



Cleaning up copy–paste clones with interactive merging

Krishna Narasimhan¹ · Christoph Reichenbach²  · Julia Lawall³

Received: 19 September 2016 / Accepted: 11 June 2018
© The Author(s) 2018

Abstract

Copy-paste-modify is a form of software reuse in which developers explicitly duplicate source code. This duplicated source code, amounting to a code clone, is adapted for a new purpose. Copy-paste-modify is popular among software developers, however, empirical evidence shows that it complicates software maintenance and increases the frequency of bugs. To allow developers to use copy-paste-modify without having to worry about these concerns, we propose an approach that automatically merges similar pieces of code by creating suitable abstractions. Because different kinds of abstractions may be beneficial in different contexts, our approach offers multiple abstraction mechanisms, which were selected based on a study of popular open-source repositories. To demonstrate the feasibility of our approach, we have designed and implemented a prototype merging tool for C++ and evaluated it on a number of code clones exhibiting some variation, i.e., near-miss clones, in popular Open Source packages. We observed that maintainers find our algorithmically created abstractions to be largely preferable to the existing duplicated code.

Keywords Program analysis · Static analysis · Clone management · Source code analysis

✉ Christoph Reichenbach
christoph.reichenbach@cs.lth.se

Krishna Narasimhan
knarasimhan@itemis.de

Julia Lawall
julia.lawall@lip6.fr
<https://pages.lip6.fr/Julia.Lawall/>

¹ Itemis, Faßnachtstraße 1, 70378 Stuttgart, Germany

² Department of Computer Science, Lund University, Lund, Sweden

³ Sorbonne University/Inria/LIP6, Paris, France

1 Introduction

As software developers add features to their programs, they often find that some new feature is very similar to an existing one. The developers then face a choice: they can either introduce a (possibly complex) abstraction into the existing, working code, or copy and paste the existing code and modify the result. Introducing the abstraction produces smaller, more maintainable code but alters existing functionality on the operational level, carrying the risk of introducing inadvertent semantic changes. Copying, pasting, and modifying introduces duplication in the form of *near-miss clones* (type-3 clones in the terminology of Koschke et al. (2006), i.e., clones with nontrivial differences), which tends to decrease maintainability, but avoids the risk of damaging existing functionality (Kapsner and Godfrey 2008).

Code duplication is widespread (Laguë et al. 1997; Baxter et al. 1998), especially if we count both exact duplicates and near-miss clones. However, code duplication is unpopular in the practitioner literature (Hunt and Thomas 1999) and “can be a substantial problem during development and maintenance” (Juergens et al. 2009) as “inconsistent clones constitute a source of faults”. Similarly, the C++ developers taking part in our user study (Sect. 2.1), preferred to read code using abstraction rather than duplication. This suggests that there is a discrepancy, which we refer to as the *reuse discrepancy*, between what developers want and what they do.

Kapsner and Godfrey (2008) offer one possible explanation: they claim that “code cloning can be used as an effective and beneficial design practice” in a number of situations, but observe that existing code bases include many clones that do not fit their criteria for a ‘good clone’. We suggest an alternative explanation, namely that developers view cloning as an implementation technique rather than a design practice: they would prefer abstraction, but find copy–paste–modify safer to use. In an informal poll, we found evidence that supports this idea.

In this paper, we look at popular open-source repositories, gather clone groups that are representative of copy–pasted code, and study them to motivate potential mechanisms for abstracting copy–pasted code. We propose a novel solution to the reuse discrepancy that offers all the speed and simplicity of copy–paste–modify together with the design benefits offered by abstraction, by designing a refactoring algorithm to merge similar code semi-automatically. With our approach, developers reuse code by copying, pasting, and modifying it, producing near-miss clones. Developers then invoke our tool to merge two or more near-miss clones back into a single abstraction. Since there may be multiple ways to introduce an abstraction, our tool may ask the developer to choose which abstractions are preferred during the merge. Our tool is easily extensible, so that developers may add support for their own abstractions (e.g., project-specific design patterns).

Conceptually, this approach can be applied at any code granularity, but we focus on method-level clones in this work because methods represent well-recognized units of reuse. Analogously, our approach utilizes methods as abstraction devices.

We find that our approach is not only effective at solving the aforementioned reuse discrepancy, but also produces code that meets the quality standards of existing open-source projects. Moreover, our approach can improve over manual abstraction in terms of correctness: as with other automatic refactoring approaches, ensuring correctness

only requires validating the (small number of) constituents of the automatic transformation mechanism (Reichenbach et al. 2009; Schäfer et al. 2009), as opposed to the (unbounded number of) hand-written *instances* of manual abstractions that are required without a tool.

Our contributions are as follows:

- We describe the results of our manual study of various clone groups in popular open-source repositories, and identify abstraction mechanisms that may be relevant to them.
- We present a small user study that we have carried out, in which C++ programmers were asked to use copy–paste–modify or abstraction in a set of coding tasks. We also present the results of a poll among these C++ programmers on their opinion of the desirability of the various forms of produced code. While our study and poll have only a small sample size, they suggest that developers prefer to use copy–paste–modify for development, but find the results of abstraction to be preferable.
- We describe an algorithm that can automatically or semi-automatically merge near-miss clones and introduce user-selected abstractions. The abstraction mechanisms supported in our implementation of this algorithm are motivated by our manual study of clone groups.
- We report on initial experiences with our algorithm on popular C++ projects drawn from open-source repositories. We find that the generated merged code is of sufficiently high quality to be accepted as a replacement for unmerged code in most cases.

The rest of this paper is organized as follows. Section 2 presents our user study and poll that further guide the design of our approach. Section 3 describes our experiences with a manual study of clone groups from popular open-source repositories. Section 4 describes our merge algorithm, and Section 5 gives an overview of its implementation. Section 6 discusses the correctness of our approach and limitations of our current implementation. Section 7 presents our evaluation on various open-source software projects, and Section 8 presents a larger example in detail. Finally, Section 9 discusses related work, and Section 10 concludes.

This article is an extension of our earlier paper Narasimhan and Reichenbach (2015), published at the conference “Automated Software Engineering”. Compared to that work, this article adds a discussion of our initial exploration to identify important abstraction patterns (Sect. 3), adds technical and algorithmic details regarding the merge process (Sect. 4), presents a more general model for describing and understanding abstraction patterns (Sect. 4.3.3), provides a sketch of our reasoning for the behaviour preservation property of the algorithm (Sect. 6), includes additional insights from our evaluation, including a comparison to the clone detector NiCad (Sect. 7), provides a detailed case study of applying our tool to additional clones detected by NiCad (Sect. 8), and expands on the related work in depth and scope (Sect. 9).

2 User study and informal poll

Past work on clone detection has found that clones are widespread (Laguë et al. 1997; Koschke et al. 2006; Baxter et al. 1998). We hypothesize that a key cause for this prevalence of clones is that *copy–paste–modify makes software developers more productive*, at least in the short term. To explore this hypothesis, we conducted a preliminary, exploratory experiment with a group of graduate student volunteers.

2.1 Benefits of copy–paste–modify

For our user study, we selected five pairs of C++ methods from Google Protobuf,¹ Facebook HHVM,² and Facebook RocksDB,³ randomly choosing from the set of near-miss clones reported by our own clone detector, described in Sect. 3. We then removed one of the methods and asked five graduate students with 2 months, 3 months, and 1 year, 4 years, and 10 years of (self-reported) C++ programming experience, respectively, to implement the missing functionality. We asked the students with 3 months and 4 years of experience to modify the existing method to support both the existing and the new functionality (i.e., to perform *manual abstraction*), and the remaining students to use *copy–paste–modify*. All students worked on all five tasks.

We found that the students using copy–paste–modify were almost universally faster in completing their objectives (2–15 min) than the students who performed manual abstraction (7–55 min, with three tasks left incomplete). We found only one exception, where the best-performing student using manual abstraction completed the task in the same time as the worst-performing student using copy–paste–modify. Since the three students using copy–paste–modify finished first and had a lot of their allocated time still left, we asked two of the copy–paste group to abstract two of the tasks, and the third copy–paste group user to abstract one of the tasks. Despite their familiarity with the code, they consistently performed worse (taking more than twice as long as before) when completing the same task again with manual abstraction. However, the same developers showed a preference for *having* abstractions as a result (in 12 cases, vs. 5 for copy–paste–modify, out of 20 responses, cf. Appendix A).

While our numbers are too small to be statistically significant, they provide evidence that copy–paste–modify can be more effective than manual abstraction at accomplishing reuse at the method level.

2.2 Copy–paste–modify versus manual abstraction

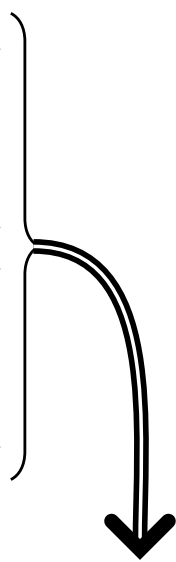
To understand *why* copy–paste–modify might be easier, consider the function `costFunction1` from Fig. 1. This function (adapted from the OpenAge⁴ project) computes the Chebyshev distance of two 2-dimensional coordinates. The implemen-

¹ <https://github.com/Google/protobuf>

² <https://github.com/Facebook/hhvm>

³ <https://github.com/Facebook/rocksdb>

⁴ <http://openage.sft.mx/>



```

cost_t costFunction1(coord start, end) {
    cost_t dx = start.ne - end.ne;
    cost_t dy = start.se - end.se;
    return std::max(dx, dy);
}

cost_t costFunction2(coord start, end) {
    cost_t dx = start.ne - end.ne;
    cost_t dy = start.se - end.se;
    return std::hypot(dx, dy);
}

cost_t costFunctionM(coord start, end,
                    bool chebyshev) {
    cost_t dx = start.ne - end.ne;
    cost_t dy = start.se - end.se;
    if (chebyshev) {
        return std::max(dx, dy);
    } else {
        return std::hypot(dx, dy);
    }
}

```

Fig. 1 An example of merging two functions by introducing a boolean parameter and an if statement

tation consists of a function header with formal parameters, a computation for the intermediate values dx and dy , and finally a computation of the actual Chebyshev distance from dx and dy .

At some point, a developer decides that a different distance function is needed, describing the beeline distance between two points (i.e., $\sqrt{dx^2 + dy^2}$). Computing this distance requires almost the same steps as implemented in `costFunction1`, except for calling the standard library function `std::hypot` instead of `std::max`. At this point, the developer faces a choice: she can copy and paste the existing code into a new function (requiring only a copy, paste, and rename action) and modify the call from `std::max` to `std::hypot` (a trivial one-word edit), or she can manually transform the function `costFunction1` into a more abstract version, such as `costFunctionM` (depicted on the bottom right in Fig. 1).

This transformation from the copy–pasted code to an abstracted code requires introducing a new parameter, introducing an if statement, adding a new line to handle the new case, and updating all call sites with a new argument (perhaps using a suitable automated refactoring). Intellectually, the developer must reason about altering the function’s control flow, formal parameters, and any callers that expect the old functionality, whereas with copy–paste–modify, they only need to concern themselves with the exact differences between what already exists and what they now need.

We observe the need to devise an algorithm that takes the definitions of `costFunction1` and `costFunction2` and abstracts them into a common

`costFunctionM`, taking care that any callers still continue to work correctly. Note that there are multiple possible strategies for `costFunctionM`. For example, we could pass `std::hypot` or `std::max` as function parameters, wrap them into delegates, or pass an enumeration parameter to support additional metrics within this one function. The ‘best’ abstraction mechanism may depend on style preferences, performance considerations, and plans for future extension.

3 Resolution patterns for merging method-level code clones

To understand the commonly occurring differences in such clones in C++ code and to motivate specific approaches to merging method-level clones, we conducted a study of clone groups from top trending Open Source GitHub repositories. In this section, we describe the set-up of the study and report on our findings and insights.

As we were not able to find a method-level clone detector for C++, we chose to implement a simple clone detector of our own based on the robust tree edit distance algorithm RTED (Pawlik and Augsten 2011). The tree edit distance represents a metric for the amount of edits required to transform one tree into another, which intuitively represents a basis for a metric of similarity for copy–paste–modified code. We adapted the existing implementation,⁵ which works on trees of strings, to support the AST nodes of the Eclipse CDT.⁶ RTED computes the nodes that we need to add to or remove from one AST to obtain another; its output is an *edit list*, i.e., a list of *delete*, *insert* or *relabel* operations:

- **delete** a node and connect its children to its parent, maintaining their order.
- **insert** a node between two neighbouring siblings
- **relabel** a node, essentially replacing one node by another.

Based on the results of the RTED algorithm, we calculated the *edit distance*, which is the size of the edit list, between every possible pair of methods in the top 6 trending C++ repositories in GitHub in the month of December 2014. The repositories were:

1. ForestDB, a key-value store, developed by Couchbase:
<https://github.com/couchbase/forestdb>
2. Google Protobuf, a library for data interchange:
<https://github.com/google/protobuf>
3. Open CV, a computer-vision library, originally from Intel:
<https://github.com/Itseez/opencv>
4. Facebook’s HHVM, a virtual machine supporting Hack and PHP:
<https://github.com/facebook/hhvm>
5. Facebook’s RocksDB, a persistent key-value store for fast storage environments:
<https://github.com/facebook/rocksdb>
6. Tiled, a map editor:
<https://github.com/bjorn/tiled>

⁵ <http://tree-edit-distance.dbresearch.uni-salzburg.at/>

⁶ <https://eclipse.org/cdt/>

We then used the edit distance to determine how similar the methods in each method pair were. Specifically, we normalized the edit distance by the size of the bigger of the two methods, as given by its number of AST nodes. This normalized edit distance is 0 whenever the methods are identical, and 1 whenever they are completely dissimilar.

We configured our tree edit distance checker to only report clones with a normalized tree edit distance below a certain threshold. For a threshold of 0.5, we observed 111 clone pairs. We further filtered our sample set by picking the top 10 clone pairs that were closest (from both above and below) to the thresholds of 0.1, 0.2, 0.3, 0.4 and 0.5. This left us with 50 clone pairs. We manually analyzed the differences between the individual clone pairs and identified potential ways of merging them. For example, if the difference was between two literal expressions (constants) we could merge them by abstracting the literal as a global variable or an extra parameter. If the difference was between two types, we could introduce a template type argument. Some kinds of differences could also be resolved e.g. by using a delegate (Gamma et al. 1995), but in this work we focus on ways of merging that are non-intrusive, in the sense that they do not require introducing new classes.

In the following, we describe the top three types of differences we observed, using real examples from our sample set and the methods of merging we propose for those differences. For readability, we illustrate each case using very small examples that would probably not be worthwhile to merge in practice. We present more realistic examples in our evaluation in Sects. 7 and 8. In the examples that follow, the code differences and the manually generated parameters and code segments are highlighted with a gray background.

Difference type 1: Constants Consider the following functions from Google’s Protobuf (normalized tree edit distance: 0.25)⁷:

```
// Return the name of the AssignDescriptors()
// function for a given file.
string GlobalAssignDescriptorsName(const string& filename) {
    return "protobuf_AssignDesc_" + FilenameIdentifier(filename);
}

// Return the name of the ShutdownFile()
// function for a given file.
string GlobalShutdownFileName(const string& filename) {
    return "protobuf_ShutdownFile_" + FilenameIdentifier(filename);
}
```

These functions differ only in the string “protobuf_AssignDesc_” or “protobuf_ShutdownFile_”, used to make up the beginning of the return value. As both strings are constants, we can create a merged version of these functions, in which the differences are resolved using a global variable or an extra parameter. The following code shows the result when using a global variable. The code includes both the merged function and the original functions modified to use the new merged version.

⁷ https://github.com/google/protobuf/blob/6ef984af4b0c63c1c33127a12dcfc8e6359f0c9e/src/google/protobuf/compiler/cpp/cpp_helpers.cc

```
string globalVar = "";
string mergedFunction(const string &filename) {
    return globalVar + FilenameIdentifier(filename);
}

string GlobalAssignDescriptorsName(const string& filename) {
    globalVar = "protobuf_AssignDesc_";
    return mergedFunction(filename);
}

string GlobalShutdownFileName(const string& filename) {
    globalVar = "protobuf_ShutDownFile_";
    return mergedFunction(filename);
}
```

The following code likewise shows the result when using an extra parameter:

```
string mergedFunction(const string &filename, string extraParam) {
    return extraParam + FilenameIdentifier(filename);
}

string GlobalAssignDescriptorsName(const string& filename) {
    return mergedMethod(fileName, "protobuf_AssignDesc_");
}

string GlobalShutdownFileName(const string& filename) {
    return mergedMethod(fileName, "protobuf_ShutdownFile_");
}
```

Difference type 2: Types The following code shows another pair of functions from Google's Protobuf⁸ that differ in a parameter type (normalized tree edit distance: 0.25):

```
string TextFormat::FieldValuePrinter::PrintInt32(int32 val) const {
    return SimpleItoa(val);
}

string TextFormat::FieldValuePrinter::PrintUInt32(uint32 val) const {
    return SimpleItoa(val);
}
```

The code below merges these functions using a template argument:

```
template <typename T>
string TextFormat::FieldValuePrinter::PrintInt(T val) const {
    return SimpleItoa(val);
}

string TextFormat::FieldValuePrinter::PrintInt32(int32 val) const {
    return PrintInt<int32>(val);
}

string TextFormat::FieldValuePrinter::PrintUInt32(uint32 val) const {
    return PrintInt<uint32>(val);
}
```

Difference type 3: Statements The following code shows a pair of functions from Facebook's RocksDB⁹ that differ in various aspects of a statement containing a function call (normalized tree edit distance: 0.41):

⁸ https://github.com/google/protobuf/blob/6ef984af4b0c63c1c33127a12dcfc8e6359f0c9e/src/google/protobuf/text_format.cc

⁹ <https://github.com/facebook/rocksdb/blob/767777c2bd7bf4be1968dbc35452e556e781ad5f/db/c.cc>

```

void rocksdb_writebatch_merge(rocksdb_writebatch_t* b,
    const char* key, size_t klen, const char* val, size_t vlen) {
    b->rep.Merge(Slice(key, klen), Slice(val, vlen));
}

void rocksdb_writebatch_merge_cf(rocksdb_writebatch_t* b,
    rocksdb_column_family_handle_t* column_family,
    const char* key, size_t klen, const char* val, size_t vlen) {
    b->rep.Merge(column_family->rep, Slice(key, klen), Slice(val, vlen));
}

```

We propose two ways to merge such statement level differences, using either conditionals or a switch statement. We show only the result using conditionals, below:

```

void abstractedFunction(rocksdb_writebatch_t* b,
    rocksdb_column_family_handle_t* column_family,
    const char* key, size_t klen, const char* val, size_t vlen,
    int functionID) {
    if(functionID == 1) {
        b->rep.Merge(Slice(key, klen), Slice(val, vlen));
    }
    else if(functionID == 2) {
        b->rep.Merge(column_family->rep, Slice(key, klen), Slice(val, vlen));
    }
}

void rocksdb_writebatch_merge(rocksdb_writebatch_t* b,
    const char* key, size_t klen, const char* val, size_t vlen) {
    abstractedFunction(b, NULL, key, klen, val, vlen, 1);
}

void rocksdb_writebatch_merge_cf(rocksdb_writebatch_t* b,
    rocksdb_column_family_handle_t* column_family,
    const char* key, size_t klen, const char* val, size_t vlen) {
    abstractedFunction(b, column_family, key, klen, val, vlen, 2);
}

```

Based on our analysis of a number of near-miss clone methods from popular open source repositories, we gathered the most common types of code near-clone differences in practice, i.e., **constants**, **types** and **statements**. We then designed strategies for merging the clones that we observed and combined these strategies in our merging algorithm.

4 Merging algorithm

To illustrate how our algorithm merges a collection of near-miss clone functions into an abstracted function, we use the three functions at the top of Fig. 2 as a running example. These synthetic functions are unlikely merge candidates, since they are both small and rather dissimilar, but they illustrate special cases in our algorithm and show that our approach works even for code with a large degree of variation.

4.1 Abstract syntax trees

Our algorithm works at the Abstract Syntax Tree (AST) level. Figure 3 shows simplified ASTs, omitting operators for conciseness, for the functions in Fig. 2.

```

void function1() {
    b(c,k(d));
    y = f1;
    x(z);
}

void function2() {
    b(c,k(e));
    y = f2;
    x(z);
}

void function3() {
    b2();
    y = f3;
    n();
    x(z);
}

void fnMerged(int functionId, int fValue, int bParam) {
    if(functionId == 12) {
        b(c, k(bParam));
    }
    else if(functionId == 3) {
        b2();
    }
    y = fValue;
    x(z);
}

```

Fig. 2 Example of a three-way merge supported by our tool

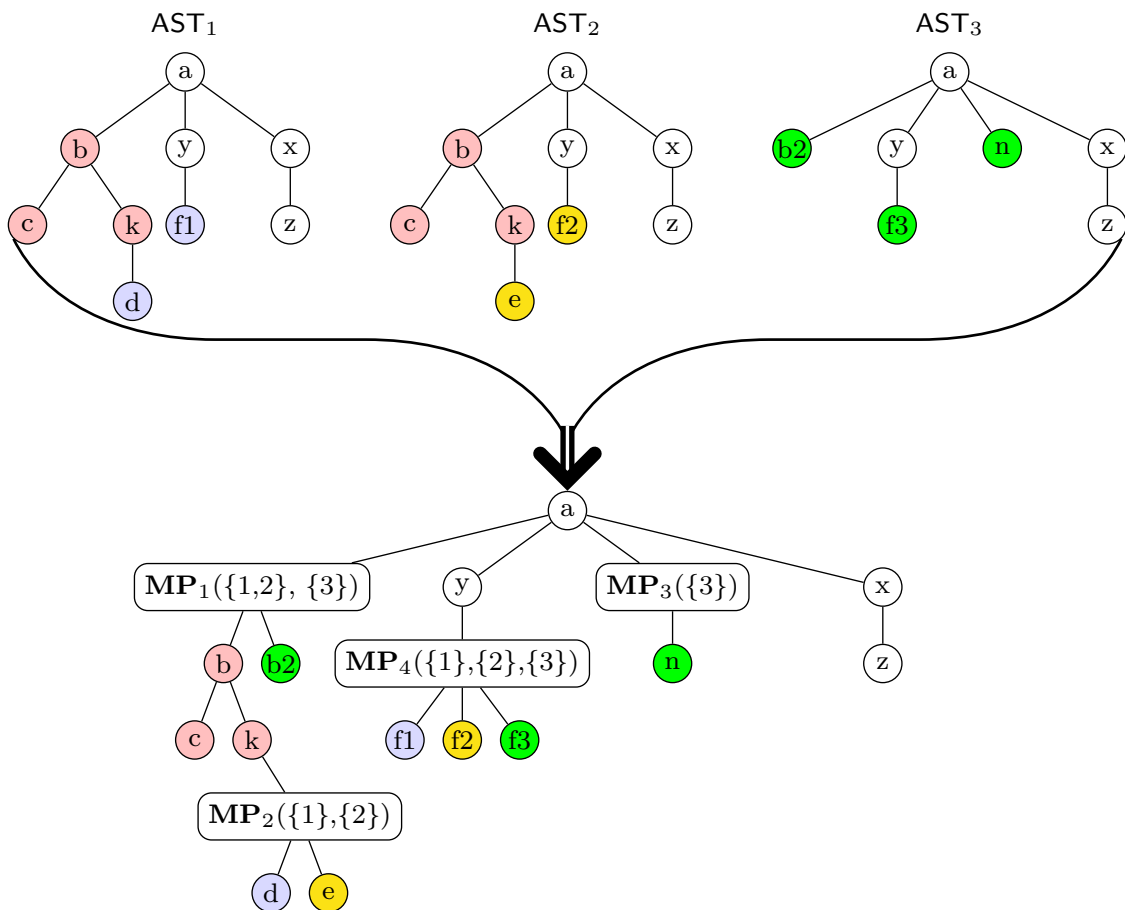


Fig. 3 Three-way merge in AST form

In our ASTs, each node contains a *label* and a *position*. The *label* of a node is the type of the node along with any content the node contains. For example, in an AST representing the declaration `int x = 10;`, the node corresponding to `int` would have type **Type** and content `int`, and the node corresponding to `10` would have type **Literal Expression** and content `10`. The *position* of a node is the traversal path to this node

from the root of the AST (Negara et al. 2012). The position is a list of numbers, with each number representing the *offset*, starting with 1, from the leftmost child of each node's parent to the node. In Fig. 3, the *position* of the node 'y' in all of the ASTs is (1, 2). The *position* of the node 'e' in AST₂ is (1, 1, 2, 1).

4.2 RTED

Our algorithm relies on the RTED algorithm to identify common nodes and inserted nodes. As described in Sect. 3, RTED computes the edit distance between two trees, i.e., the number of edit operations that are required to transform one AST into another.

Each element of an edit list is a pair (n_a, n_b) , describing a single edit operation, transforming node n_a in AST_A into node n_b in AST_B. The edit list is computed based on the tree structure and the node content, but independently of the node position. The edit lists produced by RTED are completely symmetric, i.e., the result of applying RTED to a pair of ASTs (AST_x, AST_y) is simply the reverse of the result of applying RTED to (AST_y, AST_x). At most one component of an element of an edit list can be 0, indicating that the node in the other component is inserted into its corresponding tree, or, conversely, removed from the tree that it occurs in. For example, $(a, 0)$ indicates that node a is inserted into the left-hand side tree.

The edit lists of each pair of ASTs in our example in Fig. 3 are:

- Edit List of (AST₁, AST₂): (d, e), (f1, f2)
- Edit List of (AST₂, AST₃): (b, b2), (c, 0), (k, 0), (e, 0), (f2, f3), (0, n)
- Edit List of (AST₁, AST₃): (b, b2), (c, 0), (k, 0), (d, 0), (f1, f3), (0, n)

Based on the results of RTED, our algorithm identifies nodes that do not appear in any edit list as being common to all ASTs. Note that this set of *common nodes* does not necessarily include all subtrees that look alike. For example, consider the trees $A = a(b(\text{treex}), c)$ and $B = a(b, c(\text{treex}))$. RTED could consider 'a', 'b', and 'c' to be common, or it could consider 'a' and 'treex' to be common, but it cannot consider all of them to be common at once, due to conflicting structural constraints. In the former case, for example, 'treex' would be considered to be inserted as the child of node 'b' in tree A and as the child of node 'c' in tree B.

Aligning node positions to accommodate insertions Even if seemingly identical nodes occur in multiple ASTs, we may still need to treat them as different entities. For example, the literal number 1 may occur many times in a given method, at different positions. Thus, our merging algorithm considers two nodes to be equal iff they have both the same content and the same position.

However, insertions and deletions may place nodes that we would like to be equal in different positions in different ASTs. For instance, the position of the node 'x' in AST₁ and AST₂ is (1, 3), but in AST₃ it is (1, 4). This is because the node 'n' is inserted before the node 'x' in the edit list of (AST₁, AST₃) or (AST₂, AST₃). To reduce the number of differences that our algorithm must resolve (and thereby produce less complex merged code in the end), we first *align* the node positions in all trees, based on the edit list.

Procedure AlignPositions**Input:** ASTs \leftarrow input ASTs, $\text{editList}_{x,y} \leftarrow \text{RTED}(\text{AST}_x, \text{AST}_y)$ for each pair of ASTs**Side effect:** position for any node in any AST that was impacted by an insertion

```

1 - Adjusted : (AST, position) set :=  $\emptyset$ 
2 - foreach (ASTx, ASTy)  $\in$  ASTs  $\times$  ASTs where  $x \neq y$ :
5 -   foreach ( $n_a, 0$ )  $\in$  editListx,y:           //  $n_a$  was inserted into ASTx
6 -     if ( $\langle \text{AST}_y, n_a.\text{position} \rangle \notin \text{Adjusted}$ ):
7 -       Adjusted := Adjusted  $\cup \{ \langle \text{AST}_y, n_a.\text{position} \rangle \}$ 
8 -       depth :=  $n_a.\text{position.size}$ 
9 -       foreach node  $\in$  ASTy where
. -         (node.position.size = depth  $\wedge$ 
. -         node.position[depth]  $\geq n_a.\text{position}[\text{depth}]$ ):
10 -        foreach nodesub  $\in \{ \text{node} \} \cup \text{descendants}(\text{node})$ :
11 -          nodesub.position[depth] := nodesub.position[depth] + 1

```

Fig. 4 Outline of the recalculation algorithm to accommodate insertions

The procedure **AlignPositions** (Fig. 4) normalizes the ASTs by finding insertions and shifting the positions of all nodes whose positions will be affected by the insertion to the right, if necessary. **AlignPositions** takes as input the set of ASTs and the edit list of each pair of ASTs, as obtained from RTED.

AlignPositions outputs a data structure **Position**, which is the position list of every node in every AST. **AlignPositions** aligns the position of every node that was impacted by an insertion in some AST.

AlignPositions needs to make sure the recalculation does not happen more than once for the same insertion when comparing one AST with multiple ASTs. Consider the three ASTs $\text{AST}_4 = a(b,c,d)$, $\text{AST}_5 = a(b,c,d)$, and $\text{AST}_6 = a(b,d)$. We see that going from AST_6 to either AST_4 or AST_5 will introduce the node ‘c’ before ‘d’ and thereby shift ‘d’ to the right. If we shift ‘d’ to the right in AST_6 , we thereby align AST_6 with AST_4 and AST_5 . However, since we must align AST_6 with *all* of the other ASTs, we must make sure to only shift ‘d’ once, and not twice (once for AST_4 and once for AST_5). For that purpose we rely on the data structure **Adjusted**, which collects, for each AST, the set of positions for which we have already created a ‘hole’ in that AST.

We now walk through the procedure **AlignPositions** line by line. Line 2 considers every pair of ASTs, AST_x and AST_y . Every entry that has the right (n_b) component as zero implies that the node on the left hand side n_a is inserted into AST_x when migrating from AST_y (Line 5). Line 6 checks the set **Adjusted** to test whether there are already insertions at the position of n_a for AST_y , and aborts if that is the case. Otherwise, Line 7 updates **Adjusted** to make sure that we will consider that position in AST_y only once. Lines 8–11 performs the actual realignment.

Specifically, these lines update the position of every node in AST_y that occupies the position of n_a , a position to the right of n_a , or a position below any of these positions to the right. For example, in AST_1 and AST_2 , the position of the node ‘x’ and the nodes that are part of subtree rooted at ‘x’, i.e ‘z’ are (1, 3) and (1, 3, 1) respectively before recalculation. After recalculation, the positions of node ‘x’ and ‘z’ would be (1, 4) and (1, 4, 1) in both AST_1 and AST_2 , to be consistent with the positions of these nodes in AST_3 . Our merging algorithm then utilises these aligned positions.

4.3 Merge algorithm

Our merge algorithm is split into three high-level steps:

1. Identifying the *conflict nodes* and *merge points*
2. Constructing the merge tree
3. Applying the resolution patterns

4.3.1 Identifying the conflict nodes and merge points

After normalizing the positions of the common nodes shared between all ASTs, our algorithm identifies a *conflict node* as a node at whose position there is a different node in at least one other AST.

Our algorithm begins by collecting the common nodes, as defined previously in Sect. 4.2. In our example, the common nodes are ‘a’, ‘x’, ‘y’, ‘z’. Our algorithm then maps the remaining nodes to their respective source ASTs by building a conflict node table T_{cn} , as shown in Table 1. A node, as we recall, comprises its content and its position. For example, the node with the content ‘d’ and position (1, 1, 2, 1) has the source AST AST_1 . The node with content ‘b’ and position (1, 1) has two source ASTs, AST_1 and AST_2 .

Each position in the table T_{cn} corresponds to a set of conflict nodes. These nodes must be merged in the resulting AST; we will do so later through a special AST node that we call *merge point*. For example, at the position (1, 1), we will place a merge point to merge node ‘b’ from $\{AST_1, AST_2\}$ and node ‘b2’ from $\{AST_3\}$, and we will further place a merge point at position (1, 2, 1) between ‘f1’ from $\{AST_1\}$, ‘f2’ from $\{AST_2\}$ and ‘f3’ from $\{AST_3\}$. Every merge point has a set of one or more choices, with each choice consisting of a conflict node along with the source ASTs that contain the node. For example, the merge point at (1, 1) has two choices. The first choice branches to the node ‘b’ from $\{AST_1, AST_2\}$ and the second choice branches to node ‘b2’ from $\{AST_3\}$. Merge points with one choice can occur only in the case of an insertion.

Table 1 Example table of conflict nodes (T_{cn}) generated by the common difference identification phase

| Position | Content | Source ASTs |
|--------------|---------|----------------|
| (1, 1, 2, 1) | d | AST_1 |
| (1, 1, 2, 1) | e | AST_2 |
| (1, 2, 1) | f1 | AST_1 |
| (1, 2, 1) | f2 | AST_2 |
| (1, 2, 1) | f3 | AST_3 |
| (1, 3) | n | AST_3 |
| (1, 1) | b | AST_1, AST_2 |
| (1, 1) | b2 | AST_3 |
| (1, 1, 1) | c | AST_1, AST_2 |
| (1, 1, 2) | k | AST_1, AST_2 |

4.3.2 Constructing the merge tree

Now that our algorithm has identified the positions of all conflict nodes, it constructs the *merge tree*, which represents the abstraction of the near-miss clone trees that were its inputs. Each identified conflict node position from the previous step allows us to find all affected conflict nodes with their source ASTs. Our algorithm initially creates an empty merge tree. It then places nodes into this tree level by level and for each level, from left to right. The algorithm begins by placing the common nodes in their respective positions. This process uses the positions that we aligned as part of the normalization step in Sect. 4.2.

For every position in T_{cn} , our algorithm places a merge point node, with one choice each for every conflict node in that position along with the node's corresponding source ASTs. Merge points are denoted as $\mathbf{MP}_x(\text{List of sets of ASTs})$, where each set of ASTs inside the merge point represents a choice. For example, for the position (1, 1) discussed previously, the merge point would be $\mathbf{MP}_1(\{1,2\}, \{3\})$, as shown at the bottom of Fig. 3.

Our algorithm does not create merge points for positions where the nodes under consideration are part of a subtree rooted at a previously formed merge point and are part of the same source trees. For example, the node 'c' at position (1, 1, 1) will not be part of a merge point, as 'b' at position (1, 1) is an ancestor to 'c', 'b' is already part of a merge point, and 'c' arises from the same source trees as 'b', i.e., $\{AST_1, AST_2\}$.

4.3.3 Applying resolution patterns

Finally, we eliminate each merge point by applying a *resolution pattern*. A resolution pattern is a code transformation pattern to resolve the merging of specific types of nodes at a given merge point. Recall from the introduction that our approach works on near-miss clone C++ methods. Thus, the resolution patterns in our approach construct a concrete node that corresponds to a C++ code fragment that we insert at the merge point. We use the term *merge-substitution* to describe the algorithm that generates the merged node and inserts it into the merged method. This node generated by the merge-substitution algorithm replaces the merge point in the AST.

Merge-Substitution A *merge-substitution* comprises a *selector* and *selection mechanism*. The selector is the device that the caller of an abstraction uses to choose between the various alternatives. The selection mechanism translates the selector into one of the alternatives within the abstraction, in the body of the called code. To illustrate the concept of selector and selection mechanism, consider the following example code:

```
void rocksdb_writebatch_merge(rocksdb_writebatch_t* b,
    const char* key, size_t klen, const char* val, size_t vlen) {
    b->rep.Merge(Slice(key, klen), Slice(val, vlen));
}

void rocksdb_writebatch_merge_cf(rocksdb_writebatch_t* b,
    rocksdb_column_family_handle_t* column_family,
    const char* key, size_t klen, const char* val, size_t vlen) {
    b->rep.Merge(column_family->rep, Slice(key, klen), Slice(val, vlen));
}
```

and the following merged version of the code:

```

void abstractedFunction(rocksdb_writebatch_t* b,
    rocksdb_column_family_handle_t* column_family,
    const char* key, size_t klen, const char* val, size_t vlen,
    int functionID) {
    if(functionID == 1) {
        b->rep.Merge(Slice(key, klen), Slice(val, vlen));
    }
    else if(functionID == 2) {
        b->rep.Merge(column_family->rep, Slice(key, klen), Slice(val, vlen));
    }
}

void rocksdb_writebatch_merge(rocksdb_writebatch_t* b,
    const char* key, size_t klen, const char* val, size_t vlen) {
    abstractedFunction(b, null, key, klen, val, vlen, 1);
}

void rocksdb_writebatch_merge_cf(rocksdb_writebatch_t* b,
    rocksdb_column_family_handle_t* column_family,
    const char* key, size_t klen, const char* val, size_t vlen) {
    abstractedFunction(b, column_family, key, klen, val, vlen, 2);
}

```

In this merged function `abstractedFunction`, the selector is the parameter `int functionID` added to the function `mergedFunction` and the selection mechanism is the conditional that checks `functionID` to choose which statement to execute. Each caller supplies an *actual selector* value to the *formal selector* variable. In this example, the actual selectors are 1 from the function `rocksdb_writebatch_merge` and 2 from the function `rocksdb_writebatch_merge_cf`.

Table 2 lists the resolution patterns that our prototype supports, in terms of the node types to which they are applicable and the corresponding selector and selection mechanisms. For example, if the nodes under consideration in a particular position are all literals, we can introduce a formal method parameter (selection mechanism) of the type of the literal and pass the literal as an actual method parameter (selector) value. Another possibility, although arguably less elegant, would be to introduce a global variable to act as selector, so that each caller can assign the actual selector value to that variable before calling.

Table 2 Resolution patterns supported by our tool, as options presented to the user

| Type of Node | Selection Mechanism | Selector |
|--------------|--------------------------|----------------------------|
| Statement | Switch on variable | Actual parameter |
| | Conditional on variable | Global variable assignment |
| Literal | Formal parameter | Actual parameter |
| | Global variable | Global variable assignment |
| Type | Template parameter | Actual type parameter |
| Identifier | Formal parameter | Actual parameter |
| | Formal pointer parameter | Actual pointer parameter |

Merging parameters To generate working code, it is not sufficient to merge method bodies; we must also merge the methods' parameter lists. We consider two parameters to be equal if they have the same names. If the parameters disagree on their types or type qualifiers, we introduce a fresh template type parameter that we use as their type. Apart from that, we construct the combined parameter list from the union of the parameters of the merged methods. Whenever the merged methods agree on the parameter order, we preserve the parameter order.

Handling existing call sites Whenever we merge a set of methods f_1, \dots, f_n into a merged method, we do not update call sites that refer to any of f_1, \dots, f_n . Instead, for each f_i , we replace its body by a call to the merged function. For instance, consider the following code:

```
int f1(int x)      { ... }
int f2(int x, bool y) { ... }

int g() {
    return f1(23) + f2(42, true);
}
```

When asked to merge `f1` and `f2`, we might generate a method `fnMerged` through merge-substitution and modify `f1` and `f2` to produce:

```
int fnMerged(int x, bool y, int choice) { ... }
// User choice: selection mechanism is 'parameter'
// Merged parameter list: (int x, bool y)

int f1(int x)      { return fnMerged(x, false /* default value */, 1); }
int f2(int x, bool y) { return fnMerged(x, y, 2); }

int g() {
    return f1(23) + f2(42, true); // unchanged
}
```

The user can now use the *Inline* refactoring (Fowler et al. 1999) to eliminate the methods `f1` and `f2`, or retain these methods, e.g., if external libraries might reference them. This ensures that our approach is safe even when we lack whole-program information. Alternatively, an IDE could offer merge-substitution with automatic inlining, though we have not explored this option in detail.

Below, we discuss the resolution patterns that we have implemented to evaluate our approach, and illustrate them with examples taken from open source projects hosted at GitHub. For each resolution pattern, we describe the merge resolution and the fix-up mechanism. We picked these four patterns based on the dominant kinds of differences that we observed in the samples from our earlier study in Sect. 3. We found these patterns to be sufficient to cover the merges that we had identified for all of these samples. In the examples below, the nodes highlighted in red (light gray in a black and white view) indicate the unique nodes in each function and the nodes highlighted in blue (dark gray in a black and white view) indicate the nodes produced by our merge resolution.

4.3.4 Pattern: switch statement with extra parameter

This resolution pattern can be applied if the nodes to be merged are all statements. This case is typical of Type 3 clones (Saha et al. 2013). We construct the following **switch** statement:

Selection mechanism:

```
switch (choice) {
  case 1: stmt1; break;
  ...
  case k: stmtk; break;
}
```

where *choice* is a formal selector, e.g. a fresh method

parameter, *stmt*_{*i*} is one statement alternative taken from the individual cases of the switch statement, and *i* is a unique number identifying the ASTs in the filtered map. We add the formal selector *choice* as needed, e.g., as global variable or as a formal parameter to the surrounding method or function.

Fix-up: We modify the corresponding call sites to supply their own unique actual selector values to the formal selector.

Example: Consider the function snippets

```
jobject function_openOnly__JLjava(JNIEnv* env, jobject jdb,...) {
  rocksdb::DB* db = nullptr;
  rocksdb::Status s;
  /* About 50 lines of common code */
  s = rocksdb::DB::OpenForReadOnly(*opt, db_path, column_families, &handles, &db);
  return null;
}

jobject function_open__JLjava(JNIEnv* env, jobject jdb,...) {
  rocksdb::DB* db = nullptr;
  rocksdb::Status s;
  /* About 50 lines of common code */
  s = rocksdb::DB::Open(*opt, db_path, column_families, &handles, &db);
  return null;
}
```

Our pattern merges these snippets by introducing a switch statement to choose between the two options. Modulo variable renaming and indentation, resolution produces the following output, with the generated switch statement in lines 14–20:

```
1  jobject function_openOnly__JLjava(JNIEnv* env, jobject jdb,...) {
2      return function_open_Merged__JLjava(env, jdb,..., 1);
3  }
4
5  jobject function_open__JLjava(JNIEnv* env, jobject jdb,...) {
6      return function_open_Merged__JLjava(env, jdb,..., 2);
7  }
8
9  jobject function_open_Merged__JLjava(JNIEnv* env, jobject jdb, ...,
10                                     int openType) {
11      rocksdb::DB* db = nullptr;
12      rocksdb::Status s;
13      /* About 50 lines of common code */
14      switch (openType) {
15      case 1:
16          s = rocksdb::DB::OpenForReadOnly(*opt, db_path, column_families, &handles,&db);
17          break;
18      case 2:
19          s = rocksdb::DB::Open(*opt, db_path, column_families, &handles,&db);
20          break; }
```

```

21     return null;
22 }

```

4.3.5 Pattern: extra parameter for literal expressions

This resolution pattern can be applied if the nodes to be merged all represent literal expressions, i.e., constant values such as 23 or true. We require that all of these constants have the same type. The selection mechanism here is identical to the formal selector: a fresh variable that directly supplies the relevant value. This variable, *value*, can again be a global variable or a fresh parameter that we add as formal parameter to the surrounding method or function. The actual selector would be any constant that has the same type as *value*. If *value* is an ‘int’, then the constant 10 could serve as an actual selector.

Selection mechanism: value

Fix-up: We modify existing call sites to supply their own constants as the actual parameter input.

Example: Consider the following function, taken from the Oracle’s Node-OracleDB project¹⁰:

```

Handle<Value> Connection::GetClientId (Local<String> property,
                                      const AccessorInfo& info) {
    ...
    if (!njsConn->isValid_)
        ...
    else
        msg = NJSMessages::getErrMsg(errWriteOnly, "clientId");
    NJS_SET_EXCEPTION(msg.c_str(), (int) msg.length());
    return Undefined();
}

Handle<Value> Connection::GetModule (Local<String> property,
                                    const AccessorInfo& info) {
    ...
    if (!njsConn->isValid_)
        ...
    else
        msg = NJSMessages::getErrMsg(errWriteOnly, "module");
    NJS_SET_EXCEPTION(msg.c_str(), (int) msg.length());
    return Undefined();
}

Handle<Value> Connection::GetAction (Local<String> property,
                                    const AccessorInfo& info) {
    ...
    if (!njsConn->isValid_)
        ...
    else
        msg = NJSMessages::getErrMsg(errWriteOnly, "action");
    NJS_SET_EXCEPTION(msg.c_str(), (int) msg.length());
    return Undefined();
}

```

¹⁰ <https://github.com/oracle/node-oracledb/>

Our tool would identify that the calls to `getClientId`, `getModule` and `getAction` are mergeable using an extra parameter. Modulo variable renaming and indentation, this produces the following output:

```

1  Handle<Value> Connection::GetProperty(Local<String> property,
2                                     const AccessorInfo& info,
3                                     string errorMsg)
4  {
5      ...
6      if (!njsConn->isValid_)
7          ...
8      else
9          msg = NJSMessages::getErrMsg(errWriteOnly, errorMsg);
10     NJS_SET_EXCEPTION(msg.c_str(), (int) msg.length());
11     return Undefined();
12 }
13
14 Handle<Value> Connection::GetClientId(Local<String> property,
15                                     const AccessorInfo& info)
16 {
17     return Connection::GetProperty(property, info, "clientId");
18 }
19
20 /* The methods GetModule and GetAction are analogous to GetClientId */

```

4.3.6 Pattern: templates for type expressions

We can apply this resolution pattern if the nodes to be merged all represent types. We introduce a fresh variable for a template, *type*. We also convert the method into a template method if it is not already one.

Selection mechanism: type. We also introduce a new formal template type parameter (selection mechanism) *type* to the function definition. Any type (int, char, etc.) would be a valid selector.

Consider the following functions taken from the RethinkDB project¹¹:

```

cJSON *cJSON_CreateIntArray( int *numbers, int count) {
    ...
    for (int i=0; i<count; i++) {
        ...
    }
    a->tail = p;
    return a;
}

cJSON *cJSON_CreateDoubleArray( double *numbers, int count) {
    ...
    for (int i=0; i<count; i++) {
        ...
    }
    a->tail = p;
    return a;
}

```

Our tool would identify that we can merge the definitions of `CreateIntArray` and `CreateDoubleArray` by introducing a template type parameter. Modulo variable renaming and indentation, our tool produces the following output:

¹¹ <https://github.com/rethinkdb/rethinkdb/>

```

template<typename T> cJSON *cJSON_CreateNumArray( T *numbers, int count) {
    ...
    for (int i=0; i < count; i++) {
        ...
    }
    a->tail = p;
    return a;
}

cJSON *cJSON_CreateIntArray(int *numbers, int count) {
    return cJSON_CreateNumArray<int>(numbers, count);
}

cJSON *cJSON_CreateDoubleArray(double *numbers, int count) {
    return cJSON_CreateNumArray<double>(numbers, count);
}

```

4.3.7 Pattern: extra parameter for identifiers

This resolution pattern can be applied if the nodes to be merged are all variable identifiers (identifier expression nodes). We require that all of the variables be of the same type. Again the selection mechanism is the same as the formal selector and passed through a fresh global variable or parameter, *value*.

Selection mechanism: value

The resolution here is very similar to the pattern for literals, except that our algorithm promotes L-values to pointer-typed parameters whenever required. We opted for pointers instead of references to allow our approach to also work on C code. After promotion, the formal selector variable *value* has the type t^* if the actual selector is a variable of type t .

Consider the following example:

```

int x, z;
void fn1() {
    int y = x + 1 + 25;
    x = 10;
}
void fn2() {
    int y = z + 1 + 45;
    z = 10;
}

```

Our algorithm handles this case by identifying, among other merge points, two different merge points each for the identifiers x and z . Our algorithm creates a pointer parameter to switch between x and z , and passes references. We add *value* as an additional formal parameter whose type is the type ‘pointer to the type of the identifiers being merged’. A merged version of the functions would look like this:


```
int x, z;
void fnMerged(int *ptr, int constant) {
    int y = *ptr + 1 + constant;
    *ptr = 10;
}
void fn1() {
    fnMerged(&x, 25);
}
void fn2() {
    fnMerged(&z, 45);
}
```

Fix-up: We modify the corresponding call sites to supply (the addresses of) their own identifiers as actual parameters.

Example: Consider these function snippets from Facebook's HHVM project¹²:

```
Type typeDiv(Type t1, Type t2) {
    if (auto t = eval_const_divmod(t1, t2, cellDiv))
        return *t;
    return TInitPrim;
}

Type typeMod(Type t1, Type t2) {
    if (auto t = eval_const_divmod(t1, t2, cellMod))
        return *t;
    return TInitPrim;
}
```

Our tool would identify that the functions `typeDiv` and `typeMod` can be merged by introducing an extra parameter. Modulo variable renaming and indentation, this produces the following output:

```
template<class CellOp> Type typeModDiv(Type t1, Type t2, CellOp fun) {
    if (auto t = eval_const_divmod(t1, t2, fun))
        return *t;
    return TInitPrim;
}

Type typeDiv(Type t1, Type t2) { return typeModDiv(t1, t2, cellDiv); }
Type typeMod(Type t1, Type t2) { return typeModDiv(t1, t2, cellMod); }
```

4.4 Merging identifiers

C++ permits nested scopes to introduce multiple variables of the same name. Any transformation that alters the scope or the name of such a variable risks introducing *name capture*, where the use of one variable incorrectly references a variable declared at a different point in the program, just because both variables had or now have the same name:

```
int x = 1; // t1 (binding location)
int y = 3; // t3 (binding location)

int f1(void)
{
```

¹² <https://github.com/facebook/hhvm/>

```

    /* Common code */
    print(x);    // t1
}

int f2 ( void )
{
    // The declaration of x here is inserted when considering the
    //edit list with f1
    int x = 2; // t2 (binding location)
    /* Common code */
    print(x);    // t2
}

int f3 ( void )
{
    /* Common code */
    print(y);    // t3
}

```

Here, we cannot directly merge the `print(x)` statements in `f1` and `f2`, as they refer to different variables (identified by their labels `t1`, `t2`, `t3`...). However, we *can* merge functions `f1` and `f3`, since `f1` is the same as `f3`, except for the identifier parameter of the `print` function call.

Our algorithm ensures that we do not introduce accidental name capture through the following check:

A clone group can only be merged if for every pair of ASTs in the clone group, there are no variable declarations as part of an inserted node when considering their edit lists (Sect. 4.2). For example, the merge between `f1` and `f2` would be rejected since the declaration of `x` in `f2` stems from an inserted statement (`int x = 2;`).

4.5 Optimizations

Although our core algorithm is sufficiently generic to handle many forms of merging, our algorithm contains a few optimizations to improve the end result.

Going up the parent node

Not all kinds of differences can be resolved at the level at which they occur. Consider the following pieces of code:

```

x = y + z; //A
x = y - z; //B

```

Even though the only difference is the operator in the binary expression on the right hand side of the assignment, our algorithm currently supports no resolution pattern that can resolve such a case directly. Instead, our algorithm goes up the chain of parents in the AST until it reaches a node at which a resolution pattern applies.

In our example, our algorithm would move up one parent level in both the clones. At this point, our algorithm would end up with two binary expressions, for which there is still no resolution pattern. Our algorithm would then go up one more level, reaching the assignment statement, which it can resolve through pattern *Switch Statement with Extra Parameter* (Sect. 4.3.4). This pattern also serves as a general fallback, since it can

abstract most statement-level differences, excluding only those that might introduce name capture, cf. Sect. 4.4.

In our earlier study of near-miss clones, we encountered only two kinds of expressions as differences, namely literals and identifiers. Our implementation therefore currently provides specific support for only these two kinds of differences.

Sequence of line differences

Since our algorithm operates at the AST level, it is oblivious of concrete syntactic information. Consider the following near-miss clones.

| | |
|----------------------------|----------------------------|
| <i>//Clone 1</i> | <i>//Clone 2</i> |
| <code>common1();</code> | <code>common1();</code> |
| <code>statement1();</code> | <code>statement3();</code> |
| <code>statement2();</code> | <code>statement4();</code> |
| <code>common2();</code> | <code>common2();</code> |

Our algorithm would identify two merge points, one for the difference between `statement1` and `statement3`, and one for the difference between `statement2` and `statement4`. Applying the statement level resolution pattern twice would result in two conditionals, or two switch statements, both of which always behave in the same way. Our algorithm performs an optimization to resolve such contiguous differences and treat them as one block to avoid multiple resolutions for contiguous statements. Our algorithm considers adjacent siblings of the same block that have a merge point at the statement level as contiguous.

Format Strings

C-style formatted printing, such as `printf("value is %d", 10)`, relies on format strings such as `"string %s %d"`. The meaning of these strings relies on additional parameters to the printing operation (such as 10, in the example). For this reason, we have introduced a heuristic that disallows abstracting over format strings as a value with pattern *Extra Parameter for Literal Expressions*. With our current set of patterns, this means that differences in format strings will make our algorithm fall back to *Switch Statement with Extra Parameter*.

4.6 Beyond method-level merging

Although our merging approach currently operates only on the method level, we are aware that clones happen at many levels (class, submethod, etc.). It would be straightforward to adapt our approach to submethod-level merging, by combining our algorithm with an automated ‘Extract Method’ refactoring (Fowler et al. 1999). Abstracting on the class level opens new opportunities for selectors (e.g., the dynamic type of an object) and selection mechanisms (e.g., dynamic dispatch), and we expect that our work can be extended in different dimensions, depending on the abstraction mechanisms provided by the target language.

5 Implementation

In this section, we give an overview of how we have implemented our approach and how developers can use and extend the implemented tool. We begin by discussing the libraries and frameworks we used and adapted in order to implement our tool, and then discuss our tool's public availability.

5.1 Libraries and frameworks used in our implementation

We have adapted an existing implementation of RTED¹³ to fit our CDT AST representation. The existing implementation operated on in-order representations of trees in which nodes are labeled with strings. We adjusted the representation of nodes to contain information about AST node types and content.

Our merging tool is accessible to the user as an Eclipse plug-in via the Refactoring menu. Our tool presents all the functions in the file in the currently active window using an input selection form. The requirement for all clones to be in the same file is a limitation of the current state of the implementation. The user marks the near-miss clone methods to abstract using our tool's input form. The merging tool then produces a merged function and replaces the bodies of the existing functions with calls that invoke the merged function with appropriate arguments.

We have implemented the distance calculator, the algorithm and the framework on top of Eclipse CDT.¹⁴

5.2 Availability

We have made our prototype publicly available.¹⁵ From October 2015 to November 2017, the version of our prototype that is available as a plugin in the Eclipse Marketplace has had 192 click-throughs and 70 installs, with no reported installation failures.¹⁶

6 Correctness

Our algorithm has the effect of shifting the position at which various terms are placed in the source code, and such changes may, in general, have an impact on the values the terms produce and the side-effects they cause. In this section, we review the transformations performed by our approach and discuss the possible correctness issues.

¹³ <http://www.inf.unibz.it/dis/projects/tree-edit-distance/download.php>

¹⁴ <https://eclipse.org/cdt/>

¹⁵ <http://sepl.cs.uni-frankfurt.de/~krishnanm86/clonemergeindex.html>

¹⁶ <https://marketplace.eclipse.org/content/clone-abstractor-c-methods-0/metrics>

6.1 Statements

Our approach moves statements into the branches of a conditional or switch statement. By specializing the merged function with respect to each of the possible selector values, it is easy to see that the statements are executed the same number of times as in the original code, and have access to the same set of variables and their values. The scope of any local variables declared by the moved statements, however, is reduced to the conditional or switch branch; this can happen, e.g., if the developer clones a method and changes the type of a local variable. If the scope of a declaration is shrunk in this fashion, uses of that variable can be stranded outside, leaving to ‘undefined identifier’ errors or name capture. To detect such problems, we track all variable declarations that have been stranded into their own local scopes as part of the merge process and abort the merge if any such declaration exists (Sect. 4.4). We have not observed this issue in our experiments.

6.2 Literals

Our approach moves literals from the places where they are used in the clone instances (input methods) to the call site of the merged function. A literal, by definition, evaluates to itself, independently of the context in which it occurs. If we choose to pass the literal value to the merged function via a new function parameter, we furthermore have the property that the parameter introduced by our approach to hold the value of the literal is not modified within the merged function. Thus, the parameter’s value is the same as that of the literal that it replaces, wherever it occurs.

6.3 Types

Our approach moves types into C++ template arguments. We choose unique names for the template parameters and template parameters are not updateable, so at each usage context the intended type is preserved. A limitation of this approach is that C++ does not allow us to pass `void` as a type parameter, so if the user wishes to merge `void` functions with non-`void` functions, the `void` functions must be promoted to non-`void` functions that return a dummy value first. We observed the need to handle such a scenario only once during our experiments and we handled it manually in less than 4 min of effort (Sect. 8).

Another limitation are type incompatibilities introduced by templatisation of merged identifiers. Consider merging two methods that take a parameter `x` and pass it to a method `print_string(std::string)` (in one case) or to a method `print_int(int)` (in the other case). Using our resolution patterns, these methods’ ‘straightforward’ merge would be:

```
template <class T>
void merged(int selector) {
    T x;
    switch (selector) {
    case 0:
        print_string(x); // type error if T = int
        break;
```

```

    case 1:
        print_int(x);          // type error if T = std::string
        break;
    }
}

```

However, as the comments note, this code cannot typecheck. Specifically, a variable must not be required to have incompatible types in different AST subtrees. We guarantee this property by ensuring that newly-templatised variables only occur in common AST nodes, but never below merge points. We reject the merge otherwise.

6.4 Global variables as selectors

We permit the use of fresh global variables as selectors. However, this is unsafe in the presence of recursive functions: consider a scenario in which a merged function calls itself recursively before performing a computation that depends on its global selector variable. The recursive call may update the global selector, thereby altering how the remaining computation is performed after the recursive call is over.

One way to address this correctness problem would be to copy the global variable into a local temporary variable. However, this would decrease readability, and would thus be best avoided unless necessary (e.g., using a static analysis to detect if recursion is impossible). In our current implementation, we do not perform this transformation.

6.5 Identifiers

When two cloned functions disagree on a nonlocal variable that occurs on the left-hand side of an assignment, our algorithm abstracts over this variable by passing it by reference (Sect. 4.3.7). Our implementation relies on the CDT API's `isLValue` method to determine if an identifier is used in an assignment; this also covers increment expressions and equal-to expressions, among other kinds of writes.

7 Pull requests and user study

We have evaluated our approach by exploring the following research question:

RQ: Are the abstractions performed by our algorithm of sufficient quality for production level code?

In order to evaluate this question, we first looked for clone group candidates to merge. We explored top trending GitHub repositories, identified potential candidates for merging using our RTED-inspired clone detector, and used our approach to abstract the identified candidates. We finally submitted the abstracted code back to the developers through pull requests, to see how many of them were of sufficient quality to be introduced back into production code. We performed a total of 18 abstractions of clone groups from the top trending GitHub repositories that we identified previously and sent pull requests to the repositories from which we got the code. Table 3 lists the repositories that we considered in our evaluation along with our pull request URLs, the number of clone groups abstracted per repository, and the status of the pull requests.

7.1 Identifying and merging clone groups

The clone group candidates for our approach are those with near-miss clones. We started with the repositories in Table 3 and collected all method pairs belonging to the same source file. We began by computing the edit distance of each pair. In the previous section (Sect. 3), we defined a function pair a near-miss clone function pair if the number of nodes in the larger of the two functions ($\#fnBigger$) is greater than a customizable $threshold_n$ and if the ratio of the edit distance to $\#fnBigger$ was less than a customizable $threshold_r$, where $threshold_n$ and $threshold_r$ are positive numbers.

We collect the near-miss clone function pairs into sets such that every function in each set forms a near-miss clone function pair with every other function inside the set. We call such sets of methods whose bodies are closely related to each other ‘clone groups’. We then randomly picked clone groups. Each clone group we picked contained 2–4 functions. We then merged the clone groups, using a predetermined resolution pattern for each node type, and submitted pull requests. We chose the following resolution patterns for specific node type differences:

- We resolved differences in statements using a switch and an extra method parameter as a selector specifying the switch branch to choose (Pattern: *Switch Statement with Extra Parameter*). We could have used the conditional pattern to accomplish the same effect.
- We resolved differences in literal expressions (constants) by passing additional parameters (Pattern: *Extra Parameter for Literal Expressions*). We could have chosen to use a global variable, but we believe that using an extra parameters is less intrusive to the existing code and is more likely to be preferred by the maintainers of the repositories accepting the pull requests.
- We resolved differences in types using templates (Pattern: *Templates for Type Expressions*).
- We resolved differences in identifier references using additional parameters (promoted to pointers if the identifiers occurred as LValues), and formal parameters specifying the identifier or the address of the variable (Pattern: *Extra Parameter for Identifier*).

We also performed minor manual changes. These include:

- Providing meaningful names for parameters. Our tool generates random fresh names based on the position of the merge points. These names are not suitable for production code.
- We added function prototypes to header files whenever doing so was preferred by maintainers.

We added the function prototypes after discussion with the maintainers who had previously looked at our tool generated merges. These manual changes are standard refactorings that are not central to our approach and could be automated (Raychev et al. 2015).

Table 3 Repositories with their pull request URLs

| Repository | Phase | Clone groups | Status |
|---|-------|--------------|-----------------|
| oracle/node-oracledb | 2 | 3 | Accepted |
| https://github.com/oracle/node-oracledb/pull/28 | | | |
| mongodb/mongo | 2 | 2 | Accepted |
| https://github.com/mongodb/mongo/pull/927 | | | |
| | | | Accepted |
| https://github.com/mongodb/mongo/pull/928 | | | |
| rethinkdb/rethinkdb | 2 | 2 | Accepted |
| https://github.com/rethinkdb/rethinkdb/pull/3820 | | | |
| https://github.com/rethinkdb/rethinkdb/pull/3818 | | | Accepted |
| cocos2d/cocos2d-x | 2 | 2 | Accepted |
| https://github.com/cocos2d/cocos2d-x/pull/10539 | | | |
| https://github.com/cocos2d/cocos2d-x/pull/10546 | | | Accepted |
| ideawu/ssdb | 2 | 1 | Rejected |
| https://github.com/ideawu/ssdb/pull/609 | | | |
| facebook/rocksdb | 1 | 1 | Pending |
| https://github.com/facebook/rocksdb/pull/440/ | | | |
| openexr/openexr | 1 | 3 | Pending |
| https://github.com/openexr/openexr/pull/147 | | | |
| facebook/hhvm | 1 | 1 | Rejected |
| https://github.com/facebook/hhvm/pull/4490 | | | |
| google/protobuf | 1 | 2 | Accepted |
| https://github.com/google/protobuf/pull/128 | | | |
| https://github.com/google/protobuf/pull/126 | | | Rejected |
| SFTtech/openage | 1 | 1 | Rejected |
| https://github.com/SFTtech/openage/pull/176 | | | |

Each clone group represents one abstraction. We encourage readers to go through the comments associated with the pull requests. While some of the pull requests do not explicitly have their status listed as ‘merged’ in GitHub, as with the OracleDB and the MongoDB repositories, the code has actually been merged into their existing code-bases outside of GitHub, as indicated by the maintainer comments

7.2 Results

Our evaluation involved two phases. The first phase served as a validation to show that our tool can abstract near-miss clones in real code. The first phase also gave us information about what resolution patterns developers prefer. We used the insights from the first phase in the second phase to focus our efforts on clone groups and abstraction patterns that are of greater interest to developers. The second phase of our evaluation illustrated the industry acceptability of the abstractions produced our tool.

We performed our initial evaluation (Phase 1) using an early version of our merging tool that could perform only merges of pairs of methods and did not support multiple resolution patterns for the same pair, i.e if the functions had more than one merge point, all of these merge points had to use the same resolution pattern. During Phase 1,

Table 4 Phase 1 and 2 results summary

| Phase | Submitted | Accepted | Rejected | Pending |
|-------|-----------|----------|----------|---------|
| 1 | 8 | 1 | 3 | 4 |
| 2 | 10 | 9 | 1 | 0 |

we ran our distance calculator on the top trending C++ repositories in GitHub for the month of December 2014, and selected potential clone groups by setting $threshold_r$ to 0.5 and $threshold_n$ to 0, meaning that we considered functions of all sizes. We submitted 8 abstractions as pull requests and only one of the clone groups was **Accepted**. The results of the pull requests highlighted areas of improvement needed in our first prototype (Table 4).

We performed our second evaluation (Phase 2) using an improved version of the our merging tool, capable of merging an arbitrary number of methods at the same time. This version also supported resolving multiple merge points with different resolutions for each merge point. During Phase 2, we ran our distance calculator on the top trending repositories for the month of February 2015. We set $threshold_r$ to 0.15 and $threshold_n$ to 100, to focus on clone groups involving functions that are more similar and have a larger size than required in Phase 1. We changed the thresholds in order to focus on clone groups that would save more lines of code when abstracted. The clones in the Phase 2 were very similar to each other and tied to methods of substantial size. We then submitted 10 abstractions as pull requests, summarized in Table 4, and found that all but one were **Accepted**.

We conclude that the repository maintainers found our code to be of sufficient quality (including readability and maintainability) for inclusion. Specifically, we observed no negative comments regarding readability in any of the comments that we received.

7.3 Analysis of rejected and pending results

We present the results of the pending and rejected pull requests summarized in Table 3 and provide our analysis of the these results.

7.3.1 Pending results

We begin with the feedback to pull requests that were neither **Accepted** nor Rejected. Let us first discuss the pending pull request from RocksDB. The comment from the head maintainer of the project was:

Great stuff, now its only one commit (after the squash)! Waiting for OK from @anon1 or @anon2 (since they maintain this code) before merging.

We interpret that the pull request was met with positive review. We did check later with the maintainers of the repository to no avail. We suspect that developers have many tasks and only one of them is attending to pull requests; our patch may not have been their top priority.

Table 5 Activity of repositories as illustrated by number of commits in November 2017

| Repository | # of commits |
|-----------------|--------------|
| Oracle-NodeDB | 49 |
| MongoDB | 443 |
| RethinkDB | 4 |
| Cocos-2D | 7 |
| Google-protobuf | 17 |

The other pending pull request is from the OpenExr repository. The request merged three clone groups at once, and received a mixture of responses. One maintainer requested an explanation of the advantages. Another maintainer expressed skepticism over the performance overhead of such an abstraction, as it was a low level function. A third maintainer requested a unit test of the introduced abstraction before a merge. We could not satisfy these requests due to a lack of understanding of the semantics of the functions we had merged. Indeed, generating unit tests is out of the scope of our work, but has been the subject of much recent research (Fraser and Zeller 2010). All these exchanges took place over a 3 month period.

7.3.2 Rejected results

Of the five rejected clone group abstractions, four were rejected because the maintainers felt that not enough lines were saved. We did not receive an explanation for the rejected clone group abstraction for the ideawu/ssdb repository.

7.3.3 Behavior preservation

In order to validate if the refactored code broke the behavior of the repositories, we ran unit tests after the application of our tool on the selected clone groups, whenever possible. Out of the ten repositories that we submitted to, two had automatic unit tests that ran on our pull requests without observing any errors. In addition, we explored the unit test suites shipped with the software. For two of the projects, Google-protobuf and Cocos-2d, we found test suites that we were able to build and run and that exercised our merged code (which we verified by instrumenting the code); neither test suite observed any errors.

Since the automatic pre-pull request unit testing and the manual unit-testing post-merge did not cover all the accepted repositories, we also checked whether our changes had survived in the projects' code base between the pull request submission in December 2014 (for Phase 1, or February 2015 for Phase 2) and November 2017. We observed that all repositories that accepted our pull requests were very active (Table 5). All but two of the merged methods were still part of the active repositories, indicating that the developers had not found a reason to remove them in the 35 months (for Phase 1, or 33 months for Phase 2) since we first submitted the requests. In both of the cases where the merged method was removed, we found that its removal was part of a larger-scale refactoring that (to the best of our understanding) was unconnected to our changes.

7.4 Comparison of our clone detection method against NiCad

Our evaluation relied on a custom clone detector for two purposes:

1. to motivate our resolution patterns (Sect. 3).
2. to detect clone groups to merge as part of our pull requests (Sect. 7).

While neither of these uses of clone detection are relevant to the clone-merging functionality that is our central contribution, they raise the question of how our custom clone detector compares to the state of the art in clone detection. We therefore compared our custom clone detector against NiCad (Cordy and Roy 2011), a popular clone detection tool. We ran both tools on C code, to avoid NiCad's limitations, using the Bellon benchmark.¹⁷ We configured the systems to use the same detection thresholds, though the method by which they compute these thresholds differs: NiCad counts differences and thresholds in lines of code, while our system counts AST nodes. We set both systems to a detection threshold of 30%, the maximum number of differing AST nodes in our detector to 100, and the maximum number of differing lines to 10 in NiCad. NiCad detected 1382 clone pairs, while our approach detected 2042 clone pairs, including 987 (more than 70%) of the clone pairs reported by NiCad. We hypothesize that the difference in the results primarily stems from differences in accounting for thresholds (i.e., lines of code vs. number of AST nodes).

8 Full repository evaluation: GIT

While our earlier two sets of experiments illustrated the utility that our tool provides in realistic scenarios, we biased our selection through the use of a clone detector whose similarity metric is closely related to our merging algorithm. To explore whether this bias is a concern in practice, we ran a third experiment with a mainstream off-the-shelf clone detector. Since we were not aware of any method-level clone detector for C++, we targeted our experiment to C code, allowing us to use NiCad (Cordy and Roy 2011) as a clone detector. Since our system is based on the Eclipse CDT, we can also use it on C code, as long as we disable abstraction patterns that require C++-only language features.

As target program we therefore selected one of the top trending C repositories on github, the Git¹⁸ revision control system. At the time of our experiment, Git had a total of 6251 functions. We configured NiCad for our experiment as follows:

Granularity: functions

Max difference threshold: 30%

Clone size: 20–2500 lines

NiCad detected 5 clone groups. In the following, we describe each clone group, as well as the results of merging the functions in each of these clone groups, and the insights that we obtained from each merge.

¹⁷ <http://www.softwareclones.org/research-data.php>

¹⁸ <https://github.com/git/git>

Clone Group 1

The first clone group contained two functions, namely `int_obstack_begin_1` and `int_obstack_begin`, that differed by one constant (Line 11 in `int_obstack_begin_1` and Line 9 in `int_obstack_begin`), one extra argument `arg` in `int_obstack_begin_1` and one statement that was present only in `int_obstack_begin_1` (Line 11):

```

1  int _obstack_begin_1
2      ( struct obstack *h, int size, int alignment,
3          void *(*chunkfun) (void *, long),
4          void (*freefun) (void *, void *),
5          void *arg)
6  {
7      /*Common Lines */
8
9      h->alignment_mask = alignment - 1;
10     h->extra_arg = arg;
11     h->use_extra_arg = 1;
12
13     chunk = h->chunk = CALL_CHUNKFUN(h, h->chunk_size);
14
15     /*Common Lines */
16 }

1  int _obstack_begin
2      ( struct obstack *h, int size, int alignment,
3          void *(*chunkfun) (void *, long),
4          void (*freefun) (void *, void *))
5  {
6      /*Common Lines */
7
8      h->alignment_mask = alignment - 1;
9      h->use_extra_arg = 0;
10
11     chunk = h->chunk = CALL_CHUNKFUN(h, h->chunk_size);
12
13     /*Common Lines */
14 }

```

Our tool resolved the difference in the constant values by introducing a new parameter `functionId` as formal selector for the switch statement and an additional parameter `parameter`, and using it in place of the constants. It further resolves the optional statement by introducing a switch statement around the optional line (`h->extra_arg = arg;`). It also supplied a null value to the extra parameter `arg` in the call from the `int_obstack_begin`, where the argument did not exist

```

int
_obstack_begin_merged (struct obstack *h, int size, int alignment,
    void *(*chunkfun) (void *, long),
    void (*freefun) (void *, void *),
    void *arg, int functionId, int parameter)
{
    /*Common Lines */

    h->alignment_mask = alignment - 1;
    //Generated Switch statement
    switch(functionId)
    {
        case 1:
            h->extra_arg = arg;
            break;
    }
}

```



```
h->use_extra_arg = parameter; //Generated extra parameter

chunk = h->chunk = CALL_CHUNKFUN (h, h -> chunk_size);

/*Common Lines */
}

int_obstack_begin_1(struct obstack *h, int size, int alignment,
    void *(*chunkfun) (void *, long),
    void (*freefun) (void *, void *),
    void *arg)
{
    _obstack_begin_merged(h, size, alignment, chunkfun, freefun, arg, 1, 1);
}

int_obstack_begin(struct obstack *h, int size, int alignment,
    void *(*chunkfun) (void *, long),
    void (*freefun) (void *, void *))
{
    _obstack_begin_merged(h, size, alignment, chunkfun, freefun, null, 2, 0);
}
```

Insights In this example, the values of `parameter` and `functionId` depend on each other, meaning that the two parameters could be merged into one. We envision that a future version of our tool can re-use selectors in multiple selection mechanisms.

Clone Group 2

The second clone group contains statement-level differences, which our tool merged using switch statements. There is also a constant difference, for which our tool added an extra parameter.

```
static void command_loop(int input_fd, int output_fd) {
    char buffer[MAXCOMMAND];
    while (1) {
        size_t i;
        if (!fgets(buffer, MAXCOMMAND - 1, stdin)) {
            if (ferror(stdin))
                die("Input error");
            return;
        }
        i = strlen(buffer);
        while (i > 0 && isspace(buffer[i - 1]))
            buffer[--i] = 0;
        if (!strcmp(buffer, "capabilities")) {
            printf("*connect\n\n");
            fflush(stdout);
        } else if (!strncmp(buffer, "connect", 8)) {
            printf("\n");
            fflush(stdout);
            if (bidirectional_transfer_loop(input_fd, output_fd))
                die("Copying data between descriptors failed");
            return;
        } else {
            die("Bad command: %s", buffer);
        }
    }
}

static int command_loop(const char * child) {
    char buffer[MAXCOMMAND];
    while (1) {
        size_t i;
        if (!fgets(buffer, MAXCOMMAND - 1, stdin)) {
```

```

        if (ferror(stdin)) die("Command input error");
        exit(0);
    } /* Strip end of line characters. */
    i = strlen(buffer);
    while (i > 0 && isspace(buffer[i - 1])) buffer[--i] = 0;
    if (!strcmp(buffer, "capabilities")) {
        printf("*connect\n\n");
        fflush(stdout);
    } else if (!strncmp(buffer, "connect", 8)) {
        printf("\n");
        fflush(stdout);
        return run_child(child, buffer + 8);
    } else {
        fprintf(stderr, "Bad command");
        return 1;
    }
}
}

```

Our tool also created a union of the parameters of the two functions, mapping the arguments to the ones in the merged function appropriately and passing dummy values otherwise. The near-miss clone functions before our transformations have 2 parameters and 1 parameter, respectively, as in the original code, and the merged function after transformation has 5 parameters, comprising the 3 original ones and the 2 added selectors.

```

static int command_loop_merge(int input_fd, int output_fd, const
                             char * child,
                             char * str1, int functionId) {
    char buffer[MAXCOMMAND];

    while (1) {
        size_t i;
        if (!fgets(buffer, MAXCOMMAND - 1, stdin)) {
            if (ferror(stdin))
                die(str1);
            switch (functionId) {
                case 1:
                    return 1;
                case 2:
                    exit(0);
            }
        }
        /* Strip end of line characters. */
        i = strlen(buffer);
        while (i > 0 && isspace(buffer[i - 1]))
            buffer[--i] = 0;

        if (!strcmp(buffer, "capabilities")) {
            printf("*connect\n\n");
            fflush(stdout);
        } else if (!strncmp(buffer, "connect", 8)) {
            printf("\n");
            fflush(stdout);
            switch (functionId) {
                case 1:
                    if (bidirectional_transfer_loop(input_fd,
                                                    output_fd))
                        die("Copying data between file descriptors
                            failed");
                    return;
                break;
            }
        }
    }
}

```

```
        case 2:
            return run_child(child, buffer + 8);
            break;
    }

    } else {
        switch (functionId) {
            case 1:
                die("Bad command: %s", buffer);
                break;
            case 2:
                fprintf(stderr, "Bad command");
                return 1;
                break;
        }
    }
}

static void command_loop(int input_fd, int output_fd) {
    command_loop_merge(input_fd, output_fd, null, "Input error", 1);
}

static int command_loop(const char * child) {
    return command_loop_merge(0, 0, child, "Command input error", 2);
}
```

Insights Since the return types of the functions in the clone group are different and C does not support templates, we had to manually modify the code so that both functions have the same return type. Transforming a void function so that it returns an integer only requires adding a dummy return value, so we took this option. The manual effort for performing this transformation took about 4 minutes.

The methods also existed in different files. We had to manually merge them in a separate file, introduce the merged method in a common file that was included by both the files. The overall manual effort for the process did not take more than 5 minutes and it did not cause any compilation or test issues.

We also observed that our tool is unable to detect commonalities and differences inside strings. For example, when generating the calls to the merged function, we would have preferred to only pass the strings “Command input” and “Input”, instead of the strings “Command Input Error” and “Input Error”. Such reuse would have required us to introduce additional function call statements or formatted prints. Currently, our prototype does not support this form of merging.

Clone Group 3

The third clone group contains 6 statement level differences.

```
static int keyring_get(struct credential *c)
{
    char *object = NULL;
    GList *entries;
    GnomeKeyringNetworkPasswordData *password_data;
    GnomeKeyringResult result;

    if (!c->protocol || !(c->host || c->path))
        return EXIT_FAILURE;

    /* Common Lines */
}
```

```

/* pick the first one from the list */
password_data = (GnomeKeyringNetworkPasswordData *)entries->data;
gnome_keyring_memory_free(c->password);
c->password = gnome_keyring_memory_strdup(password_data->password);
if (!c->username)
    c->username = g_strdup(password_data->user);
gnome_keyring_network_password_list_free(entries);

return EXIT_SUCCESS;
}

static int keyring_erase(struct credential *c)
{
    char *object = NULL;
    GList *entries;
    GnomeKeyringNetworkPasswordData *password_data;
    GnomeKeyringResult result;

    if (!c->protocol && !c->host && !c->path && !c->username)
        return EXIT_FAILURE;

    /* Common Lines */

    /* pick the first one from the list (delete all matches?) */
    password_data = (GnomeKeyringNetworkPasswordData *)entries->data;
    result = gnome_keyring_item_delete_sync(
        password_data->keyring, password_data->item_id);
    gnome_keyring_network_password_list_free(entries);
    if (result != GNOME_KEYRING_RESULT_OK) {
        g_critical("%s", gnome_keyring_result_to_message(result));
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}

```

Our tool merged the differences using switch statements. The parameter lists match, with both near-miss clone functions containing one parameter, of the same type, and so the abstracted method contains only one extra parameter, which allows it to switch between the differences in the two cloned functions.

```

static int keyring_get_merge(struct credential *c, int functionId) {
    char *object = NULL;
    GList *entries;
    GnomeKeyringNetworkPasswordData *password_data;
    GnomeKeyringResult result;
    switch (functionId) {
        case 1:
            if (!c->protocol || !(c->host || c->path))
                return EXIT_FAILURE;
            break;
        case 2:
            if (!c->protocol && !c->host && !c->path && !c->username)
                return EXIT_FAILURE;
            break;
    }
    /* Common Lines */

    switch (functionId) {
        case 1:
            gnome_keyring_memory_free(c->password);
            c->password = gnome_keyring_memory_strdup(password_data->
                password);

            if (!c->username)
                c->username = g_strdup(password_data->user);

```

```

        break;
    case 2:
        result = gnome_keyring_item_delete_sync(
            password_data->keyring, password_data->item_id);

        gnome_keyring_network_password_list_free(entries);

        if (result != GNOME_KEYRING_RESULT_OK) {
            g_critical("%s", gnome_keyring_result_to_message(
                result));
            return EXIT_FAILURE;
        }
        break;
    }
    return EXIT_SUCCESS;
}

static int keyring_get(struct credential *c) {
    return keyring_get_merge(c, 1);
}

static int keyring_erase(struct credential *c) {
    return keyring_get_merge(c, 2);
}

```

Insights The conditional at the beginning of each function in the clone group differs only in the conditional expression. However, our tool does not presently support expression-level differences, unless they are on constants or identifiers. Our tool manages this situation by pulling the merge point up to the smallest surrounding syntactic entity whose AST node type we support—the surrounding if statement—and applies a suitable abstraction pattern, as was described in Sect. 4.5.

Clone Group 4

The fourth clone group contains two statement-level differences and one statement inserted into the near-miss clone function `string_list_split_in_place`. Both differences are resolved using a switch statement.

```

1  int string_list_split(struct string_list *list, const char *string,
2                      int delim, int maxsplit)
3  {
4      int count = 0;
5      char *p = string, *end;
6
7      if (!list->strdup_strings)
8          die("internal error in string_list_split(): "
9              "list->strdup_strings must be set");
10     for (;;) {
11         count++;
12         if (maxsplit >= 0 && count > maxsplit) {
13             string_list_append(list, p);
14             return count;
15         }
16         end = strchr(p, delim);
17         if (end) {
18             string_list_append_nodup(list, xmemdupz(p, end - p));
19             p = end + 1;
20         } else {
21             string_list_append(list, p);
22             return count;
23         }
24     }
25 }

```

```

1  int string_list_split_in_place(struct string_list *list, char * string,
2  int delim, int maxsplit) {
3  int count = 0;
4  char * p = string, * end;
5  if (list->strdup_strings)
6  die("internal error in string_list_split_in_place(): "
7  "list->strdup_strings must not be set");
8  for (;;) {
9  count++;
10  if (maxsplit >= 0 && count > maxsplit) {
11  string_list_append(list, p);
12  return count;
13  }
14  end = strchr(p, delim);
15  if (end) {
16  *end = '\0';
17  string_list_append(list, p);
18  p = end + 1;
19  } else {
20  string_list_append(list, p);
21  return count;
22  }
23  }
24  }

```

```

int string_list_split_merge(struct string_list *list,
const char *string,
int delim, int maxsplit, int functionId) {
int count = 0;
char *p = string, *end;
switch (functionId) {
case 1:
if (!list->strdup_strings)
die("internal error in string_list_split(): "
"list->strdup_strings must be set");

break;
case 2:
if (list->strdup_strings)
die("internal error in string_list_split_in_place(): "
"list->strdup_strings must not be set");

break;
}
for (;;) {
count++;
if (maxsplit >= 0 && count > maxsplit) {
string_list_append(list, p);
return count;
}
end = strchr(p, delim);
if (end) {
switch (functionId) {
case 1:
string_list_append_nodup(list, xmemdupz(p, end - p));
break;
case 2:
*end = '\0';
string_list_append(list, p);
break;
};
p = end + 1;
} else {
string_list_append(list, p);
return count;
}
}
}
}

```

```
int string_list_split(struct string_list *list,
    const char *string,
    int delim, int maxsplit) {
    string_list_split_merge(list, string, delim, maxsplit, 1);
}
int string_list_split_in_place(struct string_list *list,
    const char *string,
    int delim, int maxsplit) {
    string_list_split_merge(list, string, delim, maxsplit, 2);
}
```

Insights Although the tool detects the second difference as one statement in the left hand side at line 18 of the function `string_list_split` and two statements at lines 15 and 16 of `string_list_split_in_place`, the post processing phase merges the two statement differences into a single one as they occur one after the other.

Clone Group 5

The fifth clone group contains two statement-level differences and one constant difference.

```
int git_inflate(git_zstream *strm, int flush)
{
    int status;

    for (;;) {
        zlib_pre_call(strm);
        /* Never say Z_FINISH unless we are feeding everything */
        status = inflate(&strm->z,
            (strm->z.avail_in != strm->avail_in)
            ? 0 : flush);
        if (status == Z_MEM_ERROR)
            die("inflate: out of memory");
        zlib_post_call(strm);

        /*
         * Let zlib work another round, while we can still
         * make progress.
         */
        if ((strm->avail_out && !strm->z.avail_out) &&
            (status == Z_OK || status == Z_BUF_ERROR))
            continue;
        break;
    }

    switch (status) {
        /* Z_BUF_ERROR: normal, needs more space in the output buffer */
        case Z_BUF_ERROR:
        case Z_OK:
        case Z_STREAM_END:
            return status;
        default:
            break;
    }
    error("inflate: %s (%s)", zerr_to_string(status),
        strm->z.msg ? strm->z.msg : "no message");
    return status;
}

int git_deflate(git_zstream *strm, int flush)
{
    int status;
```



```

for (;;) {
    zlib_pre_call(strm);

    /* Never say Z_FINISH unless we are feeding everything */
    status = deflate(&strm->z,
                    (strm->z.avail_in != strm->avail_in)
                    ? 0 : flush);
    if (status == Z_MEM_ERROR)
        die("deflate: out of memory");
    zlib_post_call(strm);

    /*
     * Let zlib work another round, while we can still
     * make progress.
     */
    if ((strm->avail_out && !strm->z.avail_out) &&
        (status == Z_OK || status == Z_BUF_ERROR))
        continue;
    break;
}

switch (status) {
    /* Z_BUF_ERROR: normal, needs more space in the output buffer */
    case Z_BUF_ERROR:
    case Z_OK:
    case Z_STREAM_END:
        return status;
    default:
        break;
}
error("deflate: %s (%s)", zerr_to_string(status),
      strm->z.msg ? strm->z.msg : "no message");
return status;
}

```

Our tool introduces two extra parameters, one, `functionId`, to switch between the statements based on which clone function calling the merged function, and another, `str1`, which our tool detects is of type `char*`.

```

int git_inflate_deflate(git_zstream * strm, int flush, char * str1,
int functionId) {
    int status;

    for (;;) {
        zlib_pre_call(strm);
        /* Never say Z_FINISH unless we are feeding everything */
        switch (functionId) {
            case 1:
                status = inflate( & strm->z,
                                (strm->z.avail_in != strm->avail_in) ? 0 : flush);

                break;
            case 2:
                status = deflate( & strm->z,
                                (strm->z.avail_in != strm->avail_in) ? 0 : flush);

                break;
        };

        if (status == Z_MEM_ERROR)
            die(str1);
        zlib_post_call(strm);

        /*
         * Let zlib work another round, while we can still
         * make progress.

```

```
        */
        if ((strm->avail_out && !strm->z.avail_out) &&
            (status == Z_OK || status == Z_BUF_ERROR))
            continue;
        break;
    }

    switch (status) {
        /* Z_BUF_ERROR: normal, needs more space in the output
           buffer */
        case Z_BUF_ERROR:
        case Z_OK:
        case Z_STREAM_END:
            return status;
        default:
            break;
    }
    switch (functionId) {
        case 1:
            error("inflate: %s (%s)", zerr_to_string(status),
                strm->z.msg ? strm->z.msg : "no message");

            break;
        case 2:
            error("deflate: %s (%s)", zerr_to_string(status),
                strm->z.msg ? strm->z.msg : "no message");

            break;
    };

    return status;
}

int git_inflate(git_zstream * strm, int flush) {
    return git_inflate_deflate(strm, flush, "inflate: out of
        memory", 1);
}

int git_deflate(git_zstream * strm, int flush) {
    return git_inflate_deflate(strm, flush, "deflate: out of
        memory", 2);
}
```

Insights As we have previously noted, the tool is unable to detect differences within strings. While we would ideally have passed only the strings “inflate” and “deflated”, our tool considered the whole strings as a difference. Our tool also considers strings that contain format directives such as %s to be differences that do not contain a resolution pattern and moves up one level to resolve the differences, as described in Sect. 4.5, so our tool had to fall back to performing a statement-level merge

Overall, we found that our tool can be integrated with a mainstream clone detector as a clone removal mechanism. We encountered three situations that required manual intervention:

1. merging return types int and void in C
2. merging a clone pair spread across two files
3. renaming variables

While manual intervention was necessary, our approach was effective in automating all other tasks involved in removing the clones detected in a non-trivial repository.

9 Related work

Since our work relates to many areas of refactoring and software clones, we have split our related work into sub-sections.

9.1 Clone detection

Our work is inspired by existing work on clone detection: Laguë et al. (1997) find that between 6.4 and 7.5% of the source code in different versions of a large, mature code base are clones. They only count clones that are exact copies (Type-1 clones, in the terminology of Koschke et al. (2006)), or copies modulo alpha-renaming (Type-2 clones). Baxter et al. (1998) report even higher numbers, sometimes exceeding 25%, on different code bases and with a different technique for clone detection that also counts near-miss clones (Type-3 clones). The prevalence of such near-miss clones is a strong indicator that copy–paste–modify is a widespread development methodology.

Other related work on clone detection detects clones and near-miss clones to identify faults (Juergens et al. 2009) and to enable refactoring (Choi et al. 2011). Similar to CCFinder/Gemini (Choi et al. 2011), our tool specifically looks for near-miss clones to merge; however, our focus is not on detecting near-miss clones in unknown code, but rather on merging detected clones. As our evaluation shows, our approach is effective on general clones.

9.2 Refactoring

The other closely related work is refactoring, which Martin Fowler (Fowler et al. 1999) defines as “the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure”. Our work can be considered as a complex form of refactoring, as we transform one version of the code with clones, into another version without the clones, without changing the program’s behavior. As in prior work, we break our transformations into individual, atomic components (Reichenbach et al. 2009; Schäfer et al. 2009), namely merges (which may be nested and require individual interaction) and fix-ups for existing code to use the re-factored code.

9.3 Clone management

Other work on clone management include tracking tools such as CloneBoard (de Wit et al. 2009) and Clone tracker (Duala-Ekoko and Robillard 2008). While CloneBoard provides the ability to organize clones and to some extent the ability to suggest the types of clones and possible resolution mechanisms, it lacks the ability to actually perform the resolution. Another approach to handling clones is linked editing (Toomim et al. 2004), which maintains the clones as they are, but allows editing of multiple clones simultaneously. This has the advantage of preserving code ‘as is’, but the disadvantage of requiring continued tool use for future evolution. Linked editing shares our view

that copy–paste–modify is an effective way to evolve software, but disagrees on how clones should be managed; it is an open question which approach is more effective for long-term software maintenance.

Krishnan and Tsantalis (Krishnan and Tsantalis 2014) have previously proposed an alternate approach to merging software clones. Their approach considers clones at all granularities, while our approach only targets method-level clones. Their strategies for the abstraction of conflicting expressions and statements are furthermore quite different than ours. For conflicting expressions, their approach is more aggressive than ours. Indeed, they abstract over various kinds of complex expressions, including function calls, allowing the clone merge to proceed only when a dependency analysis shows that moving the expression from its original position to the call site does not change the semantics. Our approach, on the other hand, only abstracts over various kinds of constants, for which the abstraction process is always correct. Finally, their approach to address Type-3 clones, in which whole statements may conflict within a clone, is to only allow the clone merge when these conflicting statements have no control or data dependencies on either the cloned code before the statement or the cloned code after the statement, and thus can be moved up or down out of the cloned region, respectively. In contrast, our approach leaves differing statements in place, to be selected by a flag value. Our approach in this case is much more flexible, freely allowing control and data dependencies between the conflicting statements and the cloned code. As the approach of Tsantalis et al. comes with many constraints, the major part of their evaluation assesses the refactorability of the clones identified by various clone detection tools on 9 Java projects. The rates range from 6.2% out of 741,610 clone pairs for NiCad to 33% out of 103,204 clone pairs for CloneDR. No evidence is provided that the resulting merged clones are acceptable to developers of the affected software projects.

Another closely related clone management approach is Cedar (Tairas and Gray 2012), which targets Java and relies on Eclipse refactorings for abstraction. Unlike our approach, Cedar is limited to Type-2 clones. As Roy et al. (2013) note, Type-3 clones are particularly common and frequently evolve out of Type-1 and 2 clones.

Another work that does clone refactoring was proposed by Zibran et al. (Zibran and Roy 2013). Although they propose clone refactoring like ours, their approach does not present an algorithm for merging generic near-clones but instead proposes approaches that are combinations of existing software refactorings. Their approach is aimed at generating an optimal schedule that will serve as a refactoring strategy for developers to remove clones and does not automatically remove clones by itself. A future direction could be to provide the steps inferred by our algorithm as an input to this scheduler and observe the results to prioritize what clones to merge.

Mandal et al. (2014) propose a tool that mines code-repositories for similarity preserving change patterns (SPCP), which are evolving code clones that are good candidates to be refactored. They perform a manual study to show that a significant portion of code available can be categorized as SPCPs as defined by their work. They then evaluate their MARC tool (Mining association rules among clones) by detecting SPCP clones in code-bases and making manual observations on the results. This system could provide input to our system, as we could detect the SPCPs using their approach and then merge them using ours.

In their work on unification and refactoring of clones, Krishnan and Tsantalis (2014) discuss a method of merging two ASTs. The approach basically matches the subtrees and detects all the subtrees that are exact matches of each other. Several preconditions are defined to determine whether the unmatched subtrees can be parametrized over the differences.

Another clone detection approach was proposed by Goto et al. (2013). Their approach simply detects candidates for the extract method refactoring. In the future, we could extend their approach to see if they can detect candidates for our resolution patterns too.

In their work on clone management for evolving software, Nguyen et al. (2012) identify types of changes to clones that may cause some inconsistencies. They do this by analyzing the code from SVN repositories and the updates that happen to these repositories. Unlike our approach, their approach works only on clone pairs.

9.4 Other related work

Our work ignores the C preprocessor (Medeiros et al. 2015) by operating only on preprocessed code. There is prior work on supporting the C preprocessor (Gazzillo and Grimm 2012). This work could be adapted to C++ to enable our system to support preprocessor-based abstraction patterns.

Our notion of ‘tree dissimilarity’ is only one possible metric for clone similarity. Smith and Horwitz (2009) propose more sophisticated approaches for similarity measurement that may be more suitable for clone-merging recommender systems than ours. By comparison, our choice penalises near-clones with substantial size differences.

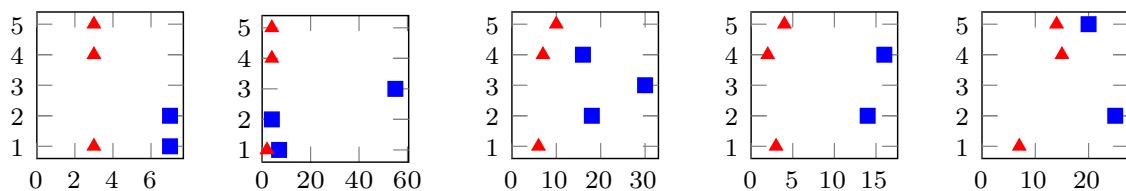
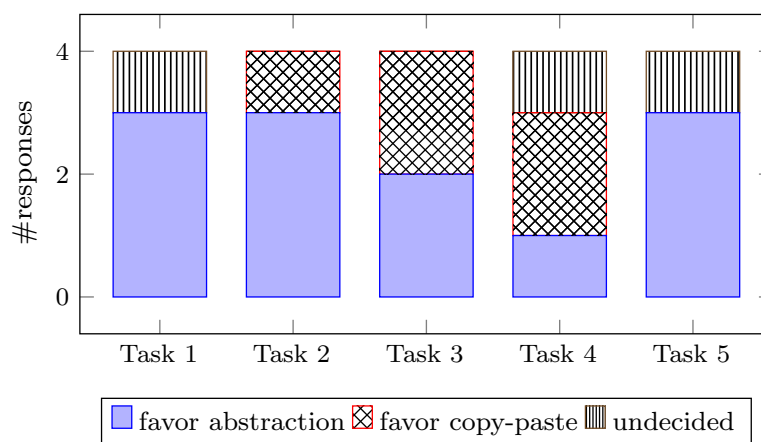
10 Conclusions

Managing code clones is a significant problem, given the amount of copied and pasted production-level code. This suggests that developers find reuse through code clones useful in practice, even when they know that reuse through manual abstraction would yield superior and more maintainable code; we find this confirmed both by prior work and by an informal poll that we conducted among C++ developers. We propose to close the gap between reuse through copy–paste based clones and abstraction through semi-automatic refactoring.

We have implemented a prototype of a suitable refactoring tool that identifies the parts of clones that can be merged, and proposes to the user suitable resolution patterns. The user then chooses one of the possible resolution patterns to decide how to merge the near-clones. We have evaluated this approach by implementing a prototype merging tool and applying a select set of resolution patterns to near-miss clones in popular GitHub repositories. We submitted the merged code back to the developers via pull requests and observed that the original developers found more than 50% (90% with

Table 6 Student experience levels (self reported)

| Student | #1 | #2 | #3 | #4 | #5 |
|------------|----------|----------|---------|---------|----------|
| Experience | 10 years | 3 months | 4 years | 1 years | 2 months |

**Fig. 5** Amount of time used for extending functionality. x -axis = time taken, y -axis = user, red triangle = copy-paste, blue triangle = abstraction (Color figure online)**Fig. 6** Preferred results after extending functionality. Out of the 20 answers we received, 3 were undecided, 5 preferred copy-pasted code, and 12 preferred abstracted code

the most recent version of our tool) of our changes to be desirable, merging them into their code bases.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

A Coding tasks and programmer poll

This appendix summarizes our informal poll. We asked five students (Table 6) to perform reuse tasks with copy-paste-modify and with manual abstraction; Fig. 5 summarizes the amount of time taken to complete the tasks. Each graph represents one task. For each task, the x -axis shows the time taken (in minutes) and the y -axis indicates the user. The red triangles represent the time taken for copy-paste tasks and the blue rectangle represent the time taken for abstraction tasks. Whenever one student performed both copy-paste-modify *and* manual abstraction, the student first completed the copy-paste-modify tasks. We later polled the students as to whether they

would prefer for the outcome to have been copy–paste–modified code or abstracted code. Four students responded; we summarize their responses for each task in Fig. 6.

References

- Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: *Proceedings of the International Conference on Software Maintenance, ICSM '98*, pp. 368–378. IEEE Computer Society, Washington (1998)
- Choi, E., Yoshida, N., Ishio, T., Inoue, K., Sano, T.: Extracting code clones for refactoring using combinations of clone metrics. In: *Proceedings of the 5th International Workshop on Software Clones, IWSC '11*, pp. 7–13. ACM, New York (2011)
- Cordy, J.R., Roy, C.K.: The NiCad clone detector. In: *2011 IEEE 19th International Conference on Program Comprehension (ICPC)*, pp. 219–220 (2011)
- de Wit, M., Zaidman, A., van Deursen, A.: Managing code clones using dynamic change tracking and resolution. In: *IEEE International Conference on Software Maintenance, 2009, ICSM 2009*, pp. 169–178 (2009)
- Duala-Ekoko, E., Robillard, M.P.: CloneTracker: tool support for code clone management. In: *ICSE*, pp. 843–846 (2008)
- Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading (1999)
- Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. In: *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy*, pp. 147–158, 12–16 July 2010
- Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Boston (1995)
- Gazzillo, P., Grimm, R.: SuperC: parsing all of C by taming the preprocessor. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, Beijing, China*, pp. 323–334 (2012)
- Goto, A., Yoshida, N., Ioka, M., Choi, E., Inoue, K.: How to extract differences from similar programs? A cohesion metric approach. In: *2013 7th International Workshop on Software Clones (IWSC)*, pp. 23–29 (2013)
- Hunt, A., Thomas, D.: *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman, Boston (1999)
- Juergens, E., Deissenboeck, F., Hummel, B., Wagner, S.: Do code clones matter? In: *IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009*, pp. 485–495 (2009)
- Kapser, C.J., Godfrey, M.W.: cloning considered harmful considered harmful: patterns of cloning in software. *Empir. Softw. Eng.* **13**(6), 645–692 (2008)
- Koschke, R., Falke, R., Frenzel, P.: Clone detection using abstract syntax suffix trees. In: *Proceedings of the 13th Working Conference on Reverse Engineering, WCRE '06*, pp. 253–262. IEEE Computer Society, Washington (2006)
- Krishnan, G.P., Tsantalis, N.: Unification and refactoring of clones. In: *2014 Software Evolution Week—IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pp. 104–113 (2014)
- Laguë, B., Proulx, D., Merlo, E.M., Mayrand, J., Hudepohl, J.: Assessing the benefits of incorporating function clone detection in a development process. In: *Proceedings of International Conference on Software Maintenance (ICSM)*, pp. 314–321. IEEE Computer Society Press (1997)
- Mandal, M., Roy, C.K., Schneider, K.A.: Automatic ranking of clones for refactoring through mining association rules. In: *2014 Software Evolution Week—IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pp. 114–123 (2014)
- Medeiros, F., Kästner, C., Ribeiro, M., Nadi, S., Gheyi, R.: The love/hate relationship with the C pre-processor: an interview study. In: Boyland JT (ed.) *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 37. pp. 495–518 (2015)
- Narasimhan, K., Reichenbach, C.: Copy and paste redeemed (t). In: *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pp. 630–640 (2015)

- Negara, S., Vakilian, M., Chen, N., Johnson, R.E., Dig, D.: Is it dangerous to use version control histories to study source code evolution? In: Proceedings of the 26th European Conference on Object-Oriented Programming, ECOOP'12, pp. 79–103. Springer, Berlin (2012)
- Nguyen, H.A., Nguyen, T.T., Pham, N.H., Al-Kofahi, J., Nguyen, T.N.: Clone management for evolving software. *IEEE Trans. Softw. Eng.* **38**(5), 1008–1026 (2012)
- Pawlik, M., Augsten, N.: RTED: A robust algorithm for the tree edit distance. In: Proceedings of the VLDB Endowment, vol. 5, no. 4 (2011)
- Raychev, V., Vechev, M., Krause, A.: Predicting program properties from “big code”. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 111–124 (2015)
- Reichenbach, C., Coughlin, D., Diwan, A.: Program metamorphosis. In: European Conference on Object-Oriented Programming (ECOOP). Springer, Berlin, pp. 394–418 (2009)
- Roy, C., Schneider, K., Perry, D.: Understanding the evolution of type-3 clones: an exploratory study. In: MSR (2013)
- Saha, R.K., Roy, C.K., Schneider, K.A., Perry, D.E.: Understanding the evolution of type-3 clones: an exploratory study. In: 2013 10th Working Conference on Mining Software Repositories (MSR), pp. 139–148 (2013)
- Schäfer, M., Verbaere, M., Ekman, T., de Moor, O.: Stepping stones over the refactoring rubicon. In: ECOOP, pp. 369–393 (2009)
- Smith, R., Horwitz, S.: Detecting and measuring similarity in code clones. In: Proceedings of the International workshop on Software Clones (IWSC) (2009)
- Tairas, R., Gray, J.: Increasing clone maintenance support by unifying clone detection and refactoring activities. *Inf. Softw. Technol.* **54**(12), 1297–1307 (2012)
- Toomim, M., Begel, A., Graham, S.L.: Managing duplicated code with linked editing. In: VLHCC (2004)
- Zibran, M.F., Roy, C.K.: Conflict-aware optimal scheduling of prioritised code clone refactoring. *IET Softw.* **7**(3), 167–186 (2013)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.