alignment of Unchanged Lines of Code

In Figure 1, we see that E should be mapped to E' since they contain mapped (unchanged) leaf nodes. E should not be mapped to A' since they have no mapped descendant. Of course, it should not be mapped to T' (i.e. the entire `if` statement). Although E and T' have the mapped descendants, E' is more *similar* to E than T' (will be explained later).

Those observations suggest that 1) the map relation of *unchanged leaf nodes* could be found based on *unchanged LOCs*; 2) the map relation should be calculated from *bottom up*, i.e. from leaf nodes up to the root, and 3) the map relation of inner nodes should be based on the similarity of their corresponding subtrees.

Mapping. Treed maps the nodes in the following steps:

1. Map Leaf Nodes of Unchanged LOCs. First, Treed uses the text line comparison feature in SVN to compare the text-based representation of two versions to detect the unchanged LOCs. After this, the alignment between unchanged lines will partition the text lines in two versions into (un)changed segments as in Figure 2.

Treed parses the versions into two trees T and T' , and marks the leaf nodes belonging to unchanged LOCs as “unchanged”. Then, it traverses T and T' in pre-order and returns two sequences of leaf nodes. Unchanged leaf nodes in such two sequences are mapped one-by-one in that order. In Figure 1, 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 LOC (`if (a > b)`), they are mapped one to one: $a \rightarrow a, b \rightarrow b$.

2. Map Leaf Nodes of Changed LOCs. The next step is to map leaf nodes belonging to changed LOCs. Segments of changed lines contain *all changed leaf nodes* and might contain also *unchanged leaf nodes* (e.g. a line might be just partially changed). For example, two sequences of nodes $[a, a, 1]$ and $[a, a, b, ok, true]$ correspond to changed text lines. However, the first node a is unchanged in $a = a + 1$.

To find mapped nodes, for each pair of aligned segments of changed lines in two versions, Treed finds the largest common subsequences between the corresponding sequences of leaf nodes. Two nodes are considered matched if they have the same *type* and *value*. The matched nodes of the resulting subsequences are mapped together as unchanged leaf nodes. In the above segments, Treed will find the common subsequences $[a, a]$ and mapped the corresponding nodes $[a, a]$.

3. Map Inner Nodes Bottom-Up. After mapping the leaf nodes, Treed maps the inner nodes bottom-up. If an inner node $t \in T$ has a descendant t_1 (inner or leaf node) and t_1 is mapped to t'_1 , t will be compared to any *ancestor* of t'_1 . Then, t will be mapped to a candidate node t' if the subtrees rooted at t and t' are sufficiently similar in structure and type. If no such t' exists, t is unmapped.

For example, both E' and T' contains mapped nodes to the

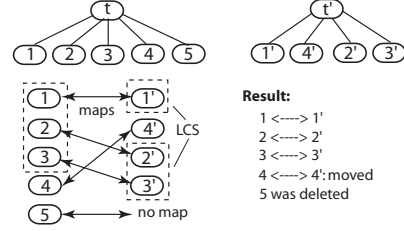


Fig. 3. Alignment of Nodes in Top-Down Pass

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.

Structural Similarity Measure. To measure the structural similarity of the trees, Treed uses Exas [33] 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, the more they are similar. 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 [33].

4. Map Nodes Top-Down and Derive Editing Operations.

The bottom-up process might be unable to map some nodes, such as the relocated nodes and renamed identifier nodes. Thus, after bottom-up mapping, Treed examines the two trees top-down and maps those not yet mapped nodes based on the already mapped nodes. Given a pair of mapped inner nodes t and t' from the bottom-up pass, Treed determines the mapping between their children nodes.

Firstly, Treed performs a greedy algorithm to find additional mapped nodes between the children nodes of t and t' . The already mapped nodes are kept. If an unmapped child node is an inner one, their descendants are compared based on Exas structural similarity as in the previous step. If it is an unmapped leaf node, Treed computes their similarity based on their *val* attributes. Since those attributes are generally identifiers and literals, they are first separated as 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, a largest common subsequence (LCS) algorithm is run on those two sequences. From the mappings, the corresponding operations are derived for nodes. For example, if a node is mapped to another node with a different *val*, it is considered updated. If they are at different locations, it is considered to be moved. The unmapped nodes are considered as deleted or inserted.

Figure 3 shows an example. t and t' are mapped from bottom-up pass. After running the greedy algorithm, Treed finds all mapped nodes except node 5. From the alignment after the LCS algorithm, Treed derives the mappings, and the *move* and *delete* operations on nodes 4 and 5, respectively.

2.3.3 Derive the Editing Script

After the map relation and the editing operations are determined, Treed traverses T and then T' to generate the editing script. T is traversed in the post-order to assure that in the editing script, the deletion of children nodes occur before that of parent nodes. In contrast, T' is traversed in the pre-order for the parent nodes to be inserted before the children nodes.

3 CLONE DETECTION AND UPDATE

3.1 Important Concepts and Formulation

Clone detection is the process to detect the clone relation between portions of code of interest. Clone updating is the process to update the clone relation between code portions when changes occur to the codebase. This section presents our formulation of clone detection and updating.

Firstly, in Clever, any sufficiently large portion of code which represents a complete syntactical unit of a program, such as a class, a method, a statement, etc is considered as a fragment. Since Clever represents a program as an AST, a **fragment** is modeled as a subtree of an AST whose size (i.e. the number of nodes in the subtree) is larger than a pre-defined threshold. Each fragment has an Exas characteristic vector [33] used for similarity measurement as described in Section 2. If two fragments are sufficiently similar, measured by a relevant code similarity measure, they are considered to be clones of each other, and are called a **clone pair**. In Clever, two fragments with the distance between their vectors smaller than a threshold are considered as a clone pair.

All clone pairs forms the **clone relation** among all fragments. The clone relation could be conveniently represented as a **clone graph**, in which each node is a cloned fragment and each edge represents a clone pair. In Clever, a **clone group** is modeled via a connected component in the clone graph.

Definition 2 (Clone Detection): Given a program as a set of fragments, clone detection is to build the corresponding clone graph and clone groups.

Definition 3 (Change Sets): The change to a program is represented by three change sets: the sets of newly created, deleted, and modified fragments.

Definition 4 (Clone Update): Given the change sets to a program, clone updating is to update the clone graph corresponding to the changes.

After updating, the newly created, deleted, modified, and unmodified clones are reported. Clever also reports unchanged and changed groups including newly created, disappearing, expanded, shrunk, and modified groups (see Section 3.2.2).

3.2 Algorithmic Solution

3.2.1 Clone Detection

Clone Detection aims to build the clone graph for the first time. It could be started at any version and includes four steps:

1. Generate Vectors. The first step of Clone Detection is to build the fragment set. Clever reads all source files of the working version. Then, each source file is parsed into an AST. Clever traverses the AST and computes the Exas characteristic vectors for fragments (i.e. subtrees in the AST).

2. Hash Vectors into Buckets. To build the clone graph, one could find the clone pairs by a pairwise comparison on all fragments. Pairwise comparison is not efficient because a program usually has a large number of fragments. For example, JDK 1.6 with about 3,200KLOCs has over 10K fragments. In Clever, we use locality-sensitive hashing (LSH) functions to find clone pairs, which have similar characteristic vectors. A LSH function is a hash function for vectors such that the probability that two vectors having a same hash code is a *strictly decreasing* function of their corresponding *distance* [1]. In other words, two vectors having a smaller distance will have a higher probability of having the same hash code, and vice versa.

We use LSH functions (described in [1]) to hash the fragments into smaller sets that we call *buckets*, based on the hash codes of their vectors. The cloned fragments, i.e. the fragments having similar vectors, tend to be hashed into the same buckets. The other ones are less likely to be so. To increase the chance for any two cloned fragments to be hashed into the same bucket, Clever uses multiple hash functions. Thus, if such two fragments are missed by a hash function, they still have chances to be mapped into the same bucket from the other LSH functions.

Each fragment will be hashed into N buckets indexed by its hash codes produced from N independent hash functions. Hash codes produced from different hash functions are made to be different, i.e. no bucket is shared for two functions.

3. Detect Clone Pairs from a Bucket. After hashing, Clever does pairwise comparison for all fragments in each bucket to detect clone pairs. All detected pairs form the clone graph.

4. Build Clone Groups. A fragment could be cloned multiple times, i.e. having more than one clones. Clever reports clones in groups to reduce the redundancy of reporting a clone multiple times in different pairs. Clever traverses the clone graph and detects its connected components as clone groups.

3.2.2 Clone Update

The steps for updating clones after changes occur include:

1. Derive the Change Sets. The first step of clone updating is to derive the change sets, i.e. the sets of newly added, deleted, and modified fragments from the changes to the program. To do this, Clever first finds the *changed files* (i.e. deleted, added, and modified source files).

- Fragments of *deleted files* are put into the set of *deleted fragments*.
- The *added files* are parsed and traversed as in clone detection to produce newly *added fragments*.
- For each *modified file*, its two versions are mapped and compared by our tree mapping algorithm in Section 2. If a fragment (i.e. a subtree) of the old version could not be mapped to any fragment of the new version, it is considered as a *deleted fragment*. Similarly, a fragment of the new version is considered as a *new fragment*, if it is not mapped to any fragment of the old version. For the fragments that can be mapped between two versions, only the fragments that are affected by the resulting edit script are considered *modified fragments*. The other fragments are considered as *unchanged fragments*.

2. Update the Clone Graph. After the change sets are built, the second step is to update the buckets and the clone graph.

- Deleted fragments are removed from the clone graph and from buckets.
- Each modified fragment is compared with each one of its clones to check whether they are still clones of each other. If not, that clone pair is removed.
- All modified and new fragments are (re)hashed into the buckets, and are compared to other fragments (existing or newly added) in those buckets to find the new pairs. The clone graph is then updated with those new pairs.

3. Re-detect Clone Groups. Clone groups are re-detected from the new clone graph and are compared with the groups in the previous version to derive the changes to the groups.

3.2.3 Clone Report

A clone might be a part of another clone. For example, if a class is cloned from another class, its methods will also be clones of the methods of that class. Therefore, any clone group whose all members belong to the members of another clone group will be considered redundant, and will not be reported.

Since the changes to the clone graph is stored after each revision and the cloned fragments could be mapped between revisions, when reporting the clone groups at each revision, Clever is able to map the current groups with the ones in the previous version. Thus, Clever is able to report the changes to each clone group. For example, a group with new members is called “*expanded*”; a group with removed members is called “*shrunk*”; a group with all un-changed members is considered as “*un-changed*”; a group appearing only in the new version is called “*new*”; a group appearing only in the old version is considered as “*disappearing*”; and other types of groups are considered as “*changed*”.

4 CLONE CHANGE ANALYSIS

For source code, a SCM tool provides the function to detect the conflicting changes to the code and merge them consistently. For cloned code, similar functions are also needed, although the scenarios and the analyses are different.

An Illustrated Example. The illustrated example in Figure 4 is similar in spirit to a real clone-related bug described in [26], with slight modifications. In Figure 4, *A* is a portion of code from developer 1 for transferring data from `nRegs` memory blocks into the `ppTotal` array. Then, it was cloned by developer 2 into the portion of code *B* for storing information of `tRegs` processed memory blocks in another array named `ppTaken`. However, developer 2 forgot to rename an instance of `ppTotal` into `ppTaken`. This caused a bug due to inconsistent editing. Both *A* and *B* were checked into the SCM repository. Later, developer 2 checks out and modifies *B* to fix the error. (S)he also adds a statement to calculate the number of unprocessed blocks (see Clone *B'*).

At the same time, developer 1 detects a potential “out of bound” error in *A*, which could happen with an access to `ppTotal[i+1]` when $i = nRegs - 1$. (S)he modifies *A* into *A'* by adding the `if` statement. Unfortunately, since traditional SCM tool does not maintain the clone relation, (s)he might

Clone A	Clone B
<pre>for (i = 0; i < nRegs; i++) { ppTotal[i].start = prMem[i].addr; ppTotal[i].nBytes = prMem[i].size; ppTotal[i].more = ppTotal[i+1]; }</pre>	<pre>for (i = 0; i < tRegs; i++) { ppTaken[i].start = prMem[i].addr; ppTaken[i].nBytes = prMem[i].size; ppTaken[i].more = ppTotal[i+1]; }</pre>
Clone A'	Clone B'
<pre>for (i = 0; i < nRegs; i++) { ppTotal[i].start = prMem[i].addr; ppTotal[i].nBytes = prMem[i].size; if (i+1 < nRegs) ppTotal[i].more = ppTotal[i+1]; }</pre>	<pre>for (i = 0; i < tRegs; i++) { ppTaken[i].start = prMem[i].addr; ppTaken[i].nBytes = prMem[i].size; ppTaken[i].more = ppTaken[i+1]; } !Regs = nRegs - tRegs;</pre>

Fig. 4. Clone Change and Inconsistency

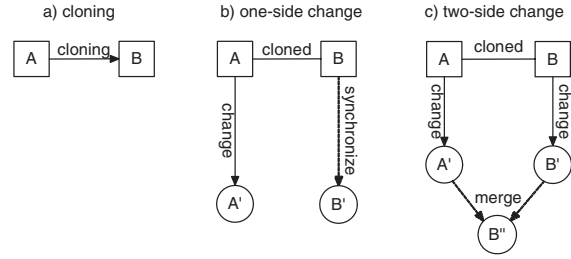


Fig. 5. Clone Change Scenarios

not know the existence of *B* and could not apply that fixing change into *B*. Similarly, developer 2 might not know about the change to *A* and could not incorporate it into *B'*.

Scenarios. The example shows that the changes to cloned code could cause the inconsistency and potential bugs. Clone-related bugs have been reported in previous research [19], [26]. More importantly, the example illustrates three possible scenarios in which clone change analysis are needed for consistent editing. Three scenarios are shown in Figure 5:

- 1) The first scenario is the *cloning* task itself. That is, when *A* is cloned into *B*, *B* is usually not textually identical to *A*, but slightly modified, such as in renaming of the identifiers. Thus, the changes in the cloning process need to be analyzed for consistency.
- 2) The second scenario, called *one-side change*, happens when *A* is modified, and *B* is unchanged (or vice versa). When *A* is checked into the repository, *B* might need to be synchronized, i.e. consistently updated with respect to the changes to *A* (e.g. the fixing changes to *A*).
- 3) The third scenario, called *two-side change*, occurs when both *A* and *B* are under modification. Changes to *A* were committed into the repository. Then, when *B* are modified and committed, the changes to *A* need to be incorporated (merged) into the changes to *B*.

Sources of Inconsistency. The example also illustrates two major sources of inconsistency: 1) inconsistency caused by renaming identifiers during cloning, and 2) inconsistency by modifying program control structures. These types of clone inconsistency were also reported in [19], [26].

There are changes that reduce the inconsistency (such as the modification to *B*). Other changes could increase the inconsistency between the cloned code (such as when *B* is cloned, or when *A* is modified in the above example). Therefore, the

clone changes need to be analyzed for consistency and then relevant clones need to be consistently updated.

Clone Change Analysis Operations. Clever provides several operations to deal with the analysis and consistent updating of clones and their changes. The rest of this section will present those operations.

4.1 Clone Matching and Differencing

Clone Matching and Differencing aims to find the matched and different elements between two cloned fragments. For example, for A and B in the example, it could find all the matches between the identifiers and program structures in the corresponding ASTs. For example, between A and B , the mapped identifiers are $i \rightarrow i$, $n\text{Regs} \rightarrow t\text{Regs}$, $pp\text{Total} \rightarrow pp\text{Taken}$, $pr\text{Mem} \rightarrow pr\text{Mem}$, and $pp\text{Total} \rightarrow pp\text{Total}$. Therefore, the bug of missing a rename operation on $pp\text{Total}$ in B can easily be found.

Clone Matching and Differencing could also be used to show the changes between two versions of a cloned fragment. For example, between A and A' , it finds the addition of the `if` statement; between B and B' , it finds the renaming of $pp\text{Total}$ to $pp\text{Taken}$ in the last expression. In general, these two operations can be used in all 3 scenarios in Figure 5.

Clone Matching and Differencing is based on Treed. From the editing script returned by Treed for two clones or two versions of a clone, their matches and differences are computed from the nodes and subtrees mapped by the editing script. For example, the differences are unmapped, updated, or moved elements of such clones (or versions).

After matched and different elements of two clones are identified, Clever finds the inconsistencies between them using Clone Consistency Validating operation.

4.2 Clone Consistency Validating

In general, it is not easy to detect all different types of inconsistencies between cloned code because the nature of inconsistency in clones depends very much on the *semantics* of the code and on the *intention of the developers* who create and change the clones. In many cases of inconsistency, there is no explicit semantics dependency between the code fragment and its cloned one. This is the major difference between clone-related inconsistency and the notion of *conflicts* in traditional SCM (In SCM, if there is no semantics dependency between two changes, there is no conflict).

Clever aims to detect only the inconsistencies involving 1) the changes of identifiers, 2) control structures, and 3) literals, which have been shown to be the major sources of clone-related bugs [19], [26]. Clever applies the following criteria on the *mapped elements* between clones and their changes to find the inconsistencies.

Definition 5 (Identifier Consistency): Given two cloned fragments (or two versions), each identifier in one fragment is mapped to one and only one identifier in the other fragment.

For example, $pp\text{Total}$ should be mapped only to $pp\text{Taken}$. However, it is mapped to both $pp\text{Taken}$ and $pp\text{Total}$, thus, the cloning change is potentially inconsistent.

Definition 6 (Structure Change Consistency): The changes to control structures (e.g. statements, expressions, etc.) of the clones should be the same.

In the illustrated example between A' and B , an additional `if` statement was inserted in A , but it does not appear in B , thus, it is potential inconsistency.

Definition 7 (Value Change Consistency): The changes to special values of literals or the names of invoked methods should be the same.

Clever is interested in only special values for literals, such as `null`, `0`, `1`, empty string, `true`, and `false`. This is based on the assumption that developers associate the special values with some meanings. When they are changed, developers might want to check them. Similarly, if a different method is called, it is likely that they intend to change the semantics of the code. Thus, such changes should be checked. Generally, Consistency Validating is needed in all 3 scenarios in Figure 5.

4.3 Clone Synchronizing

Clone Synchronizing is the operation designed for two clone change scenarios, *cloning* and *one-side change*, that is, when there is only one clone that was changed. For the two-side change, Clever uses Clone Merging, which will be discussed later. In the illustrated example, the synchronization will be applied to B when A changes into A' . Between A' and B , there are two inconsistencies: 1) of identifiers ($pp\text{Total}$ is mapped to two identifiers), and 2) of control structures (compared with B , A' has an additional `if` statement).

Clone Synchronizing works as follows. For the identifier inconsistencies, Clever recommends the mapping with the most frequencies. For example, the map $pp\text{Total}$ - $pp\text{Taken}$ appears *three* times, while $pp\text{Total}$ - $pp\text{Total}$ appears *once*. Thus, $pp\text{Total}$ - $pp\text{Taken}$ is recommended. This recommendation is based on the assumption that the developer wanted to change, and he has changed almost all instances except a few. In the cases that the mappings have the same frequency, the one with different node values is recommended (such as $pp\text{Total}$ - $pp\text{Taken}$). It is assumed that when cloning, the developers are likely to rename identifiers (In [26], it was reported that 65-67% of clones in Linux involves identifiers' renaming).

For changes in control structures, literals, and method calls, Clever recommends those changes to the mapped elements in the unchanged clone. For example, in Figure 6, it recommends the addition of the `if` statement into B at the corresponding position. Of course, when applying the changes into B , Clever also recommends the corresponding identifiers to produce consistent code ($n\text{Regs}$ in the condition is renamed to $t\text{Regs}$). Figure 6 shows the synchronized code that Clever suggests. It will apply the changes if the recommendation is accepted.

4.4 Clone Merging

Clone Merging is used for the *two-side clone change* scenario, in which two clones are modified at the same time. We assume that both A and B are modified, but only changes of A need to be synchronized into B' (since A' was committed before B').

In this section, let us show that Clone Merging operation can be solved by two operations: Clone Synchronization (in

Clone A'	Clone B*
<pre> for (i = 0; i < nRegs; i++) { ppTotal[i].start = prMem[i].addr; ppTotal[i].nBytes=prMem[i].size; if (i+1 < nRegs) ppTotal[i].more=ppTotal[i+1]; } </pre>	<pre> for (i = 0; i < tRegs; i++) { ppTaken[i].start = prMem[i].addr; ppTaken[i].nBytes=prMem[i].size; if (i+1 < tRegs) ppTaken[i].more=ppTaken[i+1]; } </pre>

Fig. 6. Clone Synchronizing Example

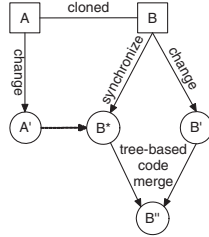


Fig. 7. Clone Merging

Section 4.3) and a classic three-way code merging operation in traditional SCM tools [29]. When B' is committed into the repository, Clever checks and sees that A , a clone of B , was modified. Thus, clone merging is required. Clever will perform a Clone Merging operation in two steps (see Figure 7):

1) Clone Synchronization is applied on B by taking into account the changes from A to A' . That is, the change to A is incorporated into B to produce a temporary version B^* .

2) A three-way Code Merge operation is applied to B^* and B' to produce the final merging result B'' . This is a three-way code merge since both B^* and B' were modified from B .

In Clever, we use Lippe's algorithm [27] for three-way code merging. It is an *operation-based merging* algorithm, which models a change between two versions as explicit operations. The merge result is produced by the application into the base version B the merged sequence of operations from two parallel sequences of operations (one sequence was applied from B to B^* and the other from B to B'). Two important inputs of Lippe's algorithm are the set of operations and the definition of conflicts among each pair of operations. The set of operations that we used was defined in Section 2 on the AST of the program. Two operations from two parallel sequences are considered as **conflict** if

- 1) They apply on the same AST node and give different

Clone A'	Clone B'
<pre> for (i = 0; i < nRegs; i++) { ppTotal[i].start = prMem[i].addr; ppTotal[i].nBytes=prMem[i].size; if (i+1 < nRegs) ppTotal[i].more = ppTotal[i+1]; } </pre>	<pre> for (i = 0; i < tRegs; i++) { ppTaken[i].start = prMem[i].addr; ppTaken[i].nBytes=prMem[i].size; ppTaken[i].more = ppTaken[i+1]; } lRegs = nRegs - tRegs; </pre>
Clone B*	Clone B''
<pre> for (i = 0; i < nRegs; i++) { ppTaken[i].start = prMem[i].addr; ppTaken[i].nBytes=prMem[i].size; if (i+1 < tRegs) ppTaken[i].more=ppTaken[i+1]; } </pre>	<pre> for (i = 0; i < tRegs; i++) { ppTaken[i].start = prMem[i].addr; ppTaken[i].nBytes=prMem[i].size; if (i+1 < tRegs) ppTaken[i].more=ppTaken[i+1]; } lRegs = nRegs - tRegs; </pre>

Fig. 8. Clone Merging Example

<pre> // ApplicationSpecificPreferencePage.java ... ApplicationSpecificRegistry.getInstance(). removeApplicationSpecificData(getSelectedAppSpecificObject()); </pre>
<pre> ... for (ApplicationSpecificObject obj: getSelectedAppSpecificObjects()) { ApplicationSpecificRegistry.getInstance(). removeApplicationSpecificData(obj); } </pre>

Fig. 9. Treed Running Result Example

results. For example, if one renames `ppTotal` to `ppTaken`, and one renames to `ppProcessed`, then they are conflicting.

2) An operation removes the nodes which the other operation needs. For example, an operation adds a new expression into a statement that was removed by the second operation. Thus, they are considered as conflicting changes.

When Clever detects the conflict between the changes from B to B^* and to B' , it requires the manual merge from users. If no conflicting change, Clever applies Lippe's algorithm for merging. Figure 8 shows the operation applied on the illustrated example. B^* is synchronized as described in Figure 6. Then, B' and B^* are merged. First, the `if` statement is added into the last statement. Then, renaming is applied to `ppTotal`. In this case, there are two renaming operations (one of B^* and one of B') applied on the same node. However, two renaming operations are identical. Thus, they could be merged. At last, the new statement for the calculation of `lRegs` is added.

5 EMPIRICAL EVALUATION

5.1 Treed Algorithm

We conducted an experiment to evaluate the accuracy and performance of our Treed algorithm. We used GEclipse from revision number 6,000 to 15,000. We randomly selected 100 revisions. For each revision, we picked one Java file and ran Treed on that file and its previous revision. We manually checked the output editing scripts for those files and found that in 92 out of 100 cases, Treed gave correct results. To illustrate interesting results, we discuss the following examples.

In the example shown in Figure 9, in the previous version, a developer invoked `removeApplicationSpecificData` function on a single `ApplicationSpecificObject`. In the new version, (s)he added a `for` loop to repeat the invocation of the same function on multiple objects returned by `getSelectedAppSpecificObjects`. Treed correctly returns the editing script that inserts the nodes representing the `for` loop and its parameters (shown in bold face), deletes the old parameter `getSelectedAppSpecificObject()` of that function call, and then inserts the new parameter `obj`.

The second example illustrates an interesting case that Treed was not quite informative. A developer modified the statement `return this.vo.getName();` into `return this.vo != null ? this.vo.getName():"Vo-Wrapper";` Since Treed found their structure not sufficiently similar, it could not map them and returned a script that deletes the old statement and inserts the new one. The other incorrect cases are similar to this case in nature. That is, when Treed could not match two inner nodes because they are too structurally

Project	LOC	Frgm	Clever			CCFinderX		
			Cov	Time	PrCs	Cov	Time	PrCs
Axis2	477K	19436	28%	47s	98%	19%	284s	100%
Columba	193K	6700	15%	18s	95%	6%	67s	100%
GEclipse	326K	10669	34%	28s	96%	12%	113s	98%
jEdit	175K	6053	6%	11s	98%	4%	50s	100%
Struts2	121K	4101	20%	27s	94%	8%	42s	100%
TomCat	321K	9649	10%	28s	98%	7%	100s	96%
Xerces	213K	6678	14%	17s	100%	11%	72s	99%

TABLE 1
Clone Detection Result

different, it returns the operation sequence [*Delete, Insert*] on those two nodes. We counted those cases as incorrect ones.

5.2 Clone Detection and Update

We conducted an experiment to evaluate Clever’s performance in both clone detection and update on a Windows XP computer using Intel(R) Core(TM) 2 Duo T7300 2GHz, 3GB RAM, and 80GB HDD. Clever was configured to process fragments with the minimum size of 50 nodes, 32 independent hash functions, and the similarity threshold σ of 0.8.

5.2.1 Clone Detection

We chose a version for each of 7 subject systems and committed it into Clever/SVN to evaluate detection performance. Table 1 shows the result on clone detection from Clever, in comparison with the clone detection tool CCFinderX [21]. Columns **LOC** and **Frgm** show the number of lines of code and that of fragments in the subject systems. Column **Time** shows the detection time, measured in seconds. Each tool was run 3 times. Processing time of Clever is indicated by the longest one to avoid the effects of file (disk) I/O caching. For CCFinderX, we took the shortest one.

Columns **PrCs** and **Cov** represent *precision* and *completeness*. In general, precision is defined as the percentage of the *correctly detected* clones in the *total detected* ones, and completeness is usually expressed in recall, i.e. the percentage of the *correctly detected* clones in the *total existing* ones. However, it is impractical to determine all existing and check all reported clones in large projects. Thus, we manually checked 100 reported clone pairs to estimate the precision. Completeness is represented by *coverage*, i.e. the percentage of detected cloned LOCs in total LOCs as in [18].

The result shows that Clever is faster and more complete than CCFinderX, while maintaining the equivalent level of high precision. For instance, on the largest subject system, Axis, of about 500KLOC, Clever takes less than 1 minute to detect more than 135KLOC of clones, while CCFinderX takes nearly 5 minutes for only about two thirds of that amount.

5.2.2 Clone Update

To examine Clever’s clone updating, we selected several consecutive revisions of three subject systems and then committed them into Clever/SVN. 100 consecutive revisions, from rev 101 to rev 200, were processed from the Columba project. In GEclipse and jEdit projects, 1,000 consecutive revisions were

Revision	Update		Re-detection		Difference	
	Cov	Time	Cov	Time	Pair	LOC
Columba						
100	15%	33.0	15%	33	0	0
110	15%	5.0	15%	33	0	0
150	15%	1.8	15%	33	0	0
200	15%	1.4	15%	34	0	0
GEclipse						
1000	37%	32.0	37%	32	0	0
1010	37%	3.4	37%	32	0	0
1050	37%	0.9	37%	32	0	0
1100	37%	0.6	37%	32	0	0
1500	37%	0.3	37%	33	0	0
2000	35%	0.4	35%	39	0	0
jEdit						
3000	7%	15.0	7%	15	0	0
3010	7%	5.8	7%	16	0	0
3050	7%	3.0	7%	17	0	0
3100	7%	2.5	7%	17	0	0
3500	6%	2.0	6%	18	0	0
4000	7%	1.8	7%	18	0	0

TABLE 2
Clone Update Result

processed, ranging from rev 1,001 to rev 2,000 and rev 3,001 to rev 4,000, respectively. Clone Detection is applied for the first revision, and Clone Update is applied for the following ones. At each revision, the result of Clone Update is compared to that of the re-run of Clone Detection at that revision. The comparison of both results including the differences in cloned lines and in clone pairs is shown in Table 2.

The result shows that Clone Update gives exactly the same result as that of re-detection (e.g. all resulting clone pairs and cloned lines of code are the same). However, the average time of updating for each revision in Table 2 is much less than that of re-detection. This is reasonable because the change at each revision is often small (Table 3). Thus, the updating process needs to process less fragments than re-detection.

Table 2 also shows that the coverage (i.e. percentage of cloned LOC) is *almost unchanged*, through the processed revisions. However, it does not imply that cloned code is unchanged. Details on clone changes will be discussed later.

For updating, Clever needs to store additional clone information (e.g. fragments’ information, buckets, clone groups). Those storage costs are acceptable. For example, for the GEclipse project, its SVN data is about 207MB, and the overhead for clone detection is about 7MB. However, the benefits of clone management and the gain in efficiency far outweigh the storage costs. Storage costs could also be reduced by reproducing characteristic vectors, instead of storing them all, even for non-cloned fragments. For example, the fragments, buckets, and groups are stored. For updating, if the tool needs to access the vector of an existing fragment (without previously stored vectors), the corresponding file could be parsed to extract vectors for that fragment. This reduces the storage cost with a slightly increasing processing time.

5.3 Clone Change Analysis

5.3.1 Changes to Clones and Groups

Table 3 shows the details on the changes to clones and groups. Column ΔF is the **total** number of changed, i.e.

Project	Revision	ΔF	C+	C-	C*	Co	Opr	LOC	P+	P1	P2	G+	G-	G*	G>	G<	Go
Columba	100-200	1316	129	25	163	1012	2.6	1.5	185	398	221	28	25	50	21	17	366
GEclipse	1700-1900	1645	118	53	315	1176	3.9	1.6	233	82	523	32	26	69	72	153	
jEdit	3000-3100	173	39	197	1008	307	1.2	0.7	121	52	976	26	9	23	164	403	

TABLE 3
Changes of Clones and Groups

Project	Revision	SI	II	VI
Columba	100-200	9	66	9
GEclipse	1700-1900	82	13	15
jEdit	3000-3100	53	78	12

TABLE 4
Clone Change Inconsistencies

newly added, deleted, and modified fragments. Column **C+**, **C-**, and **C***, are the **total** numbers of added, deleted, and modified clones, respectively, while **Co** is the **average** number of unchanged clones. Column **P+**, **P1**, and **P2** are the **total** numbers of newly created (i.e. cloning), one-side changed, and two-side changed clone pairs, respectively. The next five columns represent the **total** numbers of changed clone groups, which are the numbers of newly created (**G+**), expanded (**G>**), shrunk (**G<**), disappearing (**G-**) groups, and the groups with changed members (**G***), respectively. **Go** is the **average** number of unchanged clone groups.

Table 3 shows interesting results. Firstly, the numbers of changed fragments and clones at each revision are small. On average, there are about 20 changed fragments, one newly created pair, and 3 modified pairs. The modifications to each clone are also small: about 4 operations and 2 LOCs on average. Many of the clones and groups are unchanged. In addition, most of clone changes are two-side (i.e. both clones are modified), or cloning (i.e. new clones are created). The number of one-side changes is the least among three types.

5.3.2 Clone Consistency Validation

Table 4 shows the results of our Clone Consistency Validating experiment. The columns **SI**, **II**, **VI** are the numbers of clone pairs having structural, identifier renaming, and value changing inconsistencies, respectively (if a clone pair has two or more kinds of inconsistencies, they are counted accordingly). The table shows that most of inconsistencies are structural, that is, the cloned code tends to be modified in program structures.

As shown in Tables 3 and 4, the changes to cloned code at each revision are often small. However, those changes could potentially create many inconsistencies. Therefore, the analysis and updating of clone changes are still needed at every SCM commit, to assure that there are as few clone-related inconsistencies as possible.

Among those inconsistencies in subject systems, we found many interesting cases. Let us discuss the case of two cloned methods from two classes in Columba (see Figure 10). At one revision, only `CopyMessageCommand` was modified with the addition of the statement in boldface. In this one-side change scenario, Clever recognized the

```
public class CopyMessageCommand extends FolderCommand {
    public void updateGUI() throws Exception {
        TableChangedEvent ev=new TableChangedEvent(UPDATE,destFolder);
        MailFrameController.tableChanged(ev);
        MainInterface.treeModel.nodeChanged(destFolder);
    } ...
}

public class CheckForNewMessagesCommand extends FolderCommand {
    public void updateGUI() throws Exception {
        TableChangedEvent ev=new TableChangedEvent(UPDATE,inboxFolder);
        MailFrameController.tableChanged(ev);
    } ...
}
```

Fig. 10. Structural Inconsistency Case from Columba

Clone A → A'	Clone B
<pre>catch(IOException io) { setAbortable(false); String[] pp = {path1, io.toString()}; VFManager.error(browser, "directory-error",pp); }</pre>	<pre>catch(IOException io) { String[] args = {io.toString()}; VFManager.error(browser, "ioerror",args); }</pre>

Fig. 11. Structural Inconsistency Case from jEdit

two clones as structurally inconsistent. Clever's result is correctly confirmed because at a later revision, a change was made to `CheckForNewMessagesCommand` to fix it by adding `MainInterface.treeModel.nodeChanged(inboxFolder);`.

5.3.3 Clone Synchronization and Merging

To verify the quality of Clever's clone synchronization, we performed a controlled experiment. In 3 subject systems (Columba, GEclipse, and jEdit), we randomly chose 10 cases in which in one revision, a clone pair *A* and *B* was created and was consistently modified in a later revision into *A'* and *B'*. We made up a version consisting of *A'* and *B*. In other words, we created one-side change scenarios. Then, we ran Clever's clone synchronization on the make-up version of each case to produce the recommendation version *B**. At last, we compared *B** with *B'*. *B** is considered as a correct recommendation if it has the same program structure and exactly matched identifiers as *B'*. Otherwise, we consider it incorrect. For all 10 cases, Clever produced the correct recommendation changes.

Let us explain in details one interesting case among them. Figure 11 shows a clone pair *A* and *B* taken from jEdit revision 3791 (file `BrowserIORequest.java`). Later, *A* was changed to *A'* by adding a new statement (`setAbortable(false)`) at revision 3925. When we ran Clever in this example, Clever detected them as structural inconsistency. Because it could detect the insertion of that new statement to *A*, it provided the recommended synchronization *B* using the *Insert* operations and returned the code shown in Figure 12. We compared *B** with *B'*, a real modified version of *B* at a later time, and found that they are the same.

Clone A \rightarrow A'	Synchronize Clone B \rightarrow B* (\equiv B')
<pre>catch(IOException io) { setAbortable(false); String[] pp={path1, io.toString()}; VFSManager.error(browser, "directory-error",pp); }</pre>	<pre>catch(IOException io) { setAbortable(false); String[] args={io.toString()}; VFSManager.error(browser, "ioerror",args); }</pre>

Fig. 12. Recommended Clone Synchronizing for jEdit

Make clone A' \rightarrow A'*	Make clone B \rightarrow B'*
<pre>catch(IOException io) { setAbortable(false); String[] pp={path1,io.toString()}; VFSManager.error(browser,path1, "ioerror.directory-error",pp); }</pre>	<pre>catch(IOException io) { setAbortable(false); Log.log(Log.ERROR,this,io); String[] args={io.toString()}; VFSManager.error(browser, "ioerror",args); }</pre>
Merge Clone A'* \rightarrow A''	Merge Clone B'* \rightarrow B''
<pre>catch(IOException io) { setAbortable(false); Log.log(Log.ERROR,this,io); String[] pp={path1,io.toString()}; VFSManager.error(browser,path1, "ioerror.directory-error",pp); }</pre>	<pre>catch(IOException io) { setAbortable(false); Log.log(Log.ERROR,this,io); String[] args={path1,io.toString()}; VFSManager.error(browser,path1, "ioerror",args); }</pre>

Fig. 13. Recommended Clone Merging for jEdit

Clone Merging. In that jEdit case study, we found that at revision 4411, both A' and B' were modified into A'' and B'' shown in Figure 13. To verify Clone Merging functionality of Clever, we created two intermediate versions A'* and B'* from A' and B', respectively. Then Clone Merging was applied on both of them. The results matched with A'' and B''.

6 RELATED WORK

Recent research has shown the benefits of management tools for code clones [12], [22], [23], [32]. A related research to Clever is CloneTracker **clone management tool** [12]. It is based on a clone tracking tool with the same name [11]. CloneTracker [12] uses CRD, a light-weight clone region description scheme, to map clone groups from the previous version to the current groups. However, some detected clones could be missed due to the approximate nature of CRD mapping [11]. In contrast, Clever uses its tree mapping algorithm, a more precise approach, that avoids losing detected clones in clone tracking. Moreover, from one version to another, CloneTracker re-runs the detection only on changed and currently tracked files. Thus, it could miss cross-revision clone pairs. Such a pair is the result of a copy of a fragment from an un-changed file into a changed one. Our previous experiment showed that there are nearly 46,000 cross-revision clone pairs in GEclipse [35]. Therefore, in both tracking old and detecting/updating new groups, CloneTracker cannot fully achieve completeness. Via Treed and SVN's change report, Clever derives the changed fragments, and then updates clone groups, tracks old and detects new ones with reasonable storage overhead.

Other **clone tracking tools** include [4], [17], [31]. Clone Detection Toolbox [32] uses Unix `diff` to get the changes to clones, tracks them in different versions, and updates its clone database. However, it requires the re-run of clone detection on the entire new version. Furthermore, its line-based tracking of

clones does not adapt well with modifications [11]. Bakota *et al.* [4] proposed the mapping of clones from one version to another based on a light-weight AST-based similarity measure. Mende *et al.* [31] proposed a token-based similarity approach for grow-and-prune model in evolving software. In brief, the aforementioned clone tracking approaches might result in incompleteness in tracking/managing clones. None of them supports *clone-aware* synchronizing and merging.

To reduce the time complexity for re-detection when software changes, recent research has focused on **incremental clone detection** [16]. iClones [16] represents each fragment as a sequence of tokens, stores all of such sequences on suffix-trees, and traverses such trees to find groups of identical and similar sequences. When code changes, the suffix-trees are updated with new/deleted sequences of the changed files, and then clone groups are re-produced. Clever's clone updating was based on ClemanX [34], [35], our previous work on incremental clone detection. However, aiming for efficient re-detection, ClemanX fell short of the goal of change management for clones and groups. In addition, it handles changes at the file level, rather than at the fragment level as in Clever.

Many approaches for code **clone detection** have been proposed [6], [37]. Generally, they can be classified based on their code representations. The typical categories are text-based [10], [30], token-based [3], [21], [26], tree-based [5], [18], and graph-based [15], [24]. The text-based and token-based are usually efficient but could not detect the clones with many modifications. In contrast, graph-based approaches, though providing clones of higher level of abstraction, are time-consuming in detecting similar subgraphs. Deckard [18] introduced the use of vectors in clone detection. In [33], we showed that our vector representation for tree-based fragments is a more generalized and accurate approach than Deckard's vectors. Deckard tool counts only the distinct AST node types in a subtree for a fragment, while Clever captures structural features via paths and sibling sets. Moreover, Clever differs from Deckard in the usage of LSH. For every fragment, even non-cloned ones, Deckard uses LSH to find similar fragments. In Clever, non-cloned fragments tend to be hashed to singleton buckets, and are not compared to other fragments. Chilowicz *et al.* [9] propose a signature for a subtree using a tree fingerprint method. Treed is similar in nature to ChangeDistiller [14]. However, our Exas similarity measure captures better the structural similarity than the counting approach of matched nodes in ChangeDistiller. Also, Treed uses the alignment of lines from SCM for the better detection of mapped nodes.

Support for **consistent editing** for clones in CloneTracker [11] is interactive and only for a single user. With collaborative services from SVN, Clever supports clone change management in team development. Similar to CloneTracker, in Codelink editor [38], a user can modify a fragment and changes can be interactively applied to its cloned fragment. Both of them use token-based mapping between two clones. Clever uses tree-based alignment. Libra [28] searches fragments for simultaneous changes. Within Eclipse, CREn [17] tracks clones and helps consistently renaming of identifiers. In [25], consistency is defined in a text line-based approach.

SCM systems have a long history [13]. While early SCM

tools (e.g. CVS) provide versioning for entire files, more advanced SCM systems [8] have also fine-grained version control support. However, none of existing SCM tools support clone-aware change management. Several approaches have been introduced for conflicting change detection/resolution in SCM tools. Direct conflict detection mechanisms, often parts of software merging techniques [29], handle only the changes to the same artifact in two different ways. *Text-based* merge tools (e.g. in CVS) consider software artifacts merely as text files. *Syntactical* merging is more powerful than textual merging because it takes the syntax of artifacts into account [2]. However, they cannot detect conflicts when the merged program is syntactically correct but semantically invalid. To deal with this, *semantic-based* merge algorithms were developed. Those algorithms [39] can detect behavioral conflicts. Recent advanced techniques for indirect conflict resolution [20], [36] raise the awareness among developers on the changes that semantically affect other artifacts. However, all of aforementioned approaches cannot deal with the changes to cloned code fragments.

7 CONCLUSIONS

This paper introduces Clever, a novel clone-aware SCM system which provides clone management support, including clone detection and update, clone change management, clone consistency validating, clone synchronizing, and clone merging. Clever represents source code and clones as the subtrees of ASTs, measures code similarity based on the structural characteristic vectors, and describes codes changes as tree editing scripts. We have introduced new algorithms and techniques to compute the tree editing script; to detect and update code clones; to analyze the changes of cloned code, to validate their consistency, and to recommend the relevant synchronization. Our empirical study on many large-scale, open-source systems shows that Clever is highly efficient and accurate in clone detection and updating, and provides useful consistent validation and synchronization of clone changes.

Acknowledgment. This project was funded in part by a grant from Vietnam Education Foundation (VEF) for the first author.

REFERENCES

- [1] A. Andoni and P. Indyk. E2LSH 0.1 User manual. <http://web.mit.edu/andoni/www/LSH/manual.pdf>.
- [2] U. Askund. Identifying Conflicts During Structural Merge. In *Proceedings of Nordic Workshop on Programming Environment*, pages 231–242, 1994.
- [3] B. S. Baker. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. *SIAM Journal on Computing*, 26(5):1343–1362, October, 1997.
- [4] T. Bakota, R. Ferenc, and T. Gyimothy. Clone Smells in Software Evolution. *ICSM'07*, pp. 24–33, 2007.
- [5] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. *ICSM'98*, 1998.
- [6] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.
- [7] CCFinderX. <http://www.ccfinder.net>.
- [8] M. C. Chu-Carroll, J. Wright, D. Shields. Supporting Aggregation in Fine-grained SCM. In *FSE'06*, pp. 99–108. ACM, 2002.
- [9] M. Chilowicz, E. Duris, G. Rousset. Syntax Tree Fingerprinting for Source Code Similarity Detection. *ICPC'09*, IEEE CS, 2009.
- [10] S. Ducasse, M. Rieger, and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. *ICSM'99*.
- [11] E. Duala-Ekoko and M. P. Robillard. Tracking Code Clones in Evolving Software. In *ICSE '07*, pp. 158–167, IEEE CS, 2007.
- [12] E. Duala-Ekoko and M. P. Robillard. CloneTracker: Tool Support for Code Clone Management. In *ICSE08-Demo*, 2008.
- [13] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Tichy, D. Weber. Impact of SE Research on the practice of SCM. *ACM TOSEM*, 14(4):383–430, 2005.
- [14] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *TSE*, 33(11):725–743, 2007.
- [15] M. Gabel, L. Jiang, and Z. Su. Scalable Detection of Semantic Clones. *ICSE'08*, pages 321–330, IEEE CS, 2008.
- [16] N. Gode and R. Koschke. Incremental Clone Detection. In *CSMR'09*, pages 219–228, IEEE CS, 2009.
- [17] P. Jablonski and D. Hou. CReN: a Tool for Tracking Copy-and-Paste Code Clones and Renaming Identifiers Consistently in the IDE. In *ETX'07*, pages 16–20, ACM Press, 2007.
- [18] L. Jiang, G. Mishherghi, Z. Su, and S. Glondu. Deckard: Scalable and Accurate Tree-based Detection of Code Clones. In *ICSE'07*.
- [19] L. Jiang, Z. Su, and E. Chiu. Context-Based Detection of Clone-Related Bugs. In *FSE'07*, pages 55–64. IEEE CS, 2007.
- [20] R. Hegde and P. Dewan. Connecting Programming Environments to Support Ad-Hoc Collaboration. In *ASE'08*, 2008.
- [21] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a Multi-linguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE TSE*, 28(7):654–670, 2002.
- [22] C. Kasper and M. Godfrey. “Cloning considered harmful” Considered Harmful: Patterns of Cloning in Software. In *Empirical Software Engineering*, 13(6):645–692, 2008.
- [23] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An Empirical Study of Code Clone Genealogies. In *FSE'05*. ACM Press, 2005.
- [24] R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In *SAS'01*, pages 40–56, 2001.
- [25] J. Krinke. A Study of Consistent and Inconsistent Changes to Code Clones. In *WCRE'07*, pages 170–178. IEEE CS, 2007.
- [26] Z. Li, S. Lu, S. Myagmar. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software. *IEEE TSE*, 32(3), 2006.
- [27] E. Lippe and N. van Oosterom. Operation-Based Merging. In *ACM SIGSOFT Softw. Eng. Notes*, 17(5):78–87. ACM, 1992.
- [28] Y. Higo, Y. Ueda, S. Kusumoto, K. Inoue. Simultaneous Modification Support based on Code Clone Analysis. *APSEC'07*.
- [29] T. Mens. A State-of-the-Art Survey on Software Merging. *IEEE Trans. on Software Engineering*, 28(5):449–462, 2002.
- [30] A. Marcus and J. Maletic. Identification of High-level Concept Clones in Source Code. In *ASE'01*, pp. 107–114. IEEE CS, 2001.
- [31] T. Mende, R. Koschke, and F. Beckwermet. An Evaluation of Code Similarity Identification for the Grow-and-Prune Model. *JSME*, 21(2):143–169, John Wiley & Sons, 2009.
- [32] F. Mitter. Tracking Source Code Propagation in Software Systems via Release History Data and Code Clone Detection. Diploma Thesis, University of Zurich, 2006.
- [33] H.A. Nguyen, T.T. Nguyen, N.H. Pham, J.M. Al-Kofahi, and T.N. Nguyen. Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection. In *FASE'09*, LNCS 5503, pages 440–455. Springer-Verlag, 2009.
- [34] T.T. Nguyen, H.A. Nguyen, N.H. Pham, J.M. Al-Kofahi, and T.N. Nguyen. ClemanX: Incremental Clone Detection Tool for Evolving Software. In *ICSE'09 Demo*. IEEE CS, 2009.
- [35] T.T. Nguyen, H.A. Nguyen, N.H. Pham, J.M. Al-Kofahi, and T.N. Nguyen. Scalable and Incremental Clone Detection for Evolving Software. In *ICSM'09*, IEEE CS, 2009.
- [36] A. Sarma, G. Bortis, A. van der Hoek. Towards supporting awareness of indirect conflicts across SCM workspaces. *ASE'07*.
- [37] R. Tairas - Bibliography of Code Detection Literature. <http://students.cis.uab.edu/tairas/clones/literature/>.
- [38] M. Toomim, A. Begel, and S.L. Graham. Managing Duplicated Code with Linked Editing. In *VLHCC'04*, IEEE CS, 2004.
- [39] W. Yang, S. Horwitz, and T. Reps. A Program Integration Algorithm that accommodates semantics-preserving Transformations. *ACM Trans. Softw. Eng. Methodol.*, 1(3):310–354, 1992.