Copy and Paste Redeemed

Krishna Narasimhan
Institut für Informatik
Goethe University Frankfurt
krishna.nm86@gmail.com

Christoph Reichenbach
Institut für Informatik
Goethe University Frankfurt
reichenbach@cs.uni-frankfurt.de

***Abstract*—Modern software development relies on code re-use, which software engineers typically realise through hand-written abstractions, such as functions, methods, or classes. However, such abstractions can be challenging to develop and maintain. One alternative form of re-use is *copy-paste-modify*, a methodology in which developers explicitly duplicate source code to adapt the duplicate for a new purpose. We observe that copy-paste-modify can be substantially faster to use than manual abstraction, and past research strongly suggests that it is a popular technique among software developers.**

We therefore propose that software engineers should forego hand-written abstractions in favour of copying and pasting. However, empirical evidence also shows that copy-paste-modify complicates software maintenance and increases the frequency of bugs. To address this concern, we propose a software tool that merges together similar pieces of code and automatically creates suitable abstractions. This allows software developers to get the best of both worlds: custom abstraction together with easy re-use.

To demonstrate the feasibility of our approach, we have implemented and evaluated a prototype merging tool for C++ on a number of near-miss clones (clones with some modifications) in popular Open Source packages. We found that maintainers find our algorithmically created abstractions to be largely preferable to existing duplicated code.

I. Introduction

As software developers add new features to their programs, they often find that they must modify existing code. The developers now face a choice: they can either introduce a (possibly complex) abstraction into the existing, working code, or copy, paste, and modify that code. Introducing the *abstraction* produces a smaller, more maintainable piece of 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*, which tend to decrease maintainability, but avoid the risk of damaging existing functionality [10].

Duplication is wide-spread [2], [1], especially if we count near miss clones (Type-3 clones in the terminology of Koschke et al. [11]), i.e., clones with nontrivial modifications. However, duplication is unpopular in practitioner literature [8] and "*can be a substantial problem during development and maintenance*" [9] as "*inconsistent clones constitute a source of faults*". Similarly, we found evidence in an informal poll among C++ developers that practitioners prefer abstraction over duplication. This suggests that there is a *re-use discrepancy* between what developers want and what they do.

Kapser and Godfrey [10] offer one possible explanation: they claim that "*code cloning can be used as a 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 faster or easier to use. In our informal poll among C++ developers, we found evidence that supports this idea (Section II-A).

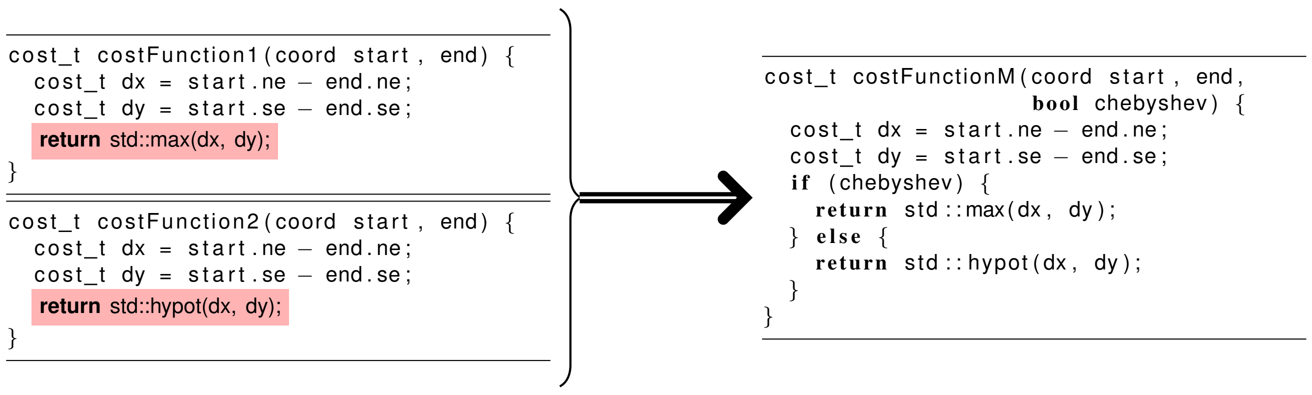
In this paper we propose a novel solution to the re-use discrepancy that offers all the speed and simplicity of copy-paste-modify together with the design benefits offered by abstraction, by using a refactoring algorithm to merge similar code semi-automatically. With our approach, developers re-use code by copying, pasting, and modifying it, producing *near-clones* (equivalent to *near miss clones*[1]). Developers then invoke our tool to merge two or more near-clones back into a single abstraction. Since there may be multiple ways to introduce an abstraction (e.g., function parameters or template parameters), our tool may ask the developer to choose which abstractions they prefer during the merge. Our tool is easily extensible, so that developers may add support for their own abstractions (e.g., project-specific design patterns).

We find that our approach is not only effective at solving the aforementioned re-use discrepancy, but also produces code that meets the qualitative 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 [14], [16], as opposed to the (unbounded number of) hand-written *instances* of manual abstractions that we see without our tool.

Our contributions are as follows:

- We describe an algorithm that can automatically or semi-automatically merge near-clones and introduces user-selected abstractions.
- We describe common abstraction patterns for C++, supported by our implementation.
- We report on initial experiences with our algorithm on popular C++ projects drawn from Open Source repositories. We find that code merged by our approach is of sufficiently high quality to be accepted as replacement to unmerged code in the majority of cases.

[1]We prefer the term *near-clones* over *near miss clones*, as the notion of a *miss* is only natural in clone *detection* but not in clone *generation*.



```
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.

- We describe the results of an informal poll among C++ programmers that involved a number of coding tasks. While the poll has only a small sample size, its findings show that copy-paste-modify can substantially outperform manual abstraction in practice.

Section II further motivates our approach and briefly sketches our algorithm. Section III describes our merge algorithm. Section IV then discusses our implementation. Section V presents our evaluation on Open Source software. Section VI discusses related work, and Section VII concludes.

II. Towards Principled Software Development with Copy-Paste-Modify

Past work in clone detection has found that clones are widespread [2], [11], [1]. We hypothesise 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.

A. Benefits of Copy-Paste-Modify

For our exploratory experiment, we selected five pairs of C++ methods from the Google Protobuf [2], Facebook HHVM [3], and Facebook RocksDB [4] Open Source repositories, randomly choosing from a set of near-clones reported by a simple clone detector (Section III-A). We then removed one of the methods and asked five graduate students with 2 months, 3 months, and 1, 4, and 10 years of 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 three students to use *copy-paste-modify*.

We found that the students using copy-paste-modify were almost universally faster in completing their objectives (2–15 minutes) than the students who performed manual abstraction (7–55 minutes, with three tasks left incomplete). We found only one exception, where the best-performing student on 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, we asked them to also perform manual abstraction on a total of five of the problems they had just solved — but despite their familiarity with the code, they consistently performed worse (taking more than twice as long as before) when completing the exact same task again with manual abstraction. However, 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 VIII).

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

B. Copy-Paste-Modify versus Manual Abstraction

To understand *why* copy-paste-modify might be easier, consider function costFunction1 from **Figure 1**. This function (adapted, like the rest of the example, from the OpenAge[5] project), computes the Chebyshev distance of two two-dimensional coordinates. The implementation 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 they need a different distance function, describing the beeline distance between two points (i.e., $\sqrt{\mathsf{dx}^2 + \mathsf{dy}^2}$). Computing this distance requires almost exactly 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: they 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 they can manually alter function costFunction1 into costFunctionM (depicted on the right in **Figure 1**) or a similar function.

This migration requires introducing a new parameter, introducing an **if** statement, adding a new line to handle the new case, and updating all call sites with the new parameter (perhaps using a suitable automated refactoring). Intellectually,

[2]https://github.com/google/protobuf
[3]https://github.com/facebook/hhvm
[4]https://github.com/facebook/rocksdb

[5]http://openage.sft.mx/

the programmer 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 needed to concern themselves with the exact differences between what already existed and what they now needed.

We observe that it is possible to devise an algorithm that takes costFunction1 and costFunction2 and abstracts them into a common costFunctionM, taking care that any callers still continue to work correctly. Note that there are other possible solutions for costFunctionM. The one illustrated here is straightforward, but different code and different requirements may call for different solutions. 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; we thus opt to rely on user interaction to choose the most appropriate abstraction mechanism for a given situation.

III. Merging Algorithm

To merge two or more such functions (or, analogously, methods), we examine and transform their abstract syntax trees (ASTs). We refer to the sets of ASTs for multiple functions as *clone groups*. Our merge algorithm first computes the pairwise differences between each of the ASTs in one clone group, then collects differences that are shared among multiple clones, and finally merges the differences by resolving them through one of several *resolution patterns*.

A. Linking Near-Clones through Robust Tree Edit Distance

We detect differences between ASTs through the Robust Tree Edit Distance (RTED) algorithm [13]. 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 *rename* operations:

- **delete** a node and connect its children to its parent, maintaining the order.
- **insert** a node between two adjacent sibling nodes.
- **rename** the label of a node, essentially replacing one node by another.

B. Identifying Common Differences

We consider any node that occurs in an edit list to not be shared between the two ASTs. To illustrate how we use this information, we will use the code from **Figure 4** as a running example. This somewhat synthetic example presents three unlikely merge candidates — the functions are both sufficiently small and sufficiently dissimilar that users might not be inclined to merge them. However, the example showcases special cases in our algorithm and illustrates that our approach works even for code with a large degree of variation.

Our algorithm now considers the ASTs of the three functions (upper half of **Figure 5**) and determines the following:

1) The set of nodes that are common to all the ASTs (i.e., which do not occur in any edit lists), which the algorithm will place in the merged tree unaltered. In the example in **Figure 5**, this set contains the nodes $\{\mathrm{a}, \mathrm{x}, \mathrm{y}, \mathrm{z}\}$.
2) All *other* nodes, together with:
    a) The alternative choices for such nodes that we must merge. These alternatives must have matching node types (e.g., statement, expression, ...), or our algorithm will abort.
    b) The ASTs that each of the alternatives belongs to.

Consider the nodes 'b' from ASTs 1 and 2 and the node 'b2' from AST 3 in **Figure 5**. These nodes are competing for the same place in the target AST, but they are not identical; we refer to such nodes as *merge candidates*. Node 'b' is an example of a node that appears in multiple ASTs without occurring in all of them. We first identify such partially shared nodes from our edit lists: these nodes show up in some edit lists, but not in all of them. We use the notion of a *Unique Set* (U in the Figure 2) to describe the nodes that are uniquely shared among a particular subset of ASTs. Consider the example in Figure 5. The nodes 'b' and 'c' are present only in AST_1 and AST_2. We write $U(\mathsf{AST}_1, \mathsf{AST}_2)$ to describe nodes that are present precisely in the two specified ASTs but in none of the other ASTs that we are considering. Since there are no further nodes that are unique to AST_1 and AST_2, $U(\mathsf{AST}_1, \mathsf{AST}_2) = \{\mathrm{b}, \mathrm{c}\}$.

Whenever we have merge candidates competing for an AST node, we introduce a special operator into the output AST to facilitate the merge. However, we only need those operators in places where there is actual disagreement between what node should be there. These places are precisely the roots of the subtrees that are not shared between all ASTs. For this purpose, we further filter the Unique Sets into Rootsets. The Rootset of a set of nodes N is the minimal set *Rootset*$(N) \subseteq N$ such that all nodes $n \in N$ have an ancestor in *Rootset*(N). (We consider n to be an ancestor of itself for our purposes.) The minimal Rootset is unique because otherwise there would be at least one node with two parents, which is not possible in an AST. The Rootset consists of all the nodes in the set that have no parent nodes that are also in the set. **Figure 3** illustrates an example computation of these sets for the ASTs in **Figure 5**.

Figure 2 summarises at a high level how our algorithm identifies the various sets in **Figure 3**. We observe precisely one Unique Set that is different from its Rootset. This is because the node 'b' the a parent of the node 'c' and so the Rootset of that subset $\{\mathsf{AST}_1, \mathsf{AST}_2\}$ can be reduced to just 'b'. We assume that there are mechanisms to retrieve the information about the position of every node, apart from its label. With that information, we gather that node 'b' from the set $\{\mathsf{AST}_1, \mathsf{AST}_2\}$ and node 'b2' from the set $\{\mathsf{AST}_3\}$ are merge candidates.

C. Merge Algorithm

Given the Unique Sets and Rootsets, we first construct an intersection tree, which is simply the intersection of all ASTs (i.e., the common nodes and tree edges). Consider the common nodes generated from **Figure 3**, $\{\mathrm{a}, \mathrm{x}, \mathrm{y}, \mathrm{z}\}$. The algorithm adds these nodes as-is, as illustrated in **Figure 5**.Our algorithm

Overview of the steps in the 'Identifying Common Differences' phase:

1) We begin with a set of all ASTs in our clone group, AST_{all}.
2) For each set of ASTS $s \subseteq AST_{all}$, we compute the unique set $U(s)$. Unique Sets describe the nodes that are present in all ASTs in s, but not in any ASTs in $AST_{all} \setminus s$.
3) We compute the Rootset of each computed Unique Set. Rootset(ndSet) of a set of nodes is the minimal set of nodes belonging to ndSet such that every node in ndSet has an ancestor in Rootset(ndSet).

Fig. 2. Overview of the preliminary phase.

$$AST_{all} = \{\mathsf{AST}_1, \mathsf{AST}_2, \mathsf{AST}_3\}$$

$$\begin{array}{lclrcl} U(\mathsf{AST}_1) & = & \{d, f1\} & \textit{Rootset} & = & \{d, f1\} \\ U(\mathsf{AST}_2) & = & \{e, f2\} & \textit{Rootset} & = & \{e, f2\} \\ U(\mathsf{AST}_3) & = & \{b2, f3, n\} & \textit{Rootset} & = & \{b2, f3, n\} \\ U(\mathsf{AST}_1, \mathsf{AST}_2) & = & \{b, c\} & \textit{Rootset} & = & \{b\} \end{array}$$

Fig. 3. Example sets generated by the common difference identification phase

assumes that every node in the Rootsets presents an opportunity to merge. For each such node, the algorithm gathers the nodes that need to be merged at the same child position with the same parent node, and introduces a *merge point*. In our example, node 'b' from the Rootset for $\{\mathsf{AST}_1\mathsf{AST}_2\}$ and node 'b2' from the Rootset for $\{\mathsf{AST}_3\}$ would be candidates for merging, as both are competing for the position of the first child of node 'a'. We introduce a merge point ('MP(12, 3)') at that position, and attach 'b' and 'b2' as children.

Another example is the conditional introduction of node 'n'. This introduction is unique to AST_3, but it moves node 'y' to child position 3 of node 'a'. Our algorithm automatically compensates to ensure that 'f1', 'f2', and 'f3' are now merged at child position 4, even though they appear as the third child of 'a' in AST_1 and AST_2.

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

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

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

```
void fnMerged(int functionId, int fValue, int bParam)
{
  if(functionId == 1 || functionId == 2)
  {
    b(c, bParam);
  }
  if(functionId == 3)
  {
    b2();
    n();
  }
  y = fValue;
  x(z);
}
```

Fig. 4. Example of a three-way merge supported by our tool.

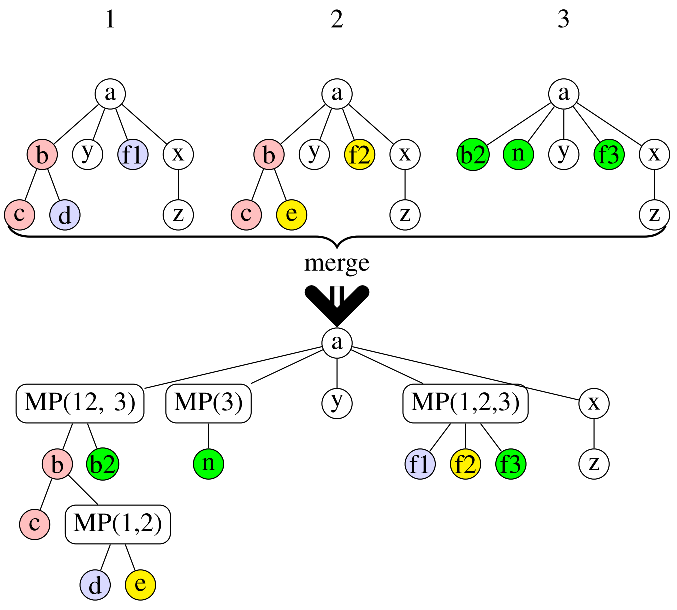


Fig. 5. Example merge

Finally, we eliminate merge points by applying *resolution patterns*. A resolution pattern is a code transformation pattern to resolve the merging of certain types of nodes at given merge points. A resolution pattern will return a node to be inserted at the merge point under consideration. Additionally, it can transform other parts of the AST. The merging algorithm is therefore not a fully automatic process. It identifies merge points and for each merge point, the user picks a resolution pattern that effects the merge at that point. The algorithm identifies the merge points separately from the resolution patterns because some resolution patterns may apply to multiple merge points. **Table I** lists the resolution patterns that our prototype system supports, together with the node types they are applicable to.

Consider the example of replacing the same integer constant in multiple points. The algorithm offers resolution patterns based on the node type for each merge point. For example, if the nodes under consideration in a particular position are all constants, we can introduce introduce an extra parameter of the type of the constant and pass the constant as the parameter value. Another possibility would be to introduce a global field that could be assigned the constant. We split resolution patterns into a 'merge-substitution' part and a 'fix-up' part.

The merge-substitution part contains the merged node that is to be inserted into the merged method. This merge-substitution part replaces the merge point in the AST. The fix-up part handles other modifications that need to be performed. These modifications could involve handling call sites or introducing parameters to the merged method or even changing other classes and introducing super-classes. While we could only use a single transformation for each resolution pattern,

TABLE I. AVAILABLE RESOLUTION PATTERNS AS OPTIONS PRESENTED TO THE USER

Type of Node	**Possible Resolution Patterns (Caller w/ Callee)**
Statement	Switch, Conditional Branch with Extra Parameter, Field
Literal	Extra Parameter, Field
Type of an argument	Template Parameter
Identifier	Template Parameter, Extra Parameter

we find that discussing the merge-substitution part separately is helpful in understanding each pattern.

Consider again our example in **Figure 4**. When we merge the ASTs generated for function1 through function3, we obtain a merged AST as in the lower half of **Figure 5**. If we then pick suitable resolution patterns, we obtain the function fnMerged from **Figure 4**.

In the following subsections, we discuss resolution patterns that we implemented to evaluate our approach. Although our approach is generic and theoretically can be applied to any AST chunks, we merged method definitions and replaced the existing definitions with calls to the merged method. Since the merging process involves creating a new merged method and introducing sensible calls to the merged method, we generate a merged version of the parameters. A merged version of the parameters is simply a combined list of the parameters of the individual methods. Two parameters are equal if their types, names and type qualifiers specifiers are equal. We maintain a map of the individual parameters to their positions in the merged parameters to generate appropriate calls. For each of the following resolution patterns, we describe the merge resolution and the fix-up part.

We illustrate our resolution patterns with examples taken from Open Source projects hosted at Github. We picked these four patterns by studying clone groups containing various near-clone methods and selecting the four patterns that we found to cover the largest subset of the cases that we considered. We found them to be fully sufficient for the examples that we had randomly selected for evaluation. The examples presented here are abbreviated for space reasons. In the examples, the nodes highlighted in red indicate the unique nodes in each function and the nodes highlighted in blue indicate the nodes emerging from the merge resolution.

1) Pattern: Switch Statement with Extra Parameter: This resolution can be applied if the nodes to be merged are all statements. We then construct the following **switch** statement: *Merge-Resolution*:

```
switch choice {
  case 1: stmt_1; break;
    ...
  case k: stmt_k; break;
}
```

where *choice* is a fresh function/method parameter, $stmt_i$ is one statement alternative taken from a Unique Set, and i is a unique number identifying the Unique Set.

Fix-up: We add *choice* as an additional formal parameter to the surrounding method or function and modify the corresponding call sites to supply their own AST Ids as actual parameters. Consider the function snippets

```
jobject
    function_openROnly__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,..) {
    return
  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, this produces the following output (with the generated switch statement in lines 23–31):

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

```
  return null;
}
```

2) Pattern: Pass Extra Parameter for Literal Expressions: This resolution can be applied if the nodes to be merged are all literal expressions. Literal expressions are nodes that have a constant value and type. We require that each of these constants are of the same type. The merge-resolution is a simple identifier expression that switches between the corresponding constants based on the values passed to *value*, a fresh parameter. We resolve this pattern with an identifier expression, which is a node that contains a name of a field or a variable.
Merge-Resolution: *value*
Fix-up: We add *value* as additional formal parameter to the surrounding method or function and modify existing call sites to supply their own constants as actual parameters input. Consider the following function snippets, taken from the Oracle's Node-OracleDB project[6]:

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

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

Our tool would identify that the calls to getClientId, getModule and getAction are resolvable using an extra parameter. Modulo variable renaming and indentation, this produces the following output, the Id Expression *errorMsg* produced in line 13:

```
1  Handle<Value>
2  Connection::GetProperty
3  (Local<String> property,
4  const AccessorInfo& info,
5  string errorMsg)
6  {
7   ...
8    if(!njsConn->isValid_)
9     ...
10   else
11     msg =
12     NJSMessages::getErrorMsg
13      (errWriteOnly, errorMsg);
14  NJS_SET_EXCEPTION(msg.c_str(),
15    (int) msg.length());
16   return Undefined();
17 }
18
19 Handle<Value>
20 Connection::GetClientId
21 (Local<String> property,
22 const AccessorInfo& info)
23 {
24   return
25   Connection::GetProperty
26   (property, info, "clientId");
27 }
28
29 /* The methods getModule and getAction
30 are constructed to be similar to getClient*/
```

3) Pattern: Templates for Type Expressions: We can apply this resolution if the nodes to be merged all represent types. We again introduce a fresh identifier expression, *type*.
Merge-Resolution: *type*
Fix-up: The fix-up introduces a new formal template type parameter to the function definition for type parameter *type*. Consider the function snippets taken from the RethinkDB project[7]

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

cJSON *cJSON_CreateDoubleArray
( double *numbers,int count) {
```

[6]https://github.com/oracle/node-oracledb/

[7]https://github.com/rethinkdb/rethinkdb/

```
    ...
    for (int i=0;a && i<count;i++) {
        ...
    }
    a->tail = p;
    return a;
}
```

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

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

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

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

4) Pattern: Pass Extra Parameter for Identifiers: This resolution can be applied if the nodes to be merged are all variable identifiers (identifier expression nodes), i.e., hold the name of a field or a variable. We require that each of these variables is of the same type.The merge-resolution is a simple id expression that switches between the corresponding variable names based on the values passed to the '*value*', a fresh parameter.
Merge-Resolution: *value*
The resolution here is very similar to the pattern 2, except that our algorithm promotes lvalues to pointer-type parameters whenever needed. Passing identifiers is more challenging than passing literals. An interesting scenario in this pattern is when the variable is assigned before reference. Consider the following example:

```
void fn1()
{
    int x = 10;
    int y = x + 1;
}
void fn1()
{
    int z = 10;
    int y = z + 1;
}
```

Our algorithm handles this case by identifying two different merge points each for the identifiers x and z and performing a post processing step to link the definition and reference of the variable.
Fix-up: We add *value* as additional formal parameter of the type of the identifiers being merged and modify the corresponding call sites to supply their own identifiers as actual parameters. Consider the function snippets taken from Facebook's HHVM project[8]:

```

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 calls 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); }
```

IV. Implementation

We implemented the distance calculator, the algorithms and the framework on top of Eclipse CDT[9]. We adapted an existing implementation of RTED[10] and modified it to fit our CDT AST representation. The existing implementation worked on in-order representations of trees with String nodes. We replaced the nodes to contain information about AST node types and content. We supplied the source file with the clone groups as input to an Eclipse environment setup to include the merging as a Refactoring menu option. We marked the potential candidates using **pragma** annotations. The result was a new file with the marked functions merged and the original functions calling the new merged function with the common functionality. We copied the result file back to the repository, overwriting the original version of the file and the repository was built, tested and run. We also identified potential candidates for merging

[8]https://github.com/facebook/hhvm/
[9]https://eclipse.org/cdt/
[10]http://www.inf.unibz.it/dis/projects/tree-edit-distance/download.php

using our modified edit distance calculator. We explain the details of the actual identification and the merges in the evaluation sectionV.

We have made prototype publicly available[11].

V. Evaluation

We 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 popular Github repositories, identified potential candidates for merging, and abstracted the identified candidates using our approach. 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 in their production code. We then performed a total of 18 abstractions of clone groups and sent them as pull requests to trending Github repositories. **Table II** lists the repositories that we looked at for our evaluation, the URLs of the pull requests, the number of clone groups abstracted per repository, and the status of the pull requests.

A. Identifying and Merging Clone Groups

The most promising clone group candidates for our approach are those with near-clones. Unfortunately, these are are particularly difficult to find for traditional clone detectors. We therefore used *edit distance* as a metric for identifying potential clone groups. Edit distance is a metric that we can compute with the RTED algorithm (Section III-A) by simply measuring the length of an edit list.

We started with the repositories in **Table II** and collected all function/method pairs belonging to the same source file. We then calculated the edit distance of each pair. For each function pair, we first determined the bigger function (by AST node count). We then marked the function pair as a potential clone group code if the number of nodes in the bigger function (*#fnBigger*)) was greater than a customisable $threshold_n$ and if the ratio of the edit distance to *#fnBigger* was less than a customisable $threshold_r$.

We then randomly picked function pairs (potential clone groups) out of the candidates. For each function pair we manually explored other related functions that we also found listed as near-clones. Whenever our developer intuition suggested that these additional functions were similar enough in behaviour and structure, we added them to the clone group, as a developer would do when using our tool in practice. Each clone group contained 2–4 functions. Note that this process was purely for evaluation purposes, as the focus of our work is on merging, and not on clone detection. We then merged the clone groups, used a predetermined resolution pattern for each node type before submitting the pull requests:

- We resolved differences in statements using switch statements and an extra parameter specifying the AST statement to branch to (Pattern 1)
- We resolved differences in literal expressions (constants) by passing additional parameters (Pattern 2)
- We resolved differences in type expressions using templates (Pattern 3)
- We resolved differences in identifier expressions using additional parameters (promoted to pointers for LValues), and formal parameters specifying the name or the address of the variable (Pattern 4)

We also performed minor manual changes. These include:

- Providing meaningful names to parameters. Our tool generated random fresh names based on the position of the merge points.
- Adding function prototypes to header files.

The manual changes are standard refactorings that are not central to our approach.

B. Results

We performed our initial evaluation (Phase 1) using an early version of our merging tool that could perform only merges of pairs of functions and did not support multiple resolution patterns for the same pair. During Phase 1, we ran our distance calculator on the top trending C++ repositories in Github during the month of December 2014, and selected potential clone groups after setting $threshold_r$ to 0.5 and $threshold_n$ to infinity, meaning that we accepted functions of all sizes. We submitted 8 abstractions as pull requests and all but one of the clone group abstractions were rejected or pending. The results of the pull requests highlighted areas of improvement in our first prototype.

TABLE III. Phase 1 results

Submitted	Accepted	Rejected	Pending
8	1	3	4

We performed our second evaluation (Phase 2) using a complete version of the our merging tool, capable of merging an arbitrary number of functions/methods at the same time. This version also supported resolving multiple merge points with different node types. During Phase 2, we ran our distance calculator on the top trending repositories during the month of February 2015. We set $threshold_r$ to 0.15 and $threshold_n$ to 100. We changed the thresholds building on experience from Phase 1 in order to focus on clone groups that would save more lines of code when abstracted. The thresholds were thus set to very strict numbers, meaning the clones were very similar to each other and tied to methods of substantial sizes. We then submitted 10 abstractions as pull requests, summarised in the table below, and found that all of them were accepted, except for one:

TABLE IV. Phase 2 results

Submitted	Accepted	Rejected	Pending
10	9	1	0

[11] http://sepl.cs.uni-frankfurt.de/copy-paste.html

TABLE II. Repositories with their pull request URLs. Each clone group represents one abstraction. We encourage readers who choose to look at the pull requests to go through the comments. While some of the pull requests don't explicitly have their status listed as 'merged', as with the OracleDB and the MongoDB repositories, the codes have actually been merged, as indicated by maintainer comments.

Repository	**Phase**	**Clone Groups**	**Status**	**URL**
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 • 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
cocos2d/cocos2d-x	2	2	Accepted	• https://github.com/cocos2d/cocos2d-x/pull/10539 • https://github.com/cocos2d/cocos2d-x/pull/10546
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	1 Accepted 1 Rejected	• https://github.com/google/protobuf/pull/128/ • https://github.com/google/protobuf/pull/126
SFTtech/openage	1	1	Rejected	• https://github.com/SFTtech/openage/pull/176

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.

C. *Analysis of Rejected and Pending Results*

We present the results of the pending and rejected pull requests summarised in **Table II** and provide our analysis on the same.

1) Pending results: Below, we note feedback to pull requests that were neither accepted nor rejected. Let us first discuss the pending pull request from RocksDB. The exact 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 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 be their top priority.

The other pending pull request was from the OpenExr repository. The request merged 3 clone groups at once, and received a mixture of responses. One maintainer requested an explanation of the advantages. Another maintainer expressed scepticism 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. All these activities took place over a 3 month period.

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 remaining rejected clone group abstraction for the ideawu/ssdb repository.

VI. Related Work

Our work is inspired by existing work on clone detection: Laguë et al. [2] 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. [11]), or copies modulo alpha-renaming (Type-2 clones). Baxter et al. [1] 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 (or Type-3 clones), which are substantially related pieces of software in which small parts of the AST subtree may differ. We consider the prevalence of such near miss clones to be strong indicators that copy-paste-modify is a wide-spread development methodology.

Other related work on clone detection focuses on detecting clones and near-clones to identify faults [9] and enable refactoring [3]. Similar to CCFinder/Gemini [3], our tool specifically looks for near-clones to merge; however, our focus is not on detecting near-clones in unknown code, but rather on merging *deliberate* and *known* clones. As our evaluation shows, our approach is effective on general clones.

The other closely related work is refactoring [6]. As in prior work, we break our transformations into individual, atomic components [14], [16], namely merges (which may be nested and require individual interaction) and fixups for existing code to use the refactored code.

Other work on clone management include tracking tools like CloneBoard [4] and Clone tracker [5]. While CloneBoard provides the ability to organise clones and to some extent suggest the types of clones and possible resolution mechanisms, clone board lacks the ability to actually perform an abstraction and merge clones into a common functionality. Another approach to handling clones is linked editing [18]. Linked editing, unlike our approach, 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 usage 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 of the two approaches is more effective for long-term software maintenance.

Perhaps the most closely related clone management approach to our algorithm is Cedar [17], which targets Java and relies on Eclipse refactorings for abstraction. Unlike our approach, Cedar is limited to Type-2 clones. To the best of our knowledge, ours is the only work to support merging the common Type-3 clones (inexact clones) in a wide variety of cases. As Roy et al. [15] note, Type-3 clones are particularly common and frequently evolve out of Type-1 and 2 clones.

While our work ignores the C preprocessor [12] in C++, there is prior work on supporting the C preprocessor in C [7]. This work could be adapted to C++ to enable our system to support preprocessor-based abstraction patterns.

VII. Conclusions

Managing code clones is a significant problem, given the amount of copied and pasted production level code. This suggests that developers find re-use through code clones useful in practice, even when they know that re-use 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 re-use through copy-paste based clones and abstraction through semi-automatic refactoring. We have implemented a prototype of a suitable refactoring tool that identifies parts of clones that can be merged, and proposes suitable resolution patterns to merge them to the user. The user then makes the design decision of how to merge. We have evaluated this approach by implementing a prototype merging tool and applying a select set of resolution patterns to near-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 the most recent version of our tool) of our changes to be desirable, merging them back into their code bases.

VIII. Appendix: Programmer Poll

This appendix summarises our informal poll. We asked five students (**Table V**) to perform re-use tasks with copy-paste-modify and with manual abstraction; **Table VI** summarises the amount of time taken to complete the tasks. Whenever one student performed both copy-paste-modify *and* manual abstraction, the student first completed the copy-paste-modify tasks. We later polled students whether they would prefer for the outcome to have been copy-paste-modified code or abstracted code. Four students responded; we summarise their responses for each task in **Figure 6**.

TABLE V. Student experience levels.

Student	#1	#2	#3	#4	#5
Experience	10 yr	3 mo	4 yr	1 yr	2 mo

TABLE VI. Amount of time used for extending functionality.

Task	#1		#2		#3		#4		#5	
	A	**C**	**A**	**C**	**A**	**C**	**A**	**C**	**A**	**C**
1	7	3	7		DNF			3		2
2	7	2	4		55			4		4
3		6	18		30		16	7		10
4		3	14		DNS		16	2		4
5		7	25		DNS			15	20	14

DNF- Did not Finish
DNS- Did not Start
A - Abstraction
C - Copy-Paste

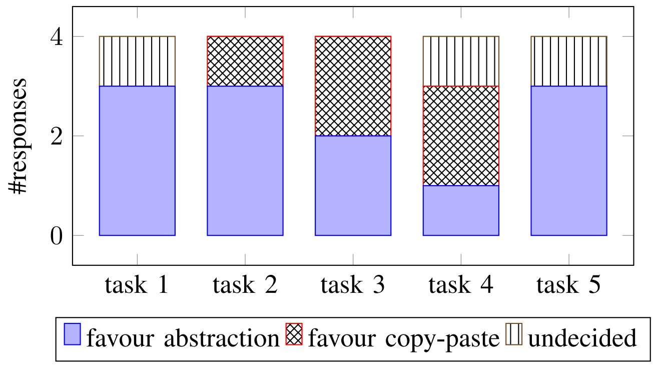


Fig. 6. Preferred results after extending functionality. Out of the 20 answers we received, 3 were undetermined, 5 preferred copy-pasted code, and 12 preferred abstracted code.

References

[1] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–, Washington, DC, USA, 1998. IEEE Computer Society.

[2] Bruno Laguë, Daniel Proulx, Ettore M. Merlo, Jean Mayrand, and John Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 314–321. IEEE Computer Society Press, 1997.

[3] Eunjong Choi, Norihiro Yoshida, Takashi Ishio, Katsuro Inoue, and Tateki Sano. Extracting code clones for refactoring using combinations of clone metrics. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 7–13, New York, NY, USA, 2011. ACM.

[4] M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. ICSM 2009.

[5] E. Duala-Ekoko and M. P. Robillard. Clonetracker: Tool support for code clone management. ICSM 2008.

[6] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[7] Paul Gazzillo and Robert Grimm. SuperC: Parsing all of C by taming the Preprocessor. *SIGPLAN Not.*, 47(6):323–334, June 2012.

[8] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[9] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 485–495, May 2009.

[10] Cory J. Kapser and Michael W. Godfrey. cloning considered harmful considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[11] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 253–262, Washington, DC, USA, 2006. IEEE Computer Society.

[12] Flavio Medeiros, Christian Kästner, Mrcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The love/hate relationship with the C Preprocessor: An Interview Study. ECOOP 2015.

[13] M.Pawlik and N.Augsten. RTED: A Robust Algorithm for the Tree Edit Distance. In *Proceedings of the VLDB Endowment, Vol. 5, No. 4*, 2011.

[14] Christoph Reichenbach, Devin Coughlin, and Amer Diwan. Program Metamorphosis. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 394–418, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] C.K. Roy, K.A. Schneider, and D.E. Perry. Understanding the evolution of Type-3 clones: An exploratory study. MSR 2013.

[16] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor. Stepping stones over the refactoring rubicon. pages 369–393. 2009.

[17] Robert Tairas and Jeff Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Inf. Softw. Technol.*, 54(12):1297–1307, December 2012.

[18] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. VLHCC 2004.