Bidirectinalizing the C Preprocessor

Yingfei Xiong[1,2], Zhengkai Wu[1,2], Yiming Wu[1,2], Meng Wang[3], Lu Zhang[1,2]

[1]Key Laboratory of High Confidence Software Technologies (Peking University), MoE
[2]Institute of Software, School of EECS, Peking University, China
[3]School of Computing, University of Kent, UK
{xiongyf, 1200012746, yiming.wu, zhanglucs}@pku.edu.cn, m.w.wang@kent.ac.uk

***Abstract*—Many tools directly change programs, such as bug-fixing tools, program migration tools, etc. We call them *program-editing tools*. On the other hand, many programing use the C preprocessor, such as C, C++, and Objective-C. Because of the complexity of preprocessors, many program-editing tools either fail to produce sound results under the presence of preprocessor directives, or give up completely and deal only with preprocessed code.**

In this paper we propose a lightweight approach that enables program-editing tools to work with the C preprocessor for (almost) free. The idea is that program-editing tools now simply target the preprocessed code, and our system, acting as a bidirectional C preprocessor, automatically propagates the changes on the preprocessed code back to the unpreprocessed code. The resulting source code is guaranteed to be correct and is kept similar to the original source as much as possible. We have evaluated our approach on Linux kernel with a set of generated changes. The evaluation results show the feasibility and effectiveness of our approach.

I. Introduction

A lot of program analysis tools involve direct modification of source code. A notable category of such tools is program-repair tools [1], [2], [3], [4]. These tools take a program and a set of tests as input, and modify the program until all tests pass. Another category is API evolution tools [5], [6], [7]. When an API is upgraded with incompatible changes, these tools automatically change the API invocations to comply with the new API. We call such tools that directly modify the source code *program-editing tools*.

On the other hand, many programming languages are provided with preprocessors [8], [9], [10]. The most widely used preprocessor is the C preprocessor (CPP). Many programming languages adopt CPP, notable C, C++, and Objective-C. Furthermore, CPP is often used by programmers in casual situations as a general purpose tool, where the preprocessor is added as a building step for any language used. For example, Korpela [11] describes the use CPP as an HTML authoring tool: instead of writing HTML pages directly, the shared code pieces between HTML pages are first defined as C macros, and HTML pages using these macros are preprocessed into final HTML files.

Program editing tools usually do not directly change preprocessor directives. However the tools must be able to map results back to the unpreprocessed source to be useful. There is no point of fixing a bug in the preprocessed code and only to have it overwritten when the unchanged source is compiled again. This is challenging, as the tools must be able to understand both the preprocessor directives and the target programming languages, and make sure whatever changes made on both levels are consistent with each other. As a matter of fact, existing program-editing tools often fail to produce correct results under the presence of preprocessor directives, or give up dealing with them entirely. We have investigated the implementations of three influential bug-fixing tools on the C programming language: GenProg [1], [2], RSRepair [3], and SemFix [4]. All the three implementations work only on preprocessed code. Users have to manually inspect the preprocessed code, and copy the changes to the original source code—risking of introducing new bugs in the process.

A closely related area is refactoring [12], [13], where tools are expected to directly manipulate preprocessor directives. For example, one may well want to rename a macro or extract a macro as part of the refactoring. In this case, tool builders have no choice but to bite the bullet and confront the preprocessor directly. Typically a new C grammar is designed such that it incorporates both the original C grammar and the preprocessor directives. However, when applied to a more wider range of code editing tools, such almost brute force approaches exhibit obvious shortcomings. First, tool developers using such a grammar basically have to start from scratch: they have to learn the new grammar and leave behind the existing tool chains on C. Second, the effort spend on the new grammars is specialized and cannot be reused for other languages, which basically rules out casual uses of CPP.

In this paper we propose a lightweight approach to support CPP in program-editing tools. Our system acts as a bidirectional CPP: the original preprocessing can be considered as a forward program transformation, and we add to it a corresponding backward transformation that maps changes on the preprocessed code back into the unpreprocessed source. As a result, program-editing tools can now focus on preprocessed programs, and have results (almost) automatically reflected to the source[1].

We list a few examples here: (1) as mentioned above the implementations of the three state-of-the-art bug-fixing approaches only deal with preprocessed code; (2) the API

[1]It is not entirely automatic because the tools need to support the extraction of the changes on the preprocessed programs. Although this step can be performed by generic code differencing [14], extracting the changes directly from the tools gives the best result.

evolution tools mentioned previously can also be implemented more conveniently by only dealing with preprocessed code; (3) all program-editing tools on languages that do not formally rely on CPP naturally fall into this category because the programs may be put under casual uses of CPP.

Although there exist several bidirectionalization techniques [15], [16], [17], they are designed for data transformation: given a source data set `s`, a transformation program `p`, and a target data set `t = p(s)`, these approaches try to reflect the changes on `t` to `s`. The C preprocessor differs from this data transformation scenario in the way that the unpreprocessed source program actually contains both the data set and the transformation program. This added complication amplifies the variance of the backward transformation: when the target data is changed, we may change either the source data set, the transformation program, or both.

A key design novelty in our approach that serves to control this variance is to allow, but at the same time minimize, changes to the transformation programs. First, our approach never introduces new macro definitions or modifies existing macro definitions, effectively confining the impact of the reflected changes to a local scope. Second, our approach only removes existing macro invocations but never invent new ones. Furthermore, we will consider removing macro invocations only when necessary. In this way, we maintain the existing structure of the original program source as much as possible.

Implementing this design is also challenging. Typical approaches to bidirectionalization [15], [16], [18] decompose the transformation along the abstract syntax tree of the program, where each subtree corresponds to a small bidirectional transformation that collectively forms the final transformation. However, a CPP program cannot be easily parsed into a tree structure. For example, in the following piece of code,

```
#define inc(x) 1+x
#define double(x) 2*x
inc(double) 2
inc(double) (x)
```

the first `inc(double)` independently expands to a new segment, but the second `inc(double)` is only part of an expansion, as the expanded `double` will form a new macro invocation with `(x)` to be recursively invoked. As a result, we cannot treat `inc(double)` as an independent unit and derive a backward transformation from it. To overcome this difficulty, we propose a new model for interpreting CPP programs. Instead of parsing a CPP program into an abstract syntax tree, we view the preprocessing as applying a set of rewriting rules to the code. This interpretation enables the bidirectionalization of a CPP program as bidirectionalization of each rewriting rule.

Furthermore, our approach is proved to be correct, in the sense of the following round-trip laws (1) if the preprocessed program is not changed, the unpreprocessed program will not be changed, and (2) preprocessing the unpreprocessed program with the reflected changes will produce exactly the same changed preprocessed program. These two properties are known as GETPUT and PUTGET [19] in the bidirectional transformation literature.

To sum up, our paper makes the following contributions:

- We propose a lightweight approach to handling the C preprocessor in program-editing tools based bidirectional transformations. We analyze different design alternatives and propose five requirements for defining the behavior of the backward transformation, including GETPUT and PUTGET (Section II).
- We propose an algorithm that meets the five requirements. This algorithms is based on an interpretation of CPP as a set of rewriting rules, which structurally decomposes the bidirectionalization of CPP into the bidirectionalizatio of each rule (Section III).
- We evaluate our approach on the Linux kernel and compare our approach with two baseline approaches: one reflecting changes by copying back the entire changed file and one reflecting changes by copying back the changed lines. The evaluation results show that our approach breaks much less macro invocations, and always produces correct results while the other two do not (Section IV).

Finally, we discuss related work in Section V and conclude the paper in Section VI.

II. Problem

A. The C Preprocessor

Table I shows the main preprocessor directives and operators. A preprocessor directive starts with a # at the beginning of the line and ends at the end of the line. Macros can be defined by preprocessor directives but themselves are not preprocessor directives. Basically, we have four main types of preprocessor directives: `#progma` providing compilation options, `#include` for including header files, `#if` for conditional compilation, and `#define` for macro definitions. Additionally, within a macro definition, we can use operators such as `##` and `#`, for concatenating two variables or quoting a variable. Finally, there are some pre-defined macros such as `__FILE__`, which will be replaced by the current values of the fields.

When the C preprocessor processes a source file, it transforms the source file according to the following rules:

- The `#include` and `#if` directives are first expanded, and then the expanded token sequences are scanned.
- For each macro invocation, the arguments are first preprocessed, and then the invocation is expanded.
- If an argument contains # or ##, the unpreprocessed argument is used, otherwise the preprocessed argument is used.
- After a macro invocation is expanded, the expanded token stream is scanned again, where any newly introduced macro invocations are again expanded.
- To avoid infinite recursive expansion, if a macro has been expanded during the expansion process, it will not be expanded again.
- Any new preprocessor directives produced in the expansions will not be used in processing.

As a concrete example, let us consider the code in Figure 1. This example is contrived to exhibit many real world

TABLE I
MAIN PREPROCESSOR DIRECTIVES AND OPERATORS

Directives	Functionality	Example	Result
#progma	Compiler options	#progma once	removed from the preprocessed file
#include	File Inclusion	#include <stdio.h>	the content of "stdio.h"

#if, #ifdef, ...	Conditional compilation	#ifdef FEATURE1
		x = x + 1;
		#endif
			x = x + 1;

#define X	Object-like macro definition	#define X 100 a = X;	a = 100;
#define X(a, b)	Function-like macro definition	#define F(x) x * 100 F(10);	10 * 100;
a ## b	Concatenation	#define X a_##100 X	a_100

#b	Stringification	#define F(x) #x;
		F(hello);
			"hello";

__FILE__, __DATE__, ...	Predefined macros	__FILE__	main.c

```
#define SAFE_FREE(x) if (x) vfree(x);
#define FREE(x) vfree(x);
#define RESIZE(array, new_size, postprocess) \
  g_resize_times++; \
  postprocess(array); \
  array = vmalloc(sizeof(int)*(new_size));
#define GARRAY(x) g_array##x;

RESIZE(GARRAY(2), 100, FREE);
```

Fig. 1. An example of code preprocessing

patterns of macro uses. This piece of code contains four macro definitions and a macro invocation. The first two macro definitions are wrappers for a user-defined free function. The third one is for resizing an array with user-defined memory management and logging. The last one is for naming a set of special global variables. When the code is scanned by the preprocessor, first the arguments of `RESIZE` are processed. In this case, `GARRAY(2)` is expanded into `g_array2`. Though the third parameter `FREE` has been defined as a function-like macro, no argument is provided for `FREE`, so the preprocessor does not treat it as a macro invocation at this stage. Then the macro invocation to `RESIZE` is expanded, leading to the following code:

```
g_resize_times++;
FREE(g_array2);
g_array2 = malloc(sizeof(int)*(100));
```

Now we can see that in the expanded macro, an argument list is actually provided for `FREE`. Then the system will expand `FREE(g_array2)`, resulting in the following code.

```
g_resize_times++;
vfree(g_array2);
g_array2 = malloc(sizeof(int)*(100));
```

In other words, `RESIZE` is in fact a high-order macro where its third parameter is also a macro.

B. Designing the Backward Transformation

Now let us consider that a program-editing tool takes the preprocessed code. Realizing the invocation to `vfree` is unsafe, the programming editing tool inserts a guard before it, as shown below.

```
g_resize_times++;
if (g_array2) vfree(g_array2);
g_array2 = malloc(sizeof(int)*(100));
```

Now let us consider how to design a backward transformation that takes this change as input, and produces a change on the unpreprocessed code, where the changed unpreprocessed code will produce exactly the changed preprocessed code. There are two options: 1. we can change the definition of `RESIZE` macro, adding this guard into the macro definition. 2. we can expand the invocation to `RESIZE` macro, and add the guard to the expanded form. The first option makes global changes, affecting all places that invoke `RESIZE`. The second option makes only local changes, affecting only one place. However, since the backward transformation do not have the knowledge whether the change should be global or local, it is safer to choose the local option. In this example, probably we do not need to add the guard to every invocation to `vfree`, as it will cause unnecessary runtime overhead. The program-editing tool will have the knowledge to determine the safety of `vfree` calls and add this guard to all necessary places. Furthermore, the local option also minimizes the interruption to the original program, as the global option affect many places in the program. This consideration forms our first requirement of the backward transformation.

Requirement 1: A backward transformation should not change any macro definition.

According to the requirement, we should expand the macro invocation with the guard added. Then there are again two options. We may subsequently expand all macro expansions as follows.

```
g_resize_times++;
if (g_array2) vfree(g_array2);                    (1)
g_array2 = malloc(sizeof(int)*(100));
```

Or we may expand only one level as follows.

```
g_resize_times++;
if (GARRAY(2)) FREE(GARRAY(2));                   (2)
GARRAY(2) = malloc(sizeof(int)*(100));
```

We think code piece (2) is better than code piece (1) because it keeps more of the original code structure that are easier to understand, reuse, and maintain. This leads to our second requirement.

Requirement 2: A backward transformation should aim to keep existing macro invocations.

Some might be tempted to go a step further by abstracting the guard into a new macro, or trying to reuse an existing macro that covers its functionality, as in the following.

```
g_resize_times++;
SAFE_FREE(GARRAY(2));
GARRAY(2) = malloc(sizeof(int)*(100));
```

However, this is dangerous as very often macro definitions are not designed to replace all occurrences of its expanded form. For example, Ernst et al. [8] reports a macro definition, `#define ISFUNC 0`, that defines a constant used in certain system calls. It obviously does not mean we shall replace all occurrences of `0` with `ISFUNC`. This leads to our third requirement.

Requirement 3: A backward transformation should not introduce new macro invocations.

Besides the three requirements, we have two additional correctness laws borrowed from the bidirectional transformation literature, namely GETPUT and PUTGET [19]. Let s be the unpreprocessed program, t be the preprocessed program, c_t be the change to t, and c_s be the change to s that is produced by a successful backward transformation.

Requirement 4 (GETPUT): If c_t is empty, then c_s is empty.

Requirement 5 (PUTGET): Let $s' = c_s(s)$ be the new unpreprocessed program obtained by applying c_s to s, and $t' = c_t(t)$ be the changed preprocessed program. Preprocessing s' produces t'.

The requirements together define the behavior of a backward transformation: it should propagate the change back into the unpreprocessed code by means of modifying macro parameters, modifying normal text, and expanding macro invocations, and at the same time expanding as fewer macro invocations as possible.

C. Naive Solutions

Naive solutions does not meet our above listed requirements as we demonstrate in this subsection.

Naive solution I (per-file). The first naive solution is to directly copy back the changed preprocessed files and replace the original unpreprocessed files. This solution is the easiest to implement but has two major deficiencies. First, the original unpreprocessed source files may contain macro definitions, which will be lost if we directly copy back the preprocessed files, causing problems elsewhere. Second, this will leads to the expansion of all macro invocations in the source file, as well as all `#include` directives, destroying modularity completely.

Naive solution II (per-line). The second solution is to use more fine-grained units, copying back only the changed lines. This is doable with the assumption that modern compilers keep a traceability relationship between the lines in the unpreprocessed files and the preprocessed files. For example, when GCC preprocess our running example, it will replace lines 1-7 into empty lines, and put the three statements expanded from line 9 in one line, so that a one-to-one correspondence between lines are kept.

Although the second solution has the benefit of not removing macro definitions, it still has problems. First, a lot of macros are still unnecessarily expanded. In our example, if we copy back the changed line, code piece (1) will be the result, which goes against our second requirement and destroys modularity. The situation becomes even worse when we consider tools that copy lines of code within source files, e.g., GenProg [1] copies statements between two positions in the source files to fix bugs. In such cases all macro invocations in the copied lines will be lost. Second, if the original macro invocations span several lines, the backward mapping produces wrong results. For example, suppose there is a line break within the original macro invocation, as follows.

```
RESIZE(GARRAY(2),
  100, FREE);
```

In GCC, the macro invocation will be expanded into two lines, where the first line contains the three expanded statements, and the second line is empty. As a result, in the backward transformation only the first line will be copied back, resulting in an incorrect program.

III. Approach

As mentioned before, the basic idea of our approach is to interpret CPP as a set of rewriting rules, where the backward transformation reverses each application of the rules. In this section, we first present this interpretation of CPP (Section III-A). Then we describe the change operations we support and first focus on one type of change: replacements. (Section III-B). Since we need to reverse each application of the rules, we first record the applications as rewriting steps (Section III-B), and then derive a backward transformation for each rule, respectively (Section III-C). We also discuss how this process satisfies the requirements (Section III-D1) and an optimization of the algorithm (Section III-D2). Finally, we discuss how to convert other types of changes into replacement (Section III-F).

For the ease of presentation, we shall consider a subset of C preprocessor first: there is no # operator nor ## operator, no #include directive, nor does any macro expansion contain any self invocation. We will later discuss how to extend this simplified model to a full model (Section III-E).

A. Modeling forward preprocessing

We view a CPP program as a sequence of tokens. To recognize preprocessor directives from the sequence of tokens, we rely on two special tokens: # at the beginning of a line, and the end of line following from a # at the beginning of the line. We also assume all macro definitions in scope are stored in a *context*.

We deem the semantics of CPP as a set of rewriting rules. Each rewriting rule is taken the form of `guard` $\hookrightarrow$ `action`. When `guard` is true, `action` is performed to replace some tokens at the beginning of the current remaining token sequence, and set the position for the next rule application.

Both `guard` and `action` are modelled as functions. Function `guard` takes the currently remaining token sequence and the current context, and returns a Boolean value to indicate whether the rule can be applied to the current token sequence or not. Function `action` takes the currently remaining sequence and the current context, and returns four components `(finalized, changed, restIndex, newContext)`. This result indicates that the tokens before `restIndex` are replaced by two consecutive subsequences, `finalized` and `changed`, where `finalized` is a sequence that does not need to be further scanned, and `changed` is a sequence that needs to be scanned. The last component, `newContext`, is the updated context. The algorithm for forward preprocessing is given in Algorithm 1. It repeatedly applies the rules in `R` until the whole token sequence has been processed.

```
Input: token sequence src, rule list R
Output: new token sequence res
ctx ← {};
while src.length > 0 do
    for r ∈ R do
        if r.guard(src, ctx) then
            (finalized, hanged, rest, ctx′) ←
            r.action(src, env);
            break;
        end
    end
    res ← res + finalized;
    src ← changed + src.sub(rest);
    // sub(l) returns a subsequence starting from l
    ctx ← ctx′;
end
```

Algorithm 1: Algorithm for forward preprocessing

There are totally four rules in the rule list `R`. One processes conditionals such as `#if`, `#ifdef`, one processes all other preprocessor directives as we basically need to remove them and updates the context, one processes macro invocations, and the last one processes normal text. These rules are described below.

- R1: This rule processes conditional compilations. Function `guard` determines whether the token sequence starts with a respective directive such as `#if`, `#ifdef`. Function `action` first evaluates the condition based on the current context, and then replaces the whole conditional with either the true branch or the false branch. The whole replacing sequence is returned as `changed` and `finalized` is empty.
- R2: This rule processes other preprocessor directives. Function `guard` determines whether the token sequence starts with a preprocess directive. Function `action` parses the directives, makes necessary changes to the context, and returns the index of the first token after the directive as `restIndex`, with `finalized` and `changed` empty.
- R3: This rule expands a macro invocation. Function `guard` determines whether the first token in the sequence is the name of an object-like macro or the first two tokens are the name of a function-like macro and an open parenthesis. Function `action` consists of two steps.
  - First, the arguments are preprocessed by recursively applying R3 and R4[2]. When expanding an argument, the current context is used, and the token sequence includes only the argument.
  - Second, the occurrences of the parameters in the macro body are substituted by the preprocessed arguments. The new body are returned as `changed`, with `finalized` empty, and `restIndex` is the index of the next token after the macro invocation.
- R4: This rule processes normal tokens not captured by the other rules. Function `guard` always returns true. Function `action` returns the first token as `finalized`, an empty `changed`, and the index of the next tokens as `restIndex`.

As an example, let us consider the following program.

```
#define x 100
hello x
```
(3)

The first applicable rule is R2, which parse the macro definition, store it in the context and moves to the next line. The remaining sequence is now `hello x`. Then R4 is applied to move `hello` into the finalized sequence. Next, R3 is applied to expand `x` into `100`. Finally, `100` is scanned again and R4 moves it into the finalized sequence. Now the remaining sequence is empty and the preprocess stops.

B. Modeling the changes

Program-editing tools can change the program with a variety of operations. In this paper we consider three basic kinds: replacement, insertion, and copy. These operations are summarized from popular bug-fixing tools such as GenProg [1] and SemFix [4], which typically fix a bug by copying or creating an expression to replace another one or inserting at some location.

All the three operations directly manipulate the token sequences. A replacement is a pair `(l, s)`, where `l` is the index of the token to be replaced, and `s` is a token sequence that will replace the token at `l`. An insertion is a pair `(l, s)`, where token sequence `s` will be inserted after the token at index `l`. A copy is a triple (l, l_b, l_e), showing the token sequence between index l_b (inclusive) and index l_e (exclusive) is copied after the token at `l`.

Note that the replacement naturally subsumes deletion. A change `(l, [])` deletes a token, where `[]` denotes an empty sequence.

[2]Including preprocessor directives in the argument list is an undefined behavior in CPP [20]. Here we ignore directives in arguments

In the rest of the section we shall first show how to perform a backward transformation with replacements and then map the other two types of operations to replacements.

C. Rewriting Steps

As mentioned in the introduction, our backward transformation reverses each application of rewriting rules. To perform a backward transformation, we use a data structure to record what rules have been applied to what parts of the source. We call such a record *rewriting steps*. Each rewriting step is a quadruple, `(src, i, ctx, r)`, specifying that rule `r` has been applied to a sub sequence of `src` starting at `i` with context `ctx`. Token sequence `src` is always the current token sequence, containing all modifications made from previous rule applications. Accordingly, a forward transformation is recorded as a sequence of rewriting steps. For example, preprocessing code piece 3 produces the following sequence of rewriting steps: `(P, 0, {}, R2)`, `(hello x, 0, {x}, R4)`, `(hello x, 1, {x}, R3)`, and `(hello 100, 1, {x}, R4)`, where `P` is the original program.

D. Backward Transformation with Replacement

The basic idea of our backward transformation is to propagate the changes backward along the sequence of rewriting steps, such that if we perform a forward preprocessing again, either exactly the same rewriting step is performed, or this rewriting step is replaced by a new sequence of equivalent rewriting steps (whose length can be zero). The algorithm for backward transformation can be found in Algorithm 2. The function `backward` propagates changes along one rewriting step, or reports a failure if no proper propagation can be found.

```
Input: a sequence of rewriting step rs
Input: a set of replacements c
Reverse sequence r;
for r ∈ rs do
    c ← backward(rs, c);
    if backward failed then
        print "Changes cannot be applied.";
        return;
    end
end
return c;
```

Algorithm 2: Algorithm for backward transformation

The key for implementing the backward transformation is to implement the function `backward`. The behavior of `backward` differs for each rewriting rule, and we discuss it from the simplest to the most complex rule.

First, if the rewriting step is performed by R2, the forward preprocessing consumes the preprocessor directive. The backward transformation shifts the replacements according to the length of the consumed macro. For example, consider the following CPP program.

```
#undef hello
hello
```

Now if we have a replacement `(0, x)`, i.e, `hello` is changed into `x`, the backward transformation shifts the replacement into `(4, x)` to ensure that it still changes the macro invocation `hello`.

Second, if the rewriting step is performed by R1, the forward transformation replaces the whole conditional directive with one of its branches. In the backward transformation, we map the changes on the replaced branch back to its original location, and shift other changes if necessary. Since we never change macro definitions, the conditional is guaranteed to evaluate to the same branch.

Third, if the rewriting step is performed by R4, the forward transformation moves one token to `finalized`. The backward transformation needs to ensure that R4 is still performed on the changed token sequence. For example, if the user replaces `hello` in `hello c` with `a b`, we need to check, starting from `a`, whether the only viable path is still to apply R4 twice to preprocess the replacing sequence, i.e., only `guard` of R4 evaluates to true on either `a b c` or `b c`. Here failures may occur. For example, if the user changes `hello` into `x`, where `hello` is not a macro but `x` is an object-like macro. Another example is that the user changes `GARRAY hello` into `GARRAY (hello)`, where `GARRAY` is a function-like macro. In this case, `GARRAY` was previously preprocessed by R4 because it is not followed by an open parenthesis, whereas in the changed sequence `GARRAY` is, and therefore the backward transformation fails at the rewriting step on `GARRAY`. Such failures are necessary as in both the above cases, there is no way to propagate back the changes.

Finally, the case of R3, which expands a macro invocation, is the most complex. R3 consists of two sub steps. We first try to reverse the two sub steps in an reversed order. If any of the sub step fails, we try to expand the macro invocation.

In the second sub step, the occurrences of parameters in a macro body are replaced by expanded arguments. The backward transformation maps the changes back to the arguments. There are two cases where an expansion may be triggered: (1) tokens that are not from the arguments are changed, and (2) multiple occurrences of the same parameter are changed to different values. For example, giving the following piece of code,

```
#define x 1
#define plus(a) a+a
plus(x);
```
(4)

if we changes `1 + 1` into `1 - 1` or `1 + 2`, an expansion will be triggered.

In the first sub step, the arguments are preprocessed. The backward transformation recursively call `backward` to propagate the changes along the rewriting steps of each argument. Finally, a safety check is performed on the propagated changes: if the changed argument contains a comma at the top level (and the comma is not enclosed by a pair of parentheses) or any unmatched parenthesis, we trigger an expansion. This is because a comma or an unmatched parenthesis can break the original structure of arguments.

If the backward transformation on any of the sub step fails, we expand the macro invocation. The expansion will generate changes that replace the macro invocation with its expanded body. As an example, let us assume the second `1` in `1 + 1` is changed into `2` in code piece 4. After backward transformation, the propagated change will replace `plus` in `plus(x)` with `x + 2` and delete `(x)`.

The key to the expansion is how to construct the replacing sequence, `x + 2`. From the forward transformation we know that the unchanged preprocessed token sequence is `1 + 1`, and both `1`s are from the argument `x`. So we copy the rewriting steps of `x` to the locations of the two `1`s, and use the copied rewriting steps to propagate back the changes on the `1`s. The first `1` is not changed, so the original `x` is kept. The second `1` is changed into `2`, so the macro invocation `x` is also expanded, and we get the final text `x + 2`.

However, we cannot always copy the rewriting steps of an argument to the occurrences of its corresponding parameter. Basically, when we copy the rewriting step of an argument, we are assuming that the occurrence of the parameter can be replaced by its unpreprocessed argument, and all rewriting steps behave exactly the same as before. For example, when expanding `plus(x)`, we can replace the two occurrences of `a` by `x`, forming `x + x` where the two `x` expands exactly the same way as the argument `x`, forming `1 + 1`. However, this is not always the case. Let us consider the following code.

```
#define p (x)
#define pplus(x) plus x hello
pplus(p)
```
(5)

With the macros defined in code piece 4, the above code is preprocessed into to `1 + 1 hello`. However, if we need to expand `pplus(p)`, we cannot expand it into `plus p hello`, because it will only expand to `plus (1) hello`. This is because the use of `p` instead `(x)` breaks the original macro invocation.

As a result, we have to add a safety check when we copy rewriting steps. We copy the rewriting steps of an argument only if none of the following conditions is satisfied.

- The preprocessed argument starts with a left parenthesis.
- The preprocessed argument contains a top-level comma.
- The preprocessed argument contains unclosed parentheses.
- The preprocessed argument becomes different if we preprocess it again.

The first three conditions correspond to situations similar to the above example: the preprocessed argument is used as part of another macro invocation in the expanded form, and replacing it with the unpreprocessed one will break the macro invocation. The last condition corresponds to the following case.

```
#define id(x) x
id(plus p)
```

This piece of code expands to `1 + 1`, but if we expands `id(plus p)` into `plus p`, the other macro expansions will be blocked. In other words, with macro invocation, an argument will be scanned twice, but when the macro invocation is expanded, an argument will only be scanned once. We need to make sure this does not affect correctness.

1) Correctness: We can directly reason that the backward transformation satisfies Requirements 1, 3, 4 and 5. Requirement 2 will later be evaluated by our experiments. Requirement 1 holds because we never propagate changes to macro definitions. Requirement 3 holds because if we happen to introduce a new macro invocation, we must have changed a normal token that should be preprocessed by R4 into part of a macro invocation, and the check in R4 will prevent such a change. Requirement 4 holds because we introduce new changes only when we expand a macro invocation, and we would not expand a macro invocation if nothing is changed. Requirement 5 has been reasoned throughout this subsection. To prove it formally, we need to inductively prove that each rewriting step either still behaves the same on the changed steps, or is replaced by a sequence of equivalent rewriting steps (the size of the sequence may be zero). A tricky point is when we copy rewriting steps of arguments to the expanded method body, the copied rewriting steps may be mixed with later rewriting steps. We can show that, still by induction, the order of applying these rewriting steps can be changed without affecting the final result. We omit the details due to space limitations.

2) Optimization: In the current algorithm, we need to record the whole program after each rewriting step, even if only a small portion has changed. In the backward direction, we also need to shift all changes at each step. This contributes to a major performance penalty of the algorithm.

To optimize the algorithm, we first divide the original token sequence into a set of subsequences, where each subsequence is independently preprocessed by a set of rewriting steps. In this way, we can treat each subsequence as an independent program to perform the backward transformation, and then merge the changes. Giving a rewriting step `(src, i, ctx, r)`, if the token at `i` is not generated by a preceding rewriting rule, i.e., is from the original program, then we say the location right before `i` is a *splitting point*. With this definition, we guarantee that no rule application crosses a splitting point. We divide the original program into subsequences along the splitting points, and perform the backward transformation independently for each subsequence before merging the changes. When merging changes from two subsequences, we need to check that the preprocessed version of left sequence and the unpreprocessed version of the right sequence would not form a new macro invocation. This check was performed by R4 in the unoptimized algorithm.

For example, code piece 3 can be divided into three subsequences: the macro definition, `hello`, and `x`. Suppose `plus` is a macro defined earlier. If the user changes `hello` into `plus` and changes `100` into `(x)`, a new macro invocation is formed across the boundary and we should report a failure.

E. Extending to full CPP

In this sub section we discuss how we can extend the above algorithm to support full CPP. Due to space limitations, we only discuss the main ideas without the full details.

To support # and ## operators in forward preprocessing, we need to add two additional types of sub steps in R3, one for stringifying tokens and one for concatenating tokens. Furthermore, parameters used with these operators are replaced by their unpreprocessed forms but not their preprocessed forms, so in the first sub step we need to keep both the unpreprocessed and the preprocessed forms. We need to add three extensions to the backward transformation. First, we need to design the backward transformation for the two new sub steps. Second, we need to add a safety check to determine whether the unpreprocessed form and the preprocessed form are changed consistently. Finally, when expanding a macro, we should not try to recover # and ## operators as the two operators cannot exist on the top level.

To support #include in the forward preprocessing, we need to add another rewriting rule to support #include. In the backward transformation, we need to trace how changes are propagated to each file, and check changes propagated back from different #include directives of the same file are consistent.

Finally, CPP does not allow expanding a macro within its own expansion to prevent infinite loops. To support this, we need to add finer control of the context in our forward rewriting rules. When we are dealing with the tokens expanded from an invocation to macro `m`, the definition of `m` should be removed from the current context.

F. Extending to other types of changes

We have discussed how to deal with replacements. Now let us proceed to insertions and copying. Note that an insertion can be directly converted into a replacement. If we insert a token `y` before a token `x`, we can convert it as replacing `x` with `y x`. However, this direct conversion may cause unnecessary macro expansions. For example, if we insert `y` before `100` in the preprocessed code piece 3, `hello 100`, we should not expand `x` in the backward transformation, but if we model the insertion as replacing `100` with `y 100`, our backward transformation will expand `x` because its body has been changed.

The above example exhibit an insertion at a splitting point, and such an insertion is guaranteed not to expand macros. To reduce the number of unnecessary expansions, We treat the inserted token sequence at a splitting point as an independent sequence and check whether only R4 is applicable to it, since there is no way to put it back if any other rewriting rules can be applied on the inserted sequence. Then we use the same method used in the optimized algorithm to merge the subsequences.

The copy operation is similar to insertion. The only difference is that copied segments may contain macro invocations and we shall try to recover these macro invocations. For this, we perform a special backward transformation. First, we generate changes on the preprocessed code that deletes all tokens except the segments being copied. Then we perform a backward transformation by ignoring rewriting steps with R1, R2, and R3 whose associated macro is not defined or defined differently at the target position. In this way we ensure that only macros that is defined at the target positions are recovered. Then, we insert the token sequence returned by the backward transformation to the target position.

IV. Evaluation

A. Research Questions

In this section we focus on the following research questions.

- **RQ1: Macro Preservation.** According to requirement 2, our approach aims to preserve existing macro invocations. How does the strategy perform on actual programs? How does it compare to other techniques?
- **RQ2: Correctness.** Our approach is guaranteed to be correct according to requirements 4 and 5. How important is this correctness? How does our approach compare to other techniques that do not ensure correctness?
- **RQ3: Failures.** Our approach may report a failure when it cannot find a proper way to propagate the change. How often does this happen? Are the failures false alarms (there exists a suitable change but our approach cannot find it)?

To answer these questions, we conducted a controlled experiment to compare our approach with the two naive approaches described in Section II-C on a set of generated changes on Linux kernel source code. In the rest of the section we describe the details of the experiment.

B. Setup

1) Implementation: We have implemented our approach in Java by modifying JCPP[3], an open source C Preprocessor. We also implemented the two naive approaches in Section II-C for comparison. Our implementation and experimental data can be found on our web site[4].

2) Benchmark: Our experiment was conducted on the Linux kernel version 3.19. We chose Linux source code because Linux kernel is one of the most widely used software projects implemented in C. It contains contributions from many developers, and has a lot of preprocessor directives and macro invocations.

To conduct our experiment, we need a set of changes on the Linux kernel code. Since we concern about how different backward transformations affect preprocessing, we generated changes only in functions that contain macro invocations. To do this, we first randomly selected 180 macros definitions from the kernel code. Since there are far more object-like macros than function-like macros, we would select very few function-like macros if we use pure random selection. So we controlled the ratio between object-like and function-like macros to be 1.5 : 1. Based on the selected macros, we randomly selected

[3]http://www.anarres.org/projects/jcpp/

[4]https://github.com/harouwu/BXCPP

a set of functions which contain invocations to the macros. Finally, we randomly selected 8000 lines from the functions. There are in total 133 macro invocations in the selected lines.

Next we generated a set of changes on the selected lines. To simulate real world changes, we randomly generated two types of changes. The first type is token-level change, in which we randomly replace/delete/insert a token. The second type is statement-level change, in which we delete a statement or copy another statement to the current location. These two types of changes are summarized from popular bug-fixing approaches [1], [3], [21]. The statement-level changes are directly used by GenProg [1] and RSRepair[3]. The token-level changes simulate small changes such as replacing the argument of a method or change an operators used in approaches such as PAR [21].

More concretely, we had a probability `p` to perform an operation on each token, where the operation is one of insertion, replacement and deletion, which had equal probability. The replacement was performed by randomly mutating some characters in the token. The insertion was performed by randomly copying a token from somewhere else. Similarly, we had a probability `q` to perform an operation on each statement, where the operation is copy or deletion. The copied statement was directly obtained from the previous statement. We recognized a statement by semicolon.

Different tools may have different editing patterns: a migration tool typically changes many places in a program, whereas a bug-fixing tool may change a few places to fix a bug. To simulate these two different densities of changes, we used two different set of probabilities. For the high-density changes, we set `p = 0.33` and `q = 0.1`. For the low-density changes, we set `p = 0.1` and `q = 0.05`.

We generated ten sets of changes, five with high-density and five with low-density. The number of the changes generated for each set is shown in Table II.

TABLE II
CHANGES GENERATED FOR THE EXPERIMENT

Low Density	Set	1	2	3	4	5
	Changes	952	885	956	967	884
High Density	Set	6	7	8	9	10
	Changes	3133	3136	3088	3123	3048

3) Independent variables: We considered the following independent variables. (1) *Techniques*, we compared our approach with the two naive solutions, per-file and per-line. (2) *Density of changes*, we evaluated both on the five high-density change sets and the five low-density change sets.

4) Dependent variables: We considered two dependent variables. (1) *Number of remaining macro invocations*. We re-ran the preprocessor after the backward transformation, and counted how macro invocations are expanded during preprocessing. Since none of the techniques will actively introduce new macro invocations, the number of expanded invocations is the number of remaining invocations. To avoid noise from included files, we count only the macro invocations in the current file. (2) *Number of errors*. We re-ran the preprocessor, and compared the new preprocessed program with the previously changed program by Unix file-comparing tool `fc`. Every time `fc` reported a difference, we counted it as an error. (3) *Failures*. Our approach may fail to propagate the changes, and we record whether a failure is reported for each change set.

C. Threats to Validity

A threat to external validity is whether the results on generated changes can be generalized to real world changes. To alleviate this threat, we used different types of changes and different density of changes, in the hope of covering a good variety of real-world changes.

A threat to internal validity is that our implementation of the three approaches may be wrong. To alleviate this threat, we investigated all errors we found in the experiments, to make sure it is a true defect of the respective approach but not a defect in our implementation.

D. Results

TABLE III
EXPERIMENTAL RESULTS

Low Density	Set	1	2	3	4	5
Our Approach	Macros	73	75	72	80	81
	Errors	0	0	0	0	0
	Failures	n	n	n	n	n
Per-Line	Macros	23	25	23	20	26
	Errors	6	7	6	7	7
Per-File	Macros	0	0	0	0	0
	Errors	0	0	0	0	0
High Density	Set	6	7	8	9	10
Our Approach	Macros	47	51	53	48	44
	Errors	0	0	0	0	0
	Failures	n	n	n	n	n
Per-Line	Macros	9	7	7	8	10
	Errors	6	6	7	6	6
Per-File	Macros	0	0	0	0	0
	Errors	0	0	0	0	0

Row "Macros" shows the number of remaining macros. Row "Errors" shows the number of errors caused. Row "Failures" indicates whether a failure is reported in the backward transformation.

The result of our evaluation is shown in Table III. We discuss the results with respect to the research questions below.

1) RQ1: As we can see, our approach preserves macro invocations. Per-line preserves very few macro invocations, while per-file, as we expected, preserves no macro invocations. We further investigated why per-line preserves so few macro invocations. One main reason we found is that there are usually multiple macro invocations per line, and per-line will expand all of them if any tokens in this line is changed.

2) RQ2: Our approach and per-file lead to no errors while several errors are caused by per-line. This is because there are quite a few macro invocations that cross multiple lines. These macros take expressions or statements as argument, which are usually too long to be included in one line.

3) RQ3: As discussed before, our approach may report a failure during the backward transformation. This is usually because the changes accidentally introduce a new macro invocation in the preprocessed code, where there is no way to satisfy PUTGET. However, we do not observe any such cases in our experiment. The reason is that macros usually have special names and it is not easy to collide with a macro name by copying or mutation. Note the other two approaches never report a failure, so the corresponding fields in Table III are left blank.

Although probably being rare in practice, theoretically our approach may report false alarms: our approach reports a failure but a correct change on the source program exists. For example, let consider the following code piece,

```
#define p (x)
plus p
```

where `plus` is the macro defined in code piece (4). After preprocessing, this code piece becomes `plus (x)`. If we change the last parenthesis into `) hello`, our approach reports a failure because first `p` will be expanded and then the expanded content forms a new macro invocation with `plus`. However, there exists a feasible change: replacing `p` with `hello p`.

V. Related Work

A. Bidirectional Transformation

Our work is inspired by research on bidirectional transformation. A classical scenario is the *view-update problem* [22], [23], [24], [25], [26] from database design: a view represents a database computed for a source by a query, and the problem comes when translating an update of the view back to a corresponding update on the source.

Languages have been designed to streamline the development of such applications involving transformations running bidirectionally. Notably the *lenses* framework [27], [28], [19], [29], [30], [31], [32], [33], [34], [35], covering a number of languages that provide bidirectional combinators as language constructs. A different approach is to mechanically transform existing unidirectional programs to obtain a backward counterpart, a technique known as *bidirectionalization* [15], [16], [17], [36], [37], [38], [39], [40], [41], [42]. In the software model transformation literature, the underlying data to be transformed are usually in the form of graphs (instead of trees), and a relational (as oppose to functional) approach that specifies the bidirectional mappings between different model formats is more common [43], [44], [45], [46], [47], [48]. However, the requirement of our work goes beyond what these languages offer: in our framework, not only data, but also transformations (macros) are subject to bidirectional updates.

B. Analyzing and editing unpreprocessed C code

The C preprocessor poses a great challenge for static program analyses. The ability of producing a number of possible preprocessed variants causes a combinatorial explosion, rendering it infeasible to employee traditional tools that are designed to analyze a single variant at a time. Only until very recently, sound parsing and analyzing unpreprocessed C code is made possible through *family-based analyses* [49], [50], [51]. Earlier tools have to resort to unsound heuristics or restrict to specific usage patterns [52], [53], [54].

Similarly, a lot of efforts in refactoring C code are devoted into dealing with multiple variants. Most approaches [55], [56], [57], [13] try to find a suitable model that represent both the C program and the preprocessor directives. A recent approach [58] suggests an alternative: perform refactoring on one variant and prevent the refactoring if problems may be caused in other variants. This is based on the observation that changes on one variant seldom causes problems in other variant.

Unlike these approaches, our approach currently considers only one variant. In the future we may combine our approach with these approaches to deal with multiple variants. However, handling only one variant is already useful in many cases: (1) many programs, though with conditional compilation, do not have many variants; (2) as revealed by Overbey et al. [58], changes in one variant often do not cause problems in other variants.

C. Empirical studies on the C preprocessors

Over the years, there has been no shortage of academic empirical studies that are critical towards the C preprocessor [59], [8], [60], and replacements of CPP are proposed such as syntactical preprocessors [61], [12] and aspect-oriented programming [62], [63], [64] are plenty. However until present, there is no sign of any adoption of these alternatives in industry, with the C preprocessor is still being seen as the tool of the trade [65].

VI. Conclusion

Handling the C preprocessor in program-editing tools is difficult, as a result many tools either produce unsound results or give up on handling CPP entirely. In this paper we show that we can separate the concerns by using bidirectional transformations to deal with the preprocessor, so that program-editing tools may focus only on the preprocessed code, achieving a more modular design.

CPP also represents a family of transformation systems where the transformation program and the source data are bound together, for example bidirectional PHP [66]. Existing approaches [66] often resort to ad-hoc treatment of the bidirectionalizing algorithm and correctness reasoning. The algorithm in this paper indicates a plausible systematic way of bidirectionalizing such systems: treating programs as data and bidirectionalize on the high-level operational semantics. We would like to see a general theory to be developed and applied to many different systems.

Acknowledge

We would like to acknowledge Yangyi Wu at Peking University for the fruitful discussions during the early stage of this work.

REFERENCES

[1] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 54–72, 2012.

[2] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *ICSE*. IEEE, 2012, pp. 3–13.

[3] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *ICSE*, 2014, pp. 254–265.

[4] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *ICSE*, 2013, pp. 772–781.

[5] J. Li, C. Wang, Y. Xiong, and Z. Hu, "Swin: Towards type-safe java program adaptation between APIs," in *PEPM*, 2015, pp. 91–102.

[6] Y. Padioleau, R. R. Hansen, J. L. Lawall, and G. Muller, "Semantic patches for documenting and automating collateral evolutions in linux device drivers," in *PLOS*, 2006.

[7] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: Generating program transformations from an example," in *PLDI*, 2011, pp. 329–342.

[8] M. D. Ernst, G. J. Badros, and D. Notkin, "An empirical analysis of c preprocessor use," *Software Engineering, IEEE Transactions on*, vol. 28, no. 12, pp. 1146–1170, 2002.

[9] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba, "Hygienic macro expansion," in *LISP and functional programming*, 1986, pp. 151–161.

[10] B. Lee, R. Grimm, M. Hirzel, and K. S. McKinley, "Marco: Safe, expressive macros for any language," in *ECOOP*. Springer, 2012, pp. 589–613.

[11] J. Korpela, "Using a c preprocessor as an html authoring tool," http://www.cs.tut.fi/~jkorpela/html/cpre.html, accessed on Jan 8, 2015, 2000.

[12] B. McCloskey and E. Brewer, "Astec: A new approach to refactoring c," in *ESEC/FSE-13*, 2005, pp. 21–30.

[13] A. Garrido and R. Johnson, "Embracing the c preprocessor during refactoring," *Journal of Software: Evolution and Process*, vol. 25, no. 12, pp. 1285–1304, 2013. [Online]. Available: http://dx.doi.org/10.1002/smr.1603

[14] B. Fluri, M. Wursch, M. PInzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *Software Engineering, IEEE Transactions on*, vol. 33, no. 11, pp. 725–743, 2007.

[15] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi, "Bidirectionalization transformation based on automatic derivation of view complement functions," in *ICFP*, R. Hinze and N. Ramsey, Eds. ACM, 2007, pp. 47–58.

[16] J. Voigtländer, "Bidirectionalization for free! (pearl)," in *POPL*, Z. Shao and B. C. Pierce, Eds. ACM, 2009, pp. 165–176.

[17] J. Voigtländer, Z. Hu, K. Matsuda, and M. Wang, "Combining syntactic and semantic bidirectionalization," in *ICFP*. ACM, 2010, pp. 181–192.

[18] K. Matsuda, S.-C. Mu, Z. Hu, and M. Takeichi, "A grammar-based approach to invertible programs," in *ESOP*, ser. Lecture Notes in Computer Science, A. D. Gordon, Ed., vol. 6012. Springer, 2010, pp. 448–467.

[19] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, "Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 3, May 2007.

[20] ISO/IEC JTC 1, SC 22, WG 14, "Programming languages – C," Committee draft, International Organization for Standardization, April 2011.

[21] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *ICSE*, 2013, pp. 802–811.

[22] F. Bancilhon and N. Spyratos, "Update semantics of relational views," *ACM Trans. Database Syst.*, vol. 6, no. 4, pp. 557–575, 1981.

[23] U. Dayal and P. A. Bernstein, "On the correct translation of update operations on relational views," *ACM Trans. Database Syst.*, vol. 7, no. 3, pp. 381–416, 1982.

[24] S. J. Hegner, "Foundations of canonical update support for closed database views," in *ICDT*, ser. Lecture Notes in Computer Science, S. Abiteboul and P. C. Kanellakis, Eds., vol. 470. Springer, 1990, pp. 422–436.

[25] Y. Cui, J. Widom, and J. L. Wiener, "Tracing the lineage of view data in a warehousing environment," *ACM Trans. Database Syst.*, vol. 25, no. 2, pp. 179–227, Jun. 2000. [Online]. Available: http://doi.acm.org/10.1145/357775.357777

[26] L. Fegaras, "Propagating updates through xml views using lineage tracing," in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, March 2010, pp. 309–320.

[27] Z. Hu, S.-C. Mu, and M. Takeichi, "A programmable editor for developing structured documents based on bidirectional transformations," in *PEPM*, N. Heintze and P. Sestoft, Eds. ACM, 2004, pp. 178–189.

[28] S.-C. Mu, Z. Hu, and M. Takeichi, "An algebraic approach to bidirectional updating," in *APLAS*, ser. Lecture Notes in Computer Science, W.-N. Chin, Ed., vol. 3302. Springer, 2004, pp. 2–20.

[29] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt, "Boomerang: resourceful lenses for string data," in *POPL*, G. C. Necula and P. Wadler, Eds. ACM, 2008, pp. 407–419.

[30] J. N. Foster, A. Pilkiewicz, and B. C. Pierce, "Quotient lenses," in *ICFP*, J. Hook and P. Thiemann, Eds. ACM, 2008, pp. 383–396.

[31] M. Wang, J. Gibbons, K. Matsuda, and Z. Hu, "Gradual refinement: Blending pattern matching with data abstraction," in *MPC*, ser. Lecture Notes in Computer Science, C. Bolduc, J. Desharnais, and B. Ktari, Eds., vol. 6120. Springer, 2010, pp. 397–425.

[32] Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, and F. Orejas, "From state- to delta-based bidirectional model transformations: The symmetric case," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, J. Whittle, T. Clark, and T. KÃijhne, Eds. Springer Berlin Heidelberg, 2011, vol. 6981, pp. 304–318. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24485-8_22

[33] M. Hofmann, B. Pierce, and D. Wagner, "Edit lenses," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: ACM, 2012, pp. 495–508. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103715

[34] N. Foster, K. Matsuda, and J. Voigtländer, "Three complementary approaches to bidirectional programming," in *SSGIP*, ser. Lecture Notes in Computer Science, J. Gibbons, Ed., vol. 7470. Springer, 2010, pp. 1–46.

[35] R. Rajkumar, S. Lindley, N. Foster, and J. Cheney, "Lenses for web data," In Preliminary Proceedings of Second International Workshop on Bidirectional Transformations (BX 2013), 2013.

[36] M. Wang, J. Gibbons, and N. Wu, "Incremental updates for efficient bidirectional transformations," in *ICFP*, M. M. T. Chakravarty, Z. Hu, and O. Danvy, Eds. ACM, 2011, pp. 392–403.

[37] J. Voigtländer, Z. Hu, K. Matsuda, and M. Wang, "Enhancing semantic bidirectionalization via shape bidirectionalizer plug-ins," *J. Funct. Program.*, vol. 23, no. 5, pp. 515–551, 2013.

[38] M. Wang, J. Gibbons, K. Matsuda, and Z. Hu, "Refactoring pattern matching," *Sci. Comput. Program.*, vol. 78, no. 11, pp. 2216–2242, 2013.

[39] K. Matsuda and M. Wang, "Flippr: A prettier invertible printing system," in *Programming Languages and Systems*, ser. Lecture Notes in Computer Science, M. Felleisen and P. Gardner, Eds. Springer Berlin Heidelberg, 2013, vol. 7792, pp. 101–120. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37036-6_6

[40] ——, "Bidirectionalization for free with runtime recording: or, a lightweight approach to the view-update problem," in *PPDP*, R. Peña and T. Schrijvers, Eds. ACM, 2013, pp. 297–308.

[41] M. Wang and S. Najd, "Semantic bidirectionalization revisited," in *PEPM*, W.-N. Chin and J. Hage, Eds. ACM, 2014, pp. 51–62.

[42] K. Matsuda and M. Wang, ""Bidirectionalization for free" for monomorphic transformations," *Science of Computer Programming*, 2014, dOI: 10.1016/j.scico.2014.07.008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642314003323

[43] I. O. Object Management Group, "Meta object facility (mof) 2.0 query/view/transformation specification," https://hackage.haskell.org/package/lens.

[44] P. Stevens, "Bidirectional model transformations in qvt: semantic issues and open questions," *Software & Systems Modeling*, vol. 9, no. 1, pp. 7–20, 2010. [Online]. Available: http://dx.doi.org/10.1007/s10270-008-0109-9

[45] A. Schürr, "Specification of graph translators with triple graph grammars," in *Graph-Theoretic Concepts in Computer Science*, ser. Lecture Notes in Computer Science, E. Mayr, G. Schmidt, and G. Tinhofer, Eds. Springer Berlin Heidelberg, 1995, vol. 903, pp. 151–163. [Online]. Available: http://dx.doi.org/10.1007/3-540-59071-4_45

[46] A. Schürr and F. Klar, "15 years of triple graph grammars," in *Graph Transformations*, ser. Lecture Notes in Computer Science, H. Ehrig, R. Heckel, G. Rozenberg, and G. Taentzer, Eds. Springer

Berlin Heidelberg, 2008, vol. 5214, pp. 411–425. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87405-8_28

[47] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano, "Bidirectionalizing graph transformations," in *ICFP*, P. Hudak and S. Weirich, Eds. ACM, 2010, pp. 205–216.

[48] S. Hidaka, Z. Hu, K. Inaba, H. Kato, and K. Nakano, "Groundtram: An integrated framework for developing well-behaved bidirectional model transformations," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 480–483. [Online]. Available: http://dx.doi.org/10.1109/ASE.2011.6100104

[49] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 805–824. [Online]. Available: http://doi.acm.org/10.1145/2048066.2048128

[50] P. Gazzillo and R. Grimm, "SuperC: Parsing all of c by taming the preprocessor," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 323–334. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254103

[51] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, "Scalable analysis of variable software," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 81–91. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491437

[52] I. D. Baxter and M. Mehlich, "Preprocessor conditional removal by simple partial evaluation," in *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, ser. WCRE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 281–. [Online]. Available: http://dl.acm.org/citation.cfm?id=832308.837146

[53] A. Garrido and R. Johnson, "Analyzing multiple configurations of a c program," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ser. ICSM '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 379–388. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2005.23

[54] Y. Padioleau, "Parsing c/c++ code without pre-processing," in *Proceedings of the 18th International Conference on Compiler Construction: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ser. CC '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 109–125. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00722-4_9

[55] A. Garrido and R. Johnson, "Challenges of refactoring c programs," in *Proceedings of the International Workshop on Principles of Software Evolution*, ser. IWPSE '02. New York, NY, USA: ACM, 2002, pp. 6–14. [Online]. Available: http://doi.acm.org/10.1145/512035.512039

[56] M. Vittek, "Refactoring browser with preprocessor," in *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, March 2003, pp. 101–110.

[57] D. Spinellis, "Global analysis and transformations in preprocessed languages," *IEEE Trans. Softw. Eng.*, vol. 29, no. 11, pp. 1019–1030, Nov. 2003. [Online]. Available: http://dx.doi.org/10.1109/TSE.2003.1245303

[58] J. L. Overbey, F. Behrang, and M. Hafiz, "A foundation for refactoring c with macros," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 75–85. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635908

[59] H. Spencer and G. Collyer, "#ifdef considered harmful, or portability experience with c news," 1992.

[60] J. Liebig, C. Kästner, and S. Apel, "Analyzing the discipline of preprocessor annotations in 30 million lines of c code," in *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, ser. AOSD '11. New York, NY, USA: ACM, 2011, pp. 191–202. [Online]. Available: http://doi.acm.org/10.1145/1960275.1960299

[61] D. Weise and R. Crew, "Programmable syntax macros," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, ser. PLDI '93. New York, NY, USA: ACM, 1993, pp. 156–165. [Online]. Available: http://doi.acm.org/10.1145/155090.155105

[62] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat, "A quantitative analysis of aspects in the ecos kernel," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, ser. EuroSys '06. New York, NY, USA: ACM, 2006, pp. 191–204. [Online]. Available: http://doi.acm.org/10.1145/1217935.1217954

[63] B. Adams, W. De Meuter, H. Tromp, and A. E. Hassan, "Can we refactor conditional compilation into aspects?" in *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, ser. AOSD '09. New York, NY, USA: ACM, 2009, pp. 243–254. [Online]. Available: http://doi.acm.org/10.1145/1509239.1509274

[64] Q. Boucher, A. Classen, P. Heymans, A. Bourdoux, and L. Demonceau, "Tag and prune: A pragmatic approach to software product line implementation," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 333–336. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859064

[65] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, , and R. Gheyi, "The love/hate relationship with the C preprocessor: An interview study," in *ECOOP*, 2015, p. to appear.

[66] X. Wang, L. Zhang, T. Xie, Y. Xiong, and H. Mei, "Automating presentation changes in dynamic web applications via collaborative hybrid analysis," in *FSE*, 2012, p. 16.