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@em.uni-frankfurt.de

***Abstract*—Software development relies critically 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 have found that copy-paste-modify can be substantially faster 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 shows that copy-paste-modify complicates software maintenance, leading to 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-clones in popular Open Source packages. We define near-clones as any set of method definitions that differ from each other in a few AST nodes. 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 their new features require modifying existing parts of the program. The developers then face a choice: they can either introduce a (possibly complex) abstraction into the existing, working code, or copy, paste, and modify the 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 and pasting produces *near-clones*, which decrease maintainability, but it is non-invasive in that it avoids altering existing, working code.

In practice, we observe that software engineers frequently employ copy-paste-modify. As an example, 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. [10]), 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. However, Juergens et al. [9] have shown that "*cloning can be a substantial problem during development and maintenance*", since "*inconsistent clones constitute a source of faults*".

These results suggest that developers should prefer to avoid duplication, as does practitioner literature [8]; we observed similar responses in an informal poll we conducted with C++ developers (Section II-A). Our poll also measured the amount of time developers needed to introduce changes to foreign code; we observed that at least for the examples we selected, developers were substantially faster when using copy-paste-modify as opposed to introducing abstraction. This suggests that out of the two approaches, copy-paste-modify is the approach that produces the most *immediate* benefit to a given project in a short amount of time, but that abstraction, the other approach, yields superior long-term results.

We propose to address this *re-use discrepancy* with a novel algorithm that offers software developers the best of both worlds. Developers copy, paste, and modify as before, but afterwards invoke a merge algorithm that acts as a refactoring over any number of near-clones, merging them into a single function or method. This merge refactoring is semi-automatic, allowing developers to choose their preferred abstraction mechanism, and it is easy to extend to support additional abstraction mechanisms (e.g., to support project-specific design patterns).

We find that our approach is not only effective at solving the aforementioned re-use discrepancy, but also produces code of sufficient quality to be accepted into 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 [13], [15], 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

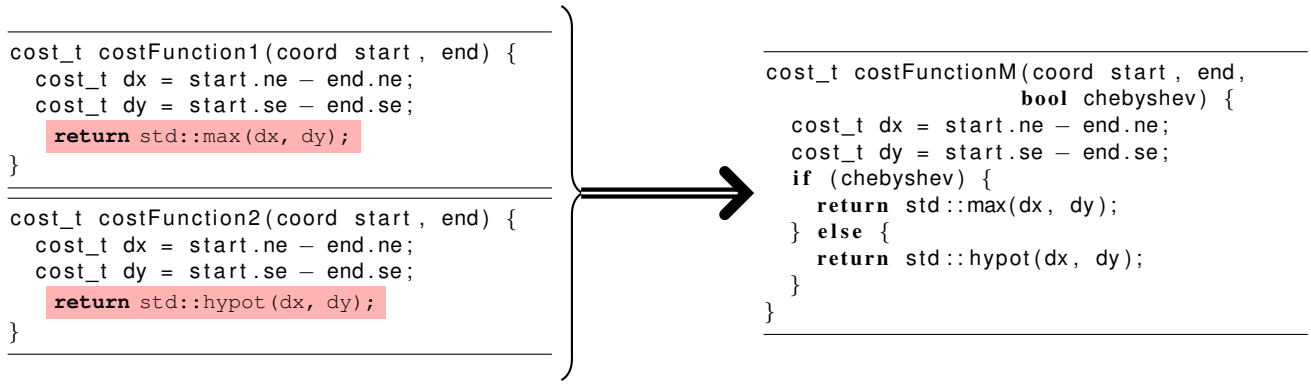

Fig. 1. An example of merging two functions by introducing a boolean parameter and an **if** statement.

sufficiently high quality to be accepted as replacement to unmerged code in the majority of cases.

- 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 introduces the core concepts underlying our algorithm. Section IV describes the merge algorithm in detail. Section V then discusses our implementation. Section VI presents our evaluation on Open Source software. Section VII discusses related work, and Section VIII concludes.

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

Past work in clone detection has found that clones are widespread [2], [10], [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 [1], Facebook HHVM [2], and Facebook Rocksdb [3] Open Source repositories, randomly choosing from a set of near-clones reported by a simple clone detector (Section III). 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) to complete the exact same task again with manual abstraction. Interestingly, 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 IX).

While our numbers are too small to be statistically significant, we argue that they are compelling evidence for copy-paste-modify being 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[4] 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., $\sqrt{dx^2 + dy^2}$). 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.

[1] https://github.com/google/protobuf
[2] https://github.com/facebook/hhvm
[3] https://github.com/facebook/rocksdb
[4] http://openage.sft.mx/

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, 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. Linking Near Clones through Robust Tree Edit Distance

We now look at merging ASTs. To merge AST fragments, we first need to show which parts of each tree disagree with each other. We utilise an existing algorithm called Robust Tree Edit Distance, or RTED [12]. RTED calculates the nodes that are added/removed from one AST when compared to another AST. We use this information for the actual merging phase. We solve this challenge with the help of the *edit distance* and *edit list* of a pair of ASTs, A and B. The edit distance is the number of edit operations needed to transform A into B. RTED defines an edit operation as one of the following:

- **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.

The edit list is the list of edit operations needed to transform A into B. Another benefit of the RTED is that the edit distance can serve as estimate for how closely related two pieces of code are. We use this observation in our evaluation VI to automatically propose code for merging.

IV. Merging Algorithm

We use the term *clone group* to refer to the AST subtrees that we intend to merge. Clone groups can be identified using a clone detector. We used the edit distance to identify clone groups because we felt ASTs with small edit distances intuitively represented the notion of copy-pasted clones. Users can identify clone groups automatically, as we do for our evaluation, or do by hand, as we expect for practical usage of our tool. The merging algorithm works on clone groups of n ASTs of method definitions.

A. Preliminaries

We summarise the preliminary phase of our merge algorithm. We use the this phase to obtain enough information from n ASTs so that they can be merged into a single AST. Consider the code in figure 4, which we will use to illustrate our approach — while the examples here are not all very closely related, our algorithm supports merging all three. Figure 5 shows the ASTs for function1 through function3, slightly simplified to remove distracting details.

To explore how we merge these trees, let us first define the term *node position*, which we will be using frequently from now on. We assume that the nodes are indexed from left to right in each level starting with the leftmost index as 1. This node position is the list of indices of child nodes that need to be traversed (from the root node going down) to reach the specified node. The position of the node 'd' in the tree 1 in Figure 5 is (1,2). We work with node sets and sets of edges between nodes. In order to construct a merged tree from n given ASTs, we compute the following information:

1) The set of nodes that are common to all the ASTs, which the algorithm placed in the merged tree in their respective positions. In the example in Figure 5, this set would be the nodes $\{$a,x,y,z$\}$.
2) The node positions where a merge must be performed along with:
   a) The nodes that the algorithm can merge at each position. When the types of the nodes from the ASTs at this position are not the same, a merge is not possible. The algorithm simply aborts the merging at this point.
   b) The ASTs that these nodes belong to.

We describe how we compute these in the rest of this section. Consider the position (1) in the merged tree in Figure 5: the nodes 'b' from AST 1,2 and the node 'b2' from AST 3 are the merge candidates, i.e, the nodes we want to merge. We begin with the notion of Unique Sets (U in the Figure 2) in order to obtain the mergeable nodes, the ASTs they come from, and the corresponding positions. Unique Sets are the set of nodes that are unique to a particular set of ASTs. Any node may be present in one or more ASTs, but not in all. The Unique sets provide information about these specialisation nodes. Consider the example in Figure 5. The nodes 'b' and 'c' are present only in AST_1 and AST_2. We use the notation $U(\mathsf{AST}_1, \mathsf{AST}_2)$ 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, $U(\mathsf{AST}_1, \mathsf{AST}_2)$ = $\{$b,c$\}$. We can arrive the U sets for all possible subsets of the set of input ASTs by using the node differences between two ASTs. The node difference between two ASTs is the nodes that exist in one AST, but don't exist in the other. Consider the example:
AST1 = (a(b(c))) , AST2 = (a(x(y)))
AST1 - AST2 = $\{$b,c$\}$
We need to introduce operators that distinguish between 'in one AST' and 'in the other AST' in order to merge the trees. 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. We require more than just the

Overview of the steps in the preliminary phase:

1) We begin with a set of ASTs, ALLASTs.
2) We compute the Unique Set for every subset of ALLASTs. Let us call this set U. Unique Sets describe the nodes that are present in the subset under consideration, but in no other ASTs in ALLASTs.
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.

$AST_s = \{\mathsf{AST}_1, \mathsf{AST}_2, \mathsf{AST}_3\}$

$\mathcal{P}(AST_S) = \{\{\mathsf{AST}_1\},\{\mathsf{AST}_2\},\{\mathsf{AST}_3\},\{\mathsf{AST}_1, \mathsf{AST}_2\}$
$,\{\mathsf{AST}_1, \mathsf{AST}_3\},\{\mathsf{AST}_2, \mathsf{AST}_3\},\{\mathsf{AST}_1, \mathsf{AST}_2, \mathsf{AST}_3\}\}$

CN = {a,x,y,z}

U(AST_1) = {d,f1}	Rootset(U(AST_1)) = {d,f1}
U(AST_2) = {e,f2}	Rootset(U(AST_2)) = {e,f2}
U(AST_3) = {b2,f3,n}	Rootset(U(AST_3)) ={b2,f3,n}
U(AST_1,AST_2) = {b,c}	Rootset(U(AST_1,AST_2)) = {b}

Fig. 3. Example sets generated by the Preliminaries

Unique Sets to find these roots. 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 also require that n be an ancestor of itself for our purpose, although that may not be what ancestor usually means. 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. We illustrate an example computation of these sets mentioned in Fig. 2 for the ASTs in Fig. 5 in Fig. 3. The common nodes are the nodes that are unique to all the sets.

Figure 2 summarises at a high level how the algorithm would identify the sets in **3**. We observe precisely one Unique set that is different from its Rootset. This is because the node 'b' is a predecessor 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 the node 'b' from the set $\{\mathsf{AST}_1, \mathsf{AST}_2\}$ and the node b2 from the set $\{\mathsf{AST}_3\}$ are potential merge candidates.

B. Merge Algorithm

In the next step, we compute AST positions for merges from the Rootsets. We begin by constructing an intersection tree, which is simply the intersection of all ASTs. Consider the common nodes generated from **Figure 3** {a,x,y,z}. The algorithm adds these nodes as it is, as illustrated in **Figure 5**. Each Rootset is a set of nodes and each node has a position. Our algorithm assumes that every available position of every node in the Rootsets presents an opportunity to merge. For every available position of every node in the Rootsets, the algorithm

```
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.

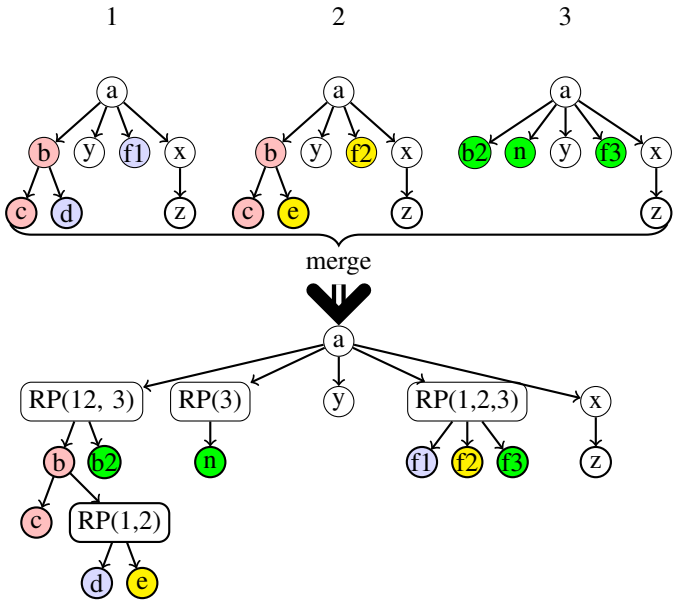

Fig. 5. Example merge

gathers the corresponding nodes with that position and if they are of the same type, the algorithm offers a resolution to the User. The User can pick on a resolution pattern(RP in **5**) based on the **Figure 6**. For the position (1), the node 'b' from the Rootset(1,2) and the node 'b2' from the Rootset(3) would be a potential candidate for merging. We call such points *merge points*. A resolution pattern is a code transformation pattern to resolve the merging of certain kinds of nodes. A resolution pattern will return a node to be inserted at the merge point under consideration. Additionally, it can also 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

Type of Node	Possible RPs (Caller w/ Callee)
Statement	Switch, Conditional Branch w/ Extra Parameter, Field
Literal Expression	Extra Parameter, Field
Type of an argument	Template Parameter
IDExpression	Template Parameter, Extra Parameter

Fig. 6. Available resolution patterns as options presented to the user

separately from the resolution patterns because some resolution patterns may apply to multiple merge points.

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 which could be set to 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 would be the replacement of the node at the merge point of the method RP. 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 possibly changing other classes and introducing super classes. Although conceptually, both the fix-up and merge-substitution could be introduced as one, understanding the separation is important since each resolution pattern could make a modification at the merge points in consideration and perform global modifications, the fix-up.

Any merge of two or more ASTs could involve handling additional nodes introduced in one or more of the ASTs. These node introductions may alter the meaning of existing node positions if the positions touch the new node's parent node. Consider in Fig. 4, the introduction of node n in tree 3. As a side effect, this node introduction alters the positions. The positions of f1,f2, i.e, (3) will be different that of f3, i.e, (4). This actually applies to all nodes that are to the right of n in tree 3. We therefore automatically adjust node positions. We used the edit list to update the paths of affected node.

Consider the example in figure 4. When we merge the ASTs generated for function1 through function3, it looks like Figure 5(4).which in turn might look like the code in fnMerged.

Another concern for readers could be the expense of performing the copy-paste as a pre requisite for invoking our tool. As we have previously established with our user study, copy-paste is definitely faster than manual abstraction. Invoking our tool has almost no overhead as it only involves providing the copy-pasted code as input.

In the following subsections, we discuss a few resolution patterns that were 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 array/pointer specifiers are equal. We maintain a map of the individual parameters to their positions in the merged parameters so that calls can be generated appropriately. For each of the following resolution patterns, we describe the merge resolution and the fix up part.

It is to be noted that the examples used for illustrating the resolution patterns were all real code example of the Open Source repository from Github. We picked these four patterns based on a manual study of various clone groups of methods. 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. Because of this, we can assume a list of Statements to construct the auxiliary AST node. We can assume for our purpose, a Statement node that has two attributes, the Statement itself and the ASTId, denoting the AST they originate from. Let us call our list of Statement alternatives StmtAlt.
Merge-Resolution:

SwitchStmt(ASTId, $[\langle n, \text{stmt}\rangle | \text{stmt} \in \text{StmtAlt}, \text{stmt} \in AST_n]$)
where ASTId *is fresh*

The constructor SwitchStmt takes as input an identifier to switch between (in this case ASTId which is a fresh identifier) and a list of tuples containing the identifier for the case block and the Statement block for each case. The List function takes in one parameter that denotes what is the type of individual elements of the list. Each tuple is constructed from the Stat list. The first element of the tuple is the ASTId of each Statement, and the second element is the Statement itself. ASTId of a node is the AST it belongs to.
Fix-up: We add ASTId 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

```
1  jobject
2      function_openROnly__JLjava(
3      JNIEnv* env, jobject jdb,..) {
4    rocksdb::DB* db = nullptr;
5    rocksdb::Status s;
6    /* About 50 lines of common code */
7    s = rocksdb::DB::OpenForReadOnly(*opt, db_path,
8                                     column_families, &handles, &db);
9    return null;
10 }
11
12
13 jobject
14     function_open__JLjava(
15     JNIEnv* env, jobject jdb,..) {
```

```
16      return
17    rocksdb::DB* db = nullptr;
18    rocksdb::Status s;
19    /* About 50 lines of common code */
20    s = rocksdb::DB::Open(*opt, db_path,
21                          column_families, &handles, &db);
22    return null;
23  }
```

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 25–34):

```
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
10 jobject
11     function_open__JLjava(
12     JNIEnv* env, jobject jdb,..) {
13     return
14   function_open_ROnly__JLjava(
15       env, jdb,.., 2);
16 }
17
18
19 jobject
20     function_open_ROnly__JLjava(
21     JNIEnv* env, jobject jdb,.., int openType) {
22   rocksdb::DB* db = nullptr;
23   rocksdb::Status s;
24   /* About 50 lines of common code */
25   switch(openType) {
26   case 1:
27       s = rocksdb::DB::OpenForReadOnly(*opt,
28       db_path,column_families, &handles,&db);
29       break;
30   case 2:
31       s = rocksdb::DB::Open(*opt,
32       db_path,column_families, &handles,&db);
33       break; }
34   return null;
35 }
```

2) Pattern: Pass Extra Parameter for Literal Expressions: This resolution can be applied if the nodes to be merged are all Constants(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. Otherwise, a merge is not possible. The merge-resolution is a simple id expression that switches between the corresponding constants based on the values passed to the 'newParam', a fresh parameter. We resolve this pattern with an Id expression, which is a node that contains a name of a field or a variable.
Merge-Resolution:

IDExpr(newParam) *where* newParam *is fresh*

Fix-up: We add newParam 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 [5]:

```
1  Handle<Value>
2  Connection::GetClientId
3  (Local<String> property,
4  const AccessorInfo& info)
5  {
6   ...
7    if(!njsConn->isValid_)
8      ...
9    else
10     msg =
11     NJSMessages::getErrorMsg
12     (errWriteOnly, "clientId");
13   NJS_SET_EXCEPTION(msg.c_str(),
14     (int) msg.length());
15   return Undefined();
16 }
17
18 Handle<Value>
19  Connection::GetModule (L
20  Local<String> property,
21  const AccessorInfo& info)
22 {
23   ...
24    if(!njsConn->isValid_)
25      ...
26    else
27      msg =
28      NJSMessages::getErrorMsg
29      (errWriteOnly, "module");
30   NJS_SET_EXCEPTION(msg.c_str(),
31     (int) msg.length());
32   return Undefined();
33 }
34
35
36 Handle<Value>
37  Connection::GetAction
38  (Local<String> property,
39 const AccessorInfo& info)
40 {
41   ...
42    if(!njsConn->isValid_)
43      ...
44    else
45      msg =
46      NJSMessages::getErrorMsg
47      (errWriteOnly, "action");
48   NJS_SET_EXCEPTION(msg.c_str(),
49     (int) msg.length());
50   return Undefined();
51 }
```

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
```

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

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


Handle<Value>
Connection::GetClientId
(Local<String> property,
const AccessorInfo& info)
{
  return
  Connection::GetProperty
  (property, info, "clientId");
}

/* The methods getModule and getAction
are constructed to be similar to getClient*/
```

3) Pattern: Templates for TypeExpressions: We can apply this resolution if the nodes to be merged are all TypeExpressions. The return is a simple IDExpr that is an object of the new typename introduced in the fix up part.
Merge-Resolution:

IDExpr(newParam) *where* newParam *is fresh*

Fix-up: The fix-up introduces a new formal template type parameter to the function definition with a fresh type name. Consider the function snippets taken from the RethinkDB project [6]

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

Our tool would identify that the calls to CreateIntArray and CreateDoubleArray are resolvable using an extra 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 ID Expressions: This resolution can be applied if the nodes to be merged are all Variable identifiers(ID Expressions). An Id expression is a node that contains a name of a field or a variable. We assumed that each of these variables are of the same type. Otherwise, a merge is not possible. The merge-resolution is a simple id expression that switches between the corresponding variable names based on the values passed to the 'newParam', a fresh parameter.
Merge-Resolution:

IDExpr(newParam) *where* newParam *is fresh*

The resolution here is very similar to the pattern 2, except that our algorithm promotes lvalues to pointer-type parameters whenever needed. Passing identifier is more challenging than passing literals. An interesting scenario in this pattern is when the variable is assigned before reference. Consider the example :

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

The algorithm would handle this case by identifying two different merge points each for the identifiers x and z and performing a post processing to link the definition and reference of the variable.
Fix-up: We add newParam as additional formal parameter of the type of the identifiers being merged and the corresponding call sites modified to supply their own identifiers as formal

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

parameters. Consider the function snippets taken from the Facebook's HHVM project [7]

```
1
2  Type typeDiv(Type t1, Type t2)
3   { if (auto t =
4     eval_const_divmod(t1, t2, cellDiv))
5    return *t;
6    return TInitPrim;  }
7  Type typeMod(Type t1, Type t2)
8   { if (auto t =
9     eval_const_divmod(t1, t2, cellMod))
10   return *t;
11   return TInitPrim;  }
```

Our tool would identify that the calls typeDiv and typeMod are resolvable using an extra parameter. Modulo variable renaming and indentation, this produces the following output:

```
1
2  template<class CellOp>
3  Type typeModDiv
4  (Type t1, Type t2, CellOp fun) {
5    if (auto t =
6     eval_const_divmod(t1, t2, fun))
7    return *t;
8    return TInitPrim;
9  }
10
11 Type typeDiv(Type t1, Type t2)
12  { return typeModDiv(t1, t2, cellDiv); }
13 Type typeMod(Type t1, Type t2)
14  { return typeModDiv(t1, t2, cellMod); }
```

V. Implementation

We implemented the distance calculator, the algorithms and the framework on top of Eclipse CDT [8]. We adapted an existing implementation of RTED [9] 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 using our modified edit distance calculator. We explain the details of the actual identification and the merges in the evaluation section VI.

VI. Evaluation

RQ: Are the abstractions performed by our algorithm of sufficient quality for production level code?
In order to evaluate if the abstractions performed by our tool was of sufficient quality for production level code, we first had to come up with clone group candidates to merge. We checked out popular repositories from Github, identified potential candidates for merging, and abstracted the identified candidates using our approach. We finally submitted the abstracted code for pull requests to see how many of them were of sufficient quality to be introduced back in their production code. We explain the details of the process here. We performed a total of 18 abstractions of clone groups and sent them as pull requests to trending Github repositories. Fig **7** 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. The edit distance between pairs of functions seemed more representative of the copy-pasted near clones than traditional clone detectors to filter out potential copy-pasted clone group candidates. Merging is particularly promising for Type-3 clones, yet those are particularly difficult for traditional clone detectors. Therefore, we identified the potential clone groups using edit distance as a metric. We started with the Repositories in 7 and collected all function pairs belonging to the same source file because edit distance applied only to AST pairs. We calculated the edit distance of each pair. Let us call the edit distance of the two functions as *editDist* and the bigger of the two functions as *fnBigger*. We calculate bigger in terms of number of CDT nodes. We marked a function pair as a potentially copy pasted code if the number of nodes in *fnBigger* was greater than a customisable $threshold_n$ and if the ratio of the *editDist* to the *fnBigger* was less than a customisable $threshold_r$. We explain the individual customisations for the two phases of our evaluation in VI-5. We randomly picked function pairs out of the potential candidates and extrapolated clone groups whenever it made sense to our developer intuition, as a developer would do when using our tool in practice. Each clone group contained 2-4 functions. Note that the focus of the approach is the ability of the merging tool to produce abstractions of production level quality and the process of identifying clone groups was to come up with samples for evaluating our algorithm. We 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) using ID Expressions (Named identifiers) and an extra parameter specifying the constant (Pattern 2)
- We resolved differences in Type Expressions using Templates (Pattern 3)
- We resolved differences in ID Expressions using ID Expressions (and pointers if LValue) and 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

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

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

Fig. 7. Repositories with their pull request URLs. Each clone group represents one abstraction.We encourage the readers who choose to look at the pull requests to go through the comments. Although some of the pull requests don't explicitly have the status as "merged", like with the OracleDB and the MongoDB repositories, the codes have actually been merged, indicated by the comments of the Maintainers.

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

5) Results: We performed our initial evaluation (Phase 1) using a very early version of our merging tool which could perform only merges of pairs of functions and did not support multiple resolution patterns for the same pair. In order to verify our claims about acceptability of our abstractions, we had to identify potential clone groups. During Phase 1, we ran our distance calculator on the top trending C++ repositories in Github during the month of December 2014. We set $threshold_r$ to 0.5 and $threshold_n$ to infinity, meaning functions of all sizes were accepted. 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.

Submitted	Accepted	Rejected	Pending
8	1	3	4

Fig. 8. Phase 1 results

We performed our second evaluation (Phase 2) using a complete version of the our merging tool. It was able to merge n functions and could resolve multiple merge points with different node types. In order to verify our claims about quality of our abstractions, we had to identify potential clone groups. 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. One can observe that the threshold are set to very strict numbers, meaning the clones are very similar to each other and are of significant method sizes. We then submitted 10 abstractions as pull requests summarised in the table below and found out that all of them except for one were accepted.

Submitted	Accepted	Rejected	Pending
10	9	1	0

Fig. 9. Phase 2 results

RQ Are the abstractions performed by our code comprehensible?
The results of the evaluation and the accepted pull requests suggest that the codes that were merged as part of the pull requests were indeed comprehensible. As far as we could tell, none of the rejected and pending discussions pointed to any issues with comprehensibility of the merged code. But, it is possible that given multiple merge points and different resolution patterns in the same merge could result in incomprehensible code. Since our approach involves user interaction, the users have the ability to choose not merge certain clones if the merge turns our incomprehensible.

Analysis of rejected and pending results: We present the results of the pending and rejected pull requests summarised in 7 and provide our analysis on the same.
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 and the code in consideration may not be their top priority. We suspect this may be the reason for the pending status of the pull request.
The other pull request from OpenExr repository that contained 3 clone groups merged had a mixture of responses. One maintainer requested an explanation of the advantages and

another maintainer expressed scepticism over the performance overhead of such an abstraction as it was a low level function. One maintainer had requested a unit test of the introduced abstraction before a merge. We could not do this since we lacked the understanding the of the semantics of the functions we merged. All these activities spanned close to 3 months. On closer observation of the activity of the repository, it seems common for the repository's pull requests to stay in the pending status for a long time as is illustrated by the 20 of their 24 open pull requests happening in 2014, a few as far away in the past as March 2014. Based on this, we believe that there are repositories which generally don't prioritise the speed in which they process pull requests. This could have been the reason for the pending status of our pull request in this case.

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.

VII. Related Work

Our work is closely related to existing work on clone detection [2], [10], [1], which 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 [13], [15], 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 [17]. 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 [16], 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. [14] note, Type-3 clones are particularly common and frequently evolve out of Type-1 and 2 clones.

While our work ignores the C preprocessor [11] 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.

VIII. Conclusions

Managing code clones is a significant problem, given the amount of copy-pasted production level code. We proposed to find a middle ground between the easiest way of handling clones, copy-paste and the preferred way, abstraction. We first collected evidence that copy-paste-modify, was the easier method to extend existing functionality by an informal poll. We used another informal poll to collect evidence that abstraction was the preferred mode of extending functionality. After having established that copy-paste is easier and abstraction is preferred, we proposed an approach to abstract code that was possibly copy-pasted. We achieved this by introducing a framework that identifies the merge-able locations of the clones, identifying potential merge resolution mechanisms for these locations and presenting them to the user, who chooses to apply one of the available resolutions per code location. We evaluated this approach by implementing a prototype merging tool and applying a select set of these resolution patterns to near clones in popular GitHub repositories. We submitted these merged versions of the code as pull requests and found out that more than 50%(90% with the most recent version of our tool) of these requests were deemed applicable to industrial codes and merged back.

IX. Appendix: Programmer Poll

This appendix summarises our informal poll. We asked five students (Figure 10) to perform re-use tasks with copy-paste-modify and with manual abstraction; Figure 11 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 12.

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

Fig. 10. Student experience levels

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. 11. Amount of time used for extending functionality.

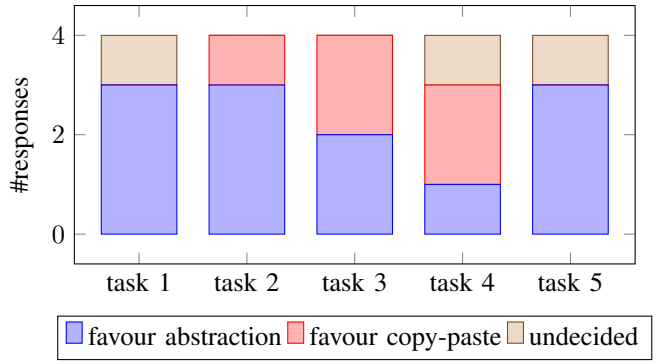

Fig. 12. Preferred results after extending functionality. Out of the 20 answers we got for preferences, 3 were undetermined, 5 preferred copy-pasted code, and 12 preferred abstracted code.

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] 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.

[11] Flavio Medeiros, Christian Kästner, Mrcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The love/hate relationship with the c preprocessor: An interview study. ECOOP 2015.

[12] 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.

[13] 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.

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

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

[16] 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.

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