TBCppA: a Tracer Approach for Automatic Accurate Analysis of C Preprocessor's Behaviors

Katsuhiko Gondow
Tokyo Institute of Technology
gondow@cs.titech.ac.jp

Hayato Kawashima
New Glass Forum
hayato-k@newglass-lab.jp

Takashi Imaizumi
Chiba University
imaizumi_takashi@faculty.chiba-u.jp

Abstract

C preprocessor (CPP) is a major cause that makes it much difficult to accurately analyze C source code, which is indispensable to refactoring tools for C programs. To accurately analyze C source code, we need to generate CPP mapping information between unpreprocessed C source code and preprocessed one. Previous works generate CPP mapping information by extending the existing CPP, which results in low portability and low maintainability due to the strong dependency of CPP implementation.

To solve this problem, this paper proposes a novel approach (called `TBCppA`) based on tracer*, which generates CPP mapping information by instrumenting the unpreprocessed C source code using XML-like tags called "tracers". The advantage of `TBCppA` is high portability and high maintainability, which the previous methods do not have. We successfully implemented a first prototype of `TBCppA`, and our preliminary evaluation of applying `TBCppA` to `gcc-4.1.1` produced promising results.*

1. Introduction

The programming language C [22, 23] is widely used even now, especially in operating systems, device drivers, embedded systems, language processors like GCC, and server applications like Apache. On the other hand, C is widely criticized since it is very easy for programmers to write undesirable code (e.g., buffer overflows). This implies we need high-quality program development tools for C to reduce undesirable code or bugs in C source code, but there are, unfortunately, few good tools in practice, mainly because it is very hard to cope with the problems of C preprocessor (CPP), pointer aliases in C, etc. For example, [16] reports the existing call-graph extractors are very imprecise, and [7, 14, 19, 21] report the functionality of the existing code refactoring tools is very limited due to the CPP problems. This paper focuses on the CPP problems.

In brief, the CPP problems are twofold: (See also Section 2.2 for more details)

- It is very hard to directly parse unpreprocessed C programs like Fig. 1 (i.e., C programs with CPP directives) using the C grammar, simply because they are syntactically incorrect.
- It is possible to parse preprocessed C programs (i.e., C programs with no CPP directives) using the C grammar. But it is very hard to recover the unpreprocessed version with CPP directives and macro calls from preprocessed one, because the information about CPP processing and macro expansions, which we call *CPP mapping information*, is not available.

```
/* gzip-1.2.4, gzip.c, line 888 */
#ifdef NO_FSTAT
  if (stat(ofname, &ostat) != 0) {
#else
  if (fstat(ofd, &ostat) != 0) {
#endif
    fprintf(stderr, "%s: ", progname);
  }
```

Figure 1. Example where braces are not balanced in unpreprocessed C source code.

```
1 /* foo.c */
2 #define  N        10
3 printf ("%d",      N     );
```

⇓

Step 1: `TBCppA` embeds XML-like tracers in C program.

⇓

```
1 /* TBCppA/foo.c */
2 #define  N     <d>10</d>
3 printf ("%d",  <c> N</c> );
```

⇓

Step 2: native CPP (e.g., `gcc -E`) preprocesses the program including the tracers.

⇓

```
1 /* foo.c.xml */
2
3 printf ("%d",  <c> <d>10</d></c> );
```

⇓

Step 3: `TBCppA` generates the CPP mapping information by analyzing the resulting tracers.

⇓

```
[CPP mapping info. (part)] Macro call N in line 3 was
expanded to 10, whose definition is in line 2.
```

Figure 2. `TBCppA`'s rough idea of how to use tracers to observe macro expansions.

CPP mapping information includes the information about macro definition, macro expansion, file inclusion, conditional compilation, and their locations (line number etc.) in the original source code.

978-0-7695-3353-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCAM.2008.13

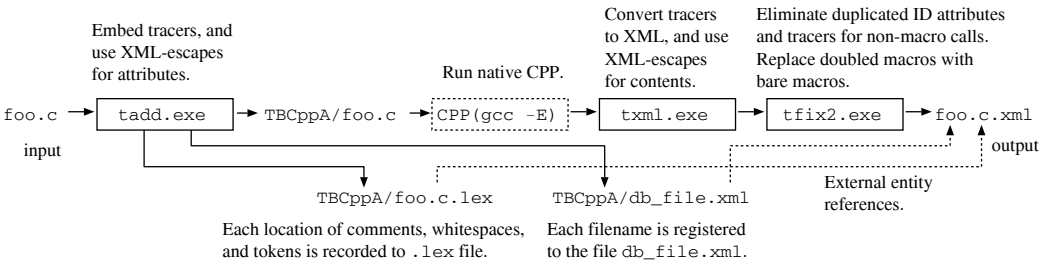

Figure 3. TBCppA's components and process flow chart

Previous papers [1–4, 8, 9, 11–13, 15, 17–21, 25, 27, 28] generate CPP mapping information using various methods, but all of them have drawbacks, mentioned in Section 2.3.

To solve the CPP problems, this paper proposes a novel approach based on *tracer*, which generates CPP mapping information by instrumenting the unpreprocessed source program using XML-like tags called "tracers". We call the approach TBCppA (tracer-based CPP analyzer) [1]. TBCppA generates CPP mapping information using the existing CPP as is. Thus TBCppA does not suffer from almost all the problems mentioned in Section 2.2.

The term *tracer* has been used, for example, in biological or ecological fields to refer to an identifiable substance (e.g., radioactive isotope), which makes it easy to observe the behavior or distribution of the observee, by attaching tracer to the observee and observing the tracer.

TBCppA generates CPP mapping information by attaching special tokens as tracers to a sequence of tokens in the unpreprocessed C source code, running native CPP "as is" to preprocess the code including tracers, and finally observing the changes of tracers.

In Fig. 2, for example, TBCppA first attaches tracers like <d> and </d>[2] to the unpreprocessed C code, and TBCppA runs the native CPP. As a result, the macro N between the tracers <c> and </c> is expanded to <c><d>10</d></c>. By comparing <c><d>10</d></c> with <c>N</c>, TBCppA knows that the macro N is expanded to 10. See also Fig. 3 here. Fig. 3 illustrates the basic components and the process flow of TBCppA, whose details are given in Section 4.2.

To show our idea of TBCppA is feasible, this paper provides the concrete design and implementation of TBCppA. The preliminary results of evaluating TBCppA suggest that TBCppA approach is quite feasible and useful both in performance and functionality. The contributions of our TBCppA in this paper are summarized as follows.

- High accuracy: TBCppA generates highly accurate CPP mapping information, even about ambiguous semantics in C (i.e., implementation-defined, and unspecified behaviors in C) and CPP-specific predefined macros, since TBCppA uses the existing CPP as is.
- High portability and high maintainability: TBCppA does not emulate the existing CPP behaviors or does not modify the existing CPP implementations. Instead, to expand tracer-embedded macros, TBCppA just uses the existing CPPs as is. Although the design and implementation of TBCppA is not trivial, the code size of TBCppA (ver 0.0.3b) is small; it consists of only around 3,700 lines. These positive characteristics makes TBCppA highly portable and maintainable. Actually, our current TBCppA implementation runs on Windows Cygwin, Linux and Mac OS X.
- Wide applicability for large open source code: TBCppA can process almost all C programs. In our preliminary evaluation, for example, TBCppA successfully processed the source code of gcc-4.1.1 (around 630,000 lines).
- High applicabilitiy for C tools: The mapping information produced by TBCppA can be easily applied for C tools. To show this, we successfully developed three small but practical tools (tmacro, tifdef_src2html, trecov_demo) in our preliminary evaluation.

This paper is organized as follows. Section 2 gives the existing approaches and their drawbacks. Section 3 introduces the definition of TBCppA tracers. Section 4 describes the design and implementation of TBCppA. Section 5 gives a preliminary evaluation of our TBCppA implementation and its small but practical applications. Section 6 gives the limitations of TBCppA. Section 7 describes related work. Finally, Section 8 gives conclusion and future work.

2. Background: CPP problems

2.1. Overview of CPP

The main features of CPP are file inclusion, macro definition and conditional compilation, which are specified by CPP directives like #include, #define, #ifdef, respectively. The C language has two grammars: CPP grammar for CPP directives and macro calls, and C grammar for other nonterminals like identifiers, expressions and statements. First, C programs with CPP directives need to be parsed by CPP using the CPP grammar and evaluated while removing CPP directives and substituting macros. This process is called *preprocessing*. After preprocessing, C programs have no CPP directives and thus can be parsed by the C parser using the C grammar.

Using CPP is inevitable in real C programming, and actually almost all C programs use CPP features. An empiri-

[1] We also uses TBCppA to refer to our prototype implementation based on TBCppA approach.

[2] Tracers used here are much simplified for readability. Actual tracers defined in Section 3 are much more complex.

cal study by Ernst et al. [5], for example, reports that "preprocessor directives make up 8.4% of program lines" and "macros pervade the code, with 0.28 uses per line". This fact led them to say "an effective program analysis tool must address the preprocessor" [5].

2.2. CPP problems

The problems of CPP is twofold: (Also refer to [5, 6].)

- **CPP problem 1:** It is very hard to parse unpreprocessed C programs (i.e., C programs with CPP directives) directly using the C grammar[3]. For example, a code fragment in Fig. 1 is unpreprocessed and the braces in the code are not balanced. Thus the code fragment is syntactically incorrect, which means that it cannot be parsed by the C grammar.

  As shown in Fig. 1, a CPP directive with a preceding newline can appear between any tokens. Furthermore, CPP can replace a keyword (e.g., `const`) with any sequence of tokens (e.g., empty string). Thus, CPP is so powerful and unstructured, which provides great flexibility for C programmers, and also great difficulty of directly parsing unpreprocessed code.
- **CPP problem 2:** It is possible to parse preprocessed C programs (i.e., C programs with no CPP directives) using the C grammar. But it is very hard to recover the unpreprocessed version with CPP directives and macro calls from preprocessed one, because *CPP mapping information*, described in Section 1, is not available. For example, GCC-3.4.3's CPP expands the macro `stdin` to `(&__iob[0])` on Solaris 10, but the compiler gives no information about this macro expansion, which is required to recover `stdin` from `(&__iob[0])`.

Some tools (e.g., refactoring tool) require *accurate* CPP mapping information. This is because, for example, refactoring tools must ensure program behavior is always preserved. For such tools, it is practically impossible to solve the CPP problem 1. Tools like GNU GLOBAL approximately and partially parse the unpreprocessed C code, so they can not provide accurate CPP mapping information due to approximation. There is an approach that restricts CPP directives to appear at certain places in the grammar. Unfortunately, there are many real-world source codes that this approach cannot handle. Fig. 1 just shows the case. Thus, both approaches are not acceptable.

2.3. The existing approaches and their drawbacks

Solving the CPP problem 2 is important. Without the CPP mapping information, the tool's output lacks the abstraction provided by CPP and thus it is not understandable nor readable. For example, the tool user sees `(&__iob[0])`, not `stdin`, as output.

Previous papers [1–4, 8, 9, 11–13, 15, 17–21, 25, 27, 28] generate the CPP mapping information by the following methods, but all of them have drawbacks mentioned below.

[3]This seems a matter of course, but some people say it is possible, since some tools (e.g., GNU GLOBAL) attempt to do this (in inaccurate ways).

- **Method 1**: Partially and/or approximately parsing unpreprocessed source code (e.g., srcML [3]).
- **Method 2**: Using compiler options (e.g., GCC options `-E -dM`) or using debugging information (e.g., DWARF2-XML [25]).
- **Method 3**: Implementing a CPP emulator with the feature of outputing CPP mapping information [4, 8, 9, 11, 19–21, 27, 28].
- **Method 4**: Modifying a real CPP to output the CPP mapping information [1, 2, 15, 17, 18] (e.g., PCp^3 [1,2] modified the GCC's CPP called Cpplib).
- **Method 5**: Using API in some language environments to obtain CPP mapping information (e.g., cppML [12, 13] for C++ uses the API in IBM VisualAge C++).

Each method above has the following drawbacks.

- **Drawback of Method 1**: **Method 1** provides inaccurate information, since parsing is incomplete. Typically, two different symbols with the same name (e.g., a variable `foo` and a function `foo`) are regarded as the same, and scopes are incorrectly maintained (e.g., in the case of Fig. 1).
- **Drawback of Method 2**: Typical compilers (e.g., GCC) and debugging information (e.g., DWARF2) do not provide the information about macro expansions, which plays an important role in the CPP mapping information, although this method readily provides the correct information about macro definitions.
- **Drawback of Method 3**: **Method 3** provides inaccurate information, since the accurate emulation of CPP is difficult due to the following issues:
  - Implementation-defined or unspecified behaviors in the CPP specification. For example, the order of evaluation of # and ## is unspecified (6.10.3 in [23]). The search path for the files enclosed with < and > in `#include` directives is implementation-defined (6.10.2 in [23]).
  - Compiler-specific CPP extensions. For example, GCC supports `#include_next` directive and variadic macros. GCC's variadic macros have different syntax from C99's.
  - Platform-specific or implementation-specific (pre-defined) macros. How system-defined macros (e.g., `stdin`) are expanded varies according to how they are defined in the system header files (e.g., `stdio.h`). Some macros are not defined in any header files. For example, CPP of GCC-3.4.4/Cygwin-1.5.19 defines 79 pre-defined macros (e.g., `unix`, `i386`), all of which are not defined in header files.
  - Difficulty conforming to the CPP specification and mimicking the existing implementation. At a glance, the CPP behavior looks simple, but not in practice. For example, CPP in GCC-3.4.4 consists of around 15,000 lines (including comments and empty lines). Thus the accurate emulation of CPP is simply difficult because the CPP specification and implementation is large and complex.

- **Drawback of Method 4**: This method suffers from low portability and low applicability problems. Modified CPP can run often only on the specific platforms, which results in less applicability. The cost of porting the modified CPP and maintaining it continuously (e.g., whenever new GCC or Linux is released) would be expensive. For example, the source code of PCp^3 is open to the public, but, as far as we know, PCp^3 has not been maintained since 1999. The source code of PCp^3 is based on very old CPP on GCC-2.7.2.2, so we cannot use PCp^3 as is.
- **Drawback of Method 5**: This method is only available in specific language environments like IBM VisualAge C++ [12], and not available in major compilers like GCC. To what extent this method is feasible when using IBM VisualAge C++ is still unknown, since sample cppML-tagged source code in cppML homepage [13] seems not to include macro expansions.

```
1 /* foo.c */
2 #define  N        10
3 printf ("%d",      N     );
```

⇓

Step 1: TBCppA embeds XML-like tracers in C program.

⇓

```
1 /* TBCppA/foo.c */
2 @"define id='F4_1' dir='#define' name='N'
3         macro_body='10' first_line='2' /"
4 #define N   @"macro_body ref='F4_1'" \
5                          10 @"/macro_body"
6 printf("%d",
7    @"macro_call name='N' first_line='3'"
8    N
9    @"/macro_call" );
```

⇓

Step 2: native CPP (e.g., `gcc -E`) preprocesses the program including the tracers.

⇓

```
1 /* foo.c.xml */
2 <define id="F4_1" dir="#define" name="N"
3         macro_body="10" first_line="2" />
4
5 printf (&quot;%d&quot;,
6  <macro_call name="N" first_line="3">
7  <macro_body ref="F4_1"> 10 </macro_body>
8  </macro_call> );
```

⇓

Step 3: TBCppA generates CPP mapping information by analyzing the resulting tracers.

⇓

```
[CPP mapping info. (part)] Macro call N in line 3 was
expanded to 10, whose definition is in line 2.
```

Figure 4. More actual analysis steps of TBCppA

3. Definition of TBCppA tracers

Although the idea of tracers is quite simple as mentioned in Section 1, the design and implementation of TBCppA is not trivial, since the tracers must be *transparent* to CPP, i.e., the tracers must not affect the CPP behaviors. To achieve this transparency, a special form of `@"tag ..."` is introduced in Section 3.1. Fig. 4 shows an example of more actual (but still simplified) analysis steps of TBCppA, where the form `@"tag ..."` is used.

The rest of this section covers the following important issues in TBCppA implementation.

- The special form `@"tag ..."` to achieve tracer transparency, described above. (Section 3.1)
- Encoding CPP directives in tracers to preserve the information of CPP directives even after C preprocessing. (CPP eliminates all CPP directives during C preprocessing.) (Section 3.2)
- Doubling macros for conditional directives. (Section 3.3)

3.1. Basic syntax of tracers

Tracers in Fig. 2 use XML-like representation like <c>. XML formats are convenient in general, but not appropriate for the tracers in practice, since XML tags might be interpreted by CPP. For example, `c` in <c> might be expanded to other tokens by macro substitution, and < in <c> might be interpreted as the less-than operator in `#if` directives, which causes CPP error or inaccurate results. Therefore the tracers should be stable during CPP preprocessing. To achieve the stability of tracers, we define the basic syntax of TBCppA tracers as follows [4].

```
@"tagname  attrname='value' ... "
  content
@"/tagname"
```

where *value* must be XML-escaped[5], *content* must not be XML-escaped yet, and ... represents 0 or more repetitions of *attrname*='*value*'. After XML, we abbreviate a tracer without *content* as `@"tagname attrname='value' ... /"`.

The above tracer is straightforwardly translated to the following XML element after CPP preprocessing, by basically replacing `@"` and `"` with < and >, respectively, and XML-escaping *content*.

```
<tagname  attrname='value' ... >
  content
</tagname>
```

The basic syntax of TBCppA tracer is simple, but quite powerful. The first reason is that the form `@"tag ..."` never appears in syntactically correct C source code, so TBCppA tracers are not confused with C source code. The second reason is that TBCppA tracers are stable and transparent to CPP. By the C specification (6.4 and 6.10.3 in [23]), the character `@`[6] and the string literals[7] do not change during CPP preprocessing and do not affect CPP preprocessing except the following contexts: constant expressions in conditional directives like `#if`, the # and ## operators, and the macros that are expanded to function-like macro names. These exceptions will be solved or discussed in Section 3.3, Section 4.5, and Section 6, respectively.

[4] For the comprehensive and formal definition of TBCppA tracers, refer to the source code in TBCppA homepage [26].

[5] XML-escaping converts the characters <, >, &, ', " to `<`, `>`, `&`, `&apos;`, `&quot;`, respectively.

[6] The GCC-specific extension allows the programmer to use the character $ in identifier names, so we cannot use $ for this purpose.

[7] Some pre-ANSI compilers expand macro parameters even inside string literals (11.18 in C FAQ [24]), but we ignore such old compilers.

3.2. Tracers for macro definitions and macro calls

For the following function-like macro definition and macro call,

```
#define F(x)    ((x)+10)
F(20)
```

TBCppA attaches the tracers as shown in Fig. 5, consisting of the following three parts: (*Double* macros introduced in Section 3.3 are not used in Fig. 5, but they should be used in actual use.)

- Line 1-3: CPP eliminates all CPP directive lines like `#define F(x) ((x)+10)` through CPP preprocessing. But we need the information in those lines even after CPP preprocessing to generate CPP mapping information. To preserve the information, TBCppA adds tracers that have the information of CPP directive lines. For example, the tracers in the line 1-3 of Fig. 5 express the macro definition of `F` with its type (function-like), unique identifier (`F23_1`), macro name (`F`), argument lists (`x`), macro body (`((x)+10)`), etc. *locinfo* will be explained later.
- Line 4-8: To know what parts are substituted in macro calls, TBCppA surrounds the whole macro body and each macro parameter in a macro definition with the tracers like `@"macro_body"` and `@"/macro_body"`.
- Line 9-11: Similarly to the tracers for macro definitions above, TBCppA surrounds the whole macro call and each macro argument in a macro call with the tracers like `@"macro_call` and `@"/macro_call`. Note that it is impossible to statically know if a given identifier (e.g., `F`, `printf`, or `if`) is a macro call or not. So TBCppA takes the conservative approach; TBCppA always adds the tracers to all identifiers as macro calls (without maintaining the list of all macro definitions), and the redundant tracers for non-macros will be removed after CPP preprocessing.

locinfo in Fig. 5 means the location information, and has the following attributes and their values, expressing the unique identifier for the source file (e.g., `filename_ref='F23'`) and line information (e.g., `first_line='1'`)[8].

```
filename_ref='F23'
first_line='1' first_column='6'
last_line='3'  last_column='7'
```

We obtain the XML document in Fig. 6, by preprocessing the code fragment that has the tracers shown in Fig. 5 and by translating the preprocessing result into the XML format.

The first XML element with `define` tag in the line 1-3 of Fig. 6 keeps the information of the macro definition of `#define F(x) ((x)+10)`.

[8] `first_column` and `last_column` are required to accurately recover the original unpreprocessed source code from preprocessed one.

The second XML element with `macro_call` tag in the line 4-10 of Fig. 6 keeps the essence of the CPP mapping information. That is, it tells that the macro call `F(20)` was expanded to `((20)+10)` using the macro definition `#define F(x) ((x)+10)`, known by tracing the ID/IDREF link with the value `F23_1`, where the macro parameter `x` was substituted to the actual argument `20`.

The example given here is simple, but this method of tracing macro expansions also works well for more complicated macro calls (e.g., deeply nested macro calls), except the cases relating to double macros (Section 3.3) and some limitations (Section 6).

3.3. Doubling macros for conditional directives

The method of tracing macro expansions, given in Section 3.2, does not work when a tracer occurs in expressions of conditional directives like `#if` and `#ifdef`, since CPP cannot evaluate expressions including the tracer. Let's consider the following CPP directives, for example.

```
#define  VER  2
#if      VER >= 2
#else
#endif
```

By attaching the tracers to the above, we obtain the following (where all attributes are omitted for simplicity).

```
#define VER @"macro_body" 2 @"/macro_body"
#if  @"macro_call" VER @"/macro_call" >= 2
#else
#endif
```

CPP cannot evaluate the above expression `@"macro_call" VER @"/macro_call" >= 2` in `#if` directive, since it is just an illegal expression. Without the tracers, however, we cannot generate the CPP mapping information for the expression. To solve this dilemma, we introduce *macro doubling*. The key idea of macro doubling is to define the following two macros for each macro:

- *Bare macro* has the same macro definition as the original one, which is used to evaluate the expressions in conditional directives.
- *Double macro* has the same macro definition as *bare macro* except that *double macro* also has the tracers, which is required to observe the macro expansion.

Fig. 7 shows an example of macro doubling for the above code with `#if` directive, simplified for explanation purpose. `VER` is the bare macro, and `VER_1528552` is the double macro. `_1528552` is used here for the suffix of double macros, but any suffix can be used as long as it does not conflict to other names. The following explains the tracers and macro doubling in Fig. 7 in more detail.

- Line 1: Just like the `define` tag in the line 1-3 of Fig. 5, the `define` tag in the line 1 of Fig. 7 keeps in the tracers the information of the macro definition `VER` not to lose even after CPP preprocessing.

```
1  @"define type='func' id='F23_1' name='F' args='x' macro_body='((x)+10)' locinfo"
2    @"param name='x'/"
3  @"/define"
4  #define F(x) @"macro_body ref='F23_1'" @"param name='x'" x @"/param"    \
5                 ((@"macro_call type='param' name='x' locinfo"              \
6                   x                                                       \
7                   @"/macro_call" )+10)                                    \
8              @"/macro_body"
9  @"macro_call type='func' name='F' args='20' locinfo"
10   F(@"arg exp='20' locinfo"  20  @"/arg")
11 @"/macro_call"
```

Figure 5. Function-like macro definition/call, where tracers are attached, but not preprocessed.

```
1  <define type='func' id='F23_1' name='F' args='x' macro_body='((x)+10)' locinfo>
2    <param name='x'/>
3  </define>
4  <macro_call type='func' name='F' args='20' locinfo>
5    <macro_body ref='F23_1'> <param name='x'>  20  </param>
6      ((  <macro_call type='param' name='x' locinfo>
7            <arg exp='20' locinfo>  20  </arg>
8          </macro_call>   )+10)
9    </macro_body>
10 </macro_call>
```

Figure 6. The result that CPP preprocessed the code fragment with the tracers in Fig. 5

- Line 2: Bare macro definition of `VER`.
- Line 3: Double macro definition of `VER`.
- Line 4-23: The `cond` tag encloses the whole of a block of conditional directives like `#if` ... `#else` ... `#endif`, to identify the conditional directives.
- Line 5-14: Managing the `#if` part:
  - Iff `VER >= 2` in the line 5 is evaluated to true by CPP, `cond_selected` element in the line 6-14 will remain and will be traced through CPP preprocessing.
  - Line 7-12: In the `exp` element, the macro call `VER` is doubled: bare macro call (line 9) and double macro call (line 10). As will be mentioned in Section 6, there is a case where double macro calls cannot be correctly traced, so bare macro calls are also placed.
- Line 15-18: The `#else` part is managed in the same way as the `#if` part in the line 5-14.
- Line 20-22: The `if`, `else`, and `endif` elements keep the information of `#if`, `#else`, and `#endif` directives, respectively, not to lose through CPP preprocessing. After CPP preprocessing and converting the result to the XML format, we obtain the XML document in Fig. 8. As shown in Fig. 8, only one `<cond_selected>` element remains and it corresponds to the `<if>` element in the line 12, which is known by following the XML ID/IDREF link with the value `F1_3`.

The XML document in Fig. 8 also tells the following:

- In the line 5 of Fig. 7 (`#if VER >= 2`), using bare macro, not double macro, allows CPP to preprocess the conditional directives without causing CPP error.
- The `double` element in the line 7 of Fig. 8 tells the macro `VER` in `#if VER >= 2` is expanded to `2` using the macro definition `#define VER 2`. This shows double macros are effective to trace macro expansions.

Thus, the macro doubling method for tracing conditional directives works fine. Unfortunately, the method cannot trace the conditionally excluded code (e.g., the code between `#else` and `#endif` in Fig. 7), since `TBCppA` is based on the analysis of the result of CPP preprocessing. Analyzing conditionally excluded code is extremely difficult and thus out of the scope of this paper.

4. Design and implementation of TBCppA

4.1. Design policy

The design policy of `TBCppA` is roughly as follows.

- **No CPP modification**: We never modify the existing CPP implementation. As mentioned in Section 1, the major advantages of `TBCppA` are high portability and high maintainability. If we modify the CPP, the advantages will be simply lost.
- **Emphasis on practicability**: `TBCppA` should process as many real-world C programs as possible, run as fast as possible, using as little memory as possible. To achieve the requirements,
  - `TBCppA` supports as many features of C99 [23] and GCC-specific extensions as possible.
  - `TBCppA` does not use the libraries that consume much memory (e.g., XML's DOM).
- **Low development/maintenance cost**: We keep `TBCppA` implementation simple and compact.

4.2. TBCppA components and process flow

Fig. 3 illustrates the basic components and the process flow of `TBCppA`.

- `tadd.exe` embeds `TBCppA` tracers to unpreprocessed C source code, and then native CPP (e.g., `gcc -E`) preprocesses the code. `txml.exe` converts the preprocessed code (including the embedded tracers) into XML format.

```
1  @"define type='obj' id='F1_1' name='VER' macro_body='2' locinfo/"
2  #define VER         2                                        /* bare macro definition */
3  #define VER_1528552 @"macro_body ref='F1_1'" 2 @"/macro_body"  /* double macro definition */
4  @"cond id='F1_2'"
5  #if     VER >= 2  /* bare macro is used here not to cause CPP error  */
6    @"cond_selected type='if' ref='F1_3' locinfo"
7      @"exp orig='VER &gt;= 2'" /* orig keeps original expression */
8        @"macro_call type='obj' name='VER' locinfo"
9          @"bare"   VER         @"/bare"    /* bare macro call */
10         @"double" VER_1528552 @"/double"  /* double macro call */
11       @"/macro_call" >= 2
12     @"/exp"
13     /* here comes the code exclusively between #if and #else with the tracers attached */
14   @"/cond_selected"
15 #else
16   @"cond_selected type='else' ref='F1_4' locinfo"
17     /* here comes the code exclusively between #else and #endif with the tracers attached */
18   @"/cond_selected"
19 #endif
20   @"if    id='F1_3' dir='#if'    exp='VER &gt;= 2' locinfo/"
21   @"else  id='F1_4' dir='#else'                    locinfo/"
22   @"endif id='F1_5' dir='#endif'                   locinfo/"
23 @"/cond"
```

Figure 7. Example of Macro Doubling (*before* CPP preprocessing)

```
1  <define type="obj" id="F1_1" name="VER" macro_body="2" locinfo/>
2  <cond id="F1_2">
3    <cond_selected type="if" ref="F1_3" locinfo>
4      <exp orig="VER &gt;= 2">
5        <macro_call type="obj" name="VER" locinfo>
6          <bare>2</bare>
7          <double><macro_body ref="F1_1">2</macro_body></double>
8        </macro_call> &gt;= 2
9      </exp>
10     /* here comes the code between #else and #endif with XML tags converted
11        from the tracers */
12   </cond_selected>
13   <if    id="F1_3" dir="#if"    exp="VER &gt;= 2" locinfo/>
14   <else  id="F1_4" dir="#else"                    locinfo/>
15   <endif id="F1_5" dir="#endif"                   locinfo/>
16 </cond>
```

Figure 8. Example of Macro Doubling (*after* CPP preprocessing)

- Finally `tfix2.exe` slightly adjusts the resulting XML data, eliminating duplicated ID attributes, eliminating tracers for non-macro calls, and replacing doubled macros with the corresponding bare macros. The output `foo.c.xml` becomes valid for `TBCppA`'s DTD (`TBCppA.dtd`) [26]. Against our intuition, `tfix2.exe` takes the most time in TBCppA components (about 50% time for `bash-3.1`).
- All lexical tokens including whitespaces and comments in unpreprocessed C source code are recorded with their location information in another file `foo.c.lex`, not in `foo.c.xml`. This is done on each file basis. For example, all tokens in `stdio.h` are recorded in `stdio.h.lex`, not `foo.c.lex`, since this information is not affected by the preprocessing environment. This considerably achieves file size reduction and makes `TBCppA` implementation simple.
- Each filename and pathname[9] are recorded to the file `db_file.xml` to assign a short unique identifier to each file. This just follows the compression mechanism of DWARF2 debugging information to reduce the resulting XML file size.
- `foo.c.xml` includes `foo.c.lex` and `db_file.xml` using XML's external entity reference, which corresponds to CPP's `#include`.

4.3. TBCppA Parser in tadd.exe

First, `tadd.exe` parses unpreprocessed C source code before embedding the tracers. The parser in `tadd.exe` is the most difficult part in `TBCppA` implementation. This section explains this difficulty.

The parser in `tadd.exe` is not a C parser, nor a CPP parser, since:

- The parser in `tadd.exe` parses CPP directives and (potential) macro calls, ignoring the C keywords and C syntax.
- The parser in `tadd.exe` does not evaluate CPP directives and does not expand any macros in parsing, while CPP parsers do.

As an example of the latter case, consider the code fragments in Fig. 9(a) and (b). The parentheses do not match in the code fragments, but CPP processes them with no problems, since, in Fig. 9(b) for example, CPP first evaluates `#ifdef`, obtains `FOO(10,30)` if `WIN32` is true, and finally parses `FOO(10,30)` where the parentheses match.

Unlike CPP, `tadd.exe` needs to directly parse the code fragments where CPP directives remain, which makes the parsing of `tadd.exe` much difficult.

Context dependency is another reason of the difficulty. As an example of context dependency in unpreprocessed C source code, parentheses have to match in the context of actual arguments in macro calls (e.g., parentheses have to match in the actual argument `(2,3)` in `FOO(1,(2,3))`), while not in the other contexts. As mentioned in Bison [10]'s manual, the context dependency is very likely to violate the LALR(1) paradigm, and some "kludge" techniques are often required.

After many trial-and-errors, we practically and successfully implemented the parser in `TBCppA` by dealing with

[9] `TBCppA` uses `realpath` to normalize the pathname.

```
#define FOO(x) (x)
#define PAREN (
PAREN )
FOO(10)
```

(a) `tadd.exe` can parse this.

```
#ifdef WIN32
  FOO (10,
#else
  FOO (20,
#endif
  30)
```

(b) `tadd.exe` cannot parse this, but continue the parsing by error recovery.

Figure 9. Examples that mismatching parentheses make parsing difficult.

the four contexts in the grammar rules, whose detailes are omitted due to the lack of space[10].

As a result of taking this approach, the parser in `tadd.exe` copes with both of two different structures: the line-oriented structure in CPP directives and the parenthesis-oriented structure in macro calls.

Generally speaking, `tadd.exe` cannot embed the tracers when a CPP directive occurs in the middle of macro arguments like Fig. 9(b). In this sense, `tadd.exe` is imperfect, but this imperfectness very little affects the accuracy of `TBCppA`. For example, there are about 140,000 macro calls, including the macro calls in included header files, in gzip-1.2.4 (consisting of around 7300 lines) and bash-3.1 (around 70,000 lines), but `tadd.exe` only reported 10 parse errors in the preliminary experiment (Section 5.2).

4.4. TBCppA code size and development time

The code size of `TBCppA`(ver 0.0.3b) is small; it consists of only 3,700 lines, using several programming languages including Bison, Lex, C, csh, XSLT and Graphviz's dot. The development time was two months by one programmer. In `TBCppA` components, `tadd.exe` (Fig. 3) is the largest one; it consists of 3,000 lines including `tadd.y` with 101 grammar rules and 1,300 lines of code.

4.5. The devil is in the details

The idea of `TBCppA` is novel, but quite simple, as `TBCppA` just embeds the tracers, runs native CPP as-is, and generates CPP mapping information by analyzing the resulting tracers. But, as is often the case in software engineering, there are a lot of small issues that we had to solve in implementation. Here we enumerate a part of them, but we do not describe their details due to the lack of space.

- Error handling in `TBCppA` parser.
- Eliminating duplicated ID attributes.
- Coping with # and ## operators.
- Identifying macro calls.
- Avoiding erroneous macro redefinition.

There are other issues not listed here. See Section 6 for unsolved issues.

[10] Due to the context dependency, it is very difficult to implement the `TBCppA` parser simply using the string pattern-matching given in some script languages.

5. Preliminary evaluation

This section provides a preliminary evaluation of `TBCppA`'s performance (its execution speed and the file sizes of the resulting CPP mapping information) to demonstrate that `TBCppA` is highly practical in the sense that `TBCppA` processes even large open source software like `GCC` in a reasonable speed. This section also provides three applications using `TBCppA`'s CPP mapping information to demonstrate the output of `TBCppA` is useful enough.

All the performance given in this section is measured in the Notebook PC platform (Intel Pentium M 1.2GHz, 1GB RAM, Windows XP SP2, Cygwin 1.5.19, GCC 3.4.4, Bison 2.1, Flex 2.5.4, Libxml 2.6.22, Graphviz 2.8).

5.1. Execution time of tadd.exe's embedding the tracers

The execution speed of `tadd.exe`'s embedding the tracers is very fast. To show this, we measured the execution time of `tadd.exe`'s embedding the tracers in 480 system header files including `/usr/include/*.h`. The result shown in Table 1 is very good; `tadd.exe` processed the total of 140 KLOC [11] (including empty lines and comments) only in 89 seconds.

As far as we know, the result of embedding the tracers is correct except three header files, all of which use the GCC-specific variadic macros. This is simply because the current version of `TBCppA` cannot cope with the GCC-specific variadic macros.

5.2. Execution time of TBCppA's generating CPP mapping information

The execution speed of `TBCppA`'s generating the CPP mapping information is reasonably fast. To show this, we measured the execution time of `TBCppA` for several programs including gzip-1.2.4, bash-3.1 and gcc-4.1.4 . The result is shown in Table 2.

Before `TBCppA` processes the file `foo.c`, `TBCppA` needs to embed the tracers in all the header files `#included` in `foo.c`. So, in Table 2, the processing time of embedding the tracers in the header files (except the system header files) and the processing time of generating the CPP mapping information for `foo.c` are separately measured. The execution time for gcc-4.1.1 (consisting of around 630 KLOC) is about one hour. This result demonstrates that `TBCppA` runs fast enough for non-interactive applications. `TBCppA` can be acceptable even for interactive applications, since the CPP mapping information can be updated incrementally for each file (i.e., translation unit).

Table 2 also shows the file size of the CPP mapping information, which is 13 to 25 times as large as the file size of the preprocessed source code. XML technology often increases the file size by 10 to 100 times, so this result is natural.

[11] KLOC: 1,000 lines of code.

Table 1. Execution time of `TBCppA`'s embedding tracers in system header files and the resulting file size

# of files	original size of `*.h`	execution time	size of `*.h` + tracers
480	140 KLOC (5.0MB)	89 sec	73.8MB

Table 2. Execution time of `TBCppA` generating CPP mapping information and the generated file size

	size of `*.h`	size of `*.c` before CPP	size of `*.c` after CPP ¶	processing time of `*.h`	processing time of `*.c`	size of `*.c` ¶ + tracers	generated file size `*.c.{xml,lex}`
`hello.c`†	—	5 lines (76B)	76B	—	0.45 sec	2.8KB	0.8KB+4.4KB
`hello2.c`†	—	5 lines (63B)	24KB	—	1.4 sec	1.7KB	520KB+3KB
gzip-1.2.4	7.4KB	7.3 KLOC (230KB)	1.5MB	0.6 sec	32 sec	3.1MB	11.9MB+7.0MB
bash-3.1‡	420KB	70 KLOC (1.8MB)	10MB	13 sec	485 sec	33MB	191MB+61MB
gcc-4.1.1§	3.7MB	630 KLOC (18MB)	102MB	72 sec	3,602 sec	332MB	1.2GB+554MB

† `hello.c` does not `#include <stdio.h>`, while `hello2.c` `#includes <stdio.h>`.
‡ The files under `lib` directory in bash-3.1 are not analyzed here.
§ The files of `gcc/{,cp/}*.c` are analyzed here after `configured` with the options `--enable-threads=win32 --with-cpu=i686 --with-arch=i686 --with-tune=i686 --enable-languages=c,c++`.
¶"size of `*.c` after CPP" refers to the size of preprocessed source code in normal compilation (i.e., tracers are not included), while "size of `*.c` + tracers" refers to the size of `TBCppA/foo.c` in Fig. 3.

5.3. Applying CPP mapping information to tools

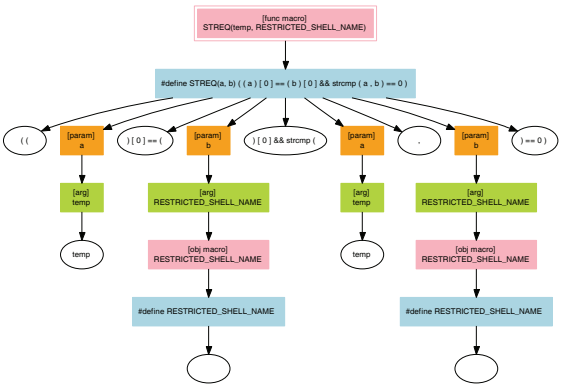

Figure 10. `tmacro` visualizing macro expansion of `STREQ` in `shell.c` (line 1104 of bash-3.1)

To show how the CPP mapping information generated by `TBCppA` (e.g., in Section 5.2) is practical and useful, we experimentally developed the following small but practical applications that utilize the `TBCppA`'s output:

- **tmacro** visualizes the macro expansion. A screen snapshot is shown in Fig. 10.
- **tifdef_src2html** grays the conditionally excluded code as shown in Fig. 11.
- **trecov_demo** recovers the original file from the CPP mapping information generated by `TBCppA` and the preprocessed source code, not using the token level information in `*.c.lex` (Section 4.2) for recovering the macro calls. In Fig. 12, the same file as the original `shell.c` of bash-3.1 is identically recovered from `shell.c.xml`, `shell.c.lex`, and the preprocessed `shell.c`.

The above tools are all small; 65 lines in csh and 136 lines in XSLT for **tmacro**, 22 lines in XSLT and 160 lines in C for **tifdef_src2html**, and 755 lines in C for **trecov_demo**.

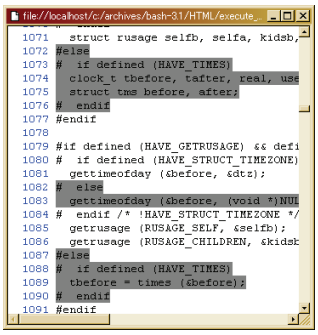

Figure 11. `tifdef_src2html` graying the conditionally excluded code in `execute_cmd.c` of bash-3.1.

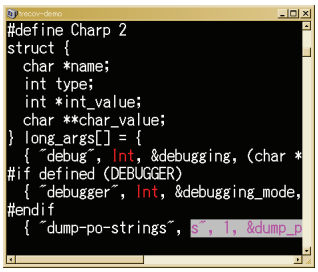

Figure 12. `trecov_demo` recovering the original file (`shell.c` of bash-3.1) using CPP mapping info.

The development time for them was short; each of them only required 0.5 to 4 man-days.

6. Limitations

Although `TBCppA` has many positive characteristics, `TBCppA` has several limitations. Here we enumerate the major limitations[12]:

[12] Many other CPP analyzers do not state their limitations, but the limitations should be more open to the public.

- The current TBCppA does not support the GCC-specific variadic macros.
- The current TBCppA provides the wrong information about the macro parameters (__VA_ARGS__) of C99 variadic macros in the <double> element.
- In the macro calls like F1(a) in the following, TBCppA reports the wrong process of macro expansion in the <double> element, although TBCppA reports the correct result of macro expansion in the <bare> element.

```
#define F1    F2
#define F2(x) (x)
F1 (a)
```

- It can be time-consuming for the user to collect all the CPP command options (e.g., -I and -D in GCC) given in the original build, which is required to use TBCppA.
- TBCppA does not analyze conditionally excluded code.
- TBCppA's CPP mapping information does not include the macro definitions given by the CPP command line (e.g., -DNDEBUG in GCC) and the macro definitions that CPP pre-defined in an implementation-specific manner (e.g., unix, i386 mentioned in Section 2.3).

7. Related work

To our knowledge, this is the first paper that proposes a method using tracers to generate CPP mapping information.

Previous works [1–4, 8, 9, 11–13, 15, 17–21, 27, 28] implement CPP emulators or modify real CPPs to generate the CPP mapping information, which all causes problems like inaccuracy or low portability as mentioned in Section 2.

Garrido proposed CRefactory to analyze multiple configurations introduced by #ifdef directives [8, 9]. The idea of CRefactory is very attractive because it analyzes all possible configurations simultaneously by completing (or normalizing) preprocessor conditionals and directly parsing unpreprocessed source code. As far as we know, CRefactory has not been released yet to the public, which might imply the difficulty of implementing CRefactory that can handle real-world C programs.

[14] proposed a new AST-based macro language ASTEC that causes less CPP problems, but the ASTEC approach requires to rewrite all the existing code to ASTEC.

The method using DWARF2 debugging information [25] simply lacks the information about the macro expansions.

8. Conclusion

This paper proposed a novel approach based on *tracer* to generate CPP mapping information, which is indispensable to accurately analyze C source code. To demonstrate our approach, we developed TBCppA, and applied it to gcc-4.1.1. We also developed three small tools using TBCppA's CPP mapping information. The preliminary result suggests that our tracer approach works fine for large software products like gcc-4.1.1.

As future work, we need to relax the limitations given in Section 6, although we believe the limitations very little affect the accuracy and usability of TBCppA. We also need to provide more development tools using TBCppA's CPP mapping information.

References

[1] G. J. Badros: PCp^3: A C Front End for Preprocessor Analysis and Transformation, Masters Thesis, 1997.

[2] G. J. Badros and D. Notkin: A framework for preprocessor-aware C source code analyses, Softw. Pract. & Exper., vol.30, Issue 8, pp. 907–924, 2000.

[3] M. L. Collard, J. I. Maletic, and A. Marcus: Supporting Document and Data Views of Source Code, Proc. ACM Sympo. on Document Engineering (DocEng'02), pp.34–41, 2002.

[4] A. Cox and C. Clarke: Relocating XML Elements from Preprocessed to Unprocessed Code, 10th Int. Workshop on Program Comprehension (IWPC'02), p.229, 2002.

[5] M. D. Ernst, G. J. Badros and D. Notkin: An Empirical Analysis of C Preprocessor Use, IEEE Trans. on Software Engineering, vol.28, no.12, pp.1146–1170, 2002.

[6] J. M. Frave: The CPP paradox, 9th European Workshop on Software Maintenance, DURHAM'95, 1995.

[7] A. Garrido and R. Johnson: Challenges of refactoring C programs, Proc. Int. Workshop on Principles of Software Evolution (IWPSE), pp.6–14, 2002.

[8] A. Garrido and R. Johnson: Refactoring C with Conditional Compilation, 18th IEEE Int. Conf. on Automated Software Engineering (ASE 2003). pp.323–326, 2003.

[9] A. Garrido and R. Johnson: Analyzing Multiple Configurations of a C Program, Proc. IEEE Int. Conf. on Software Maintenance (ICSM'05), pp.379–388, 2005.

[10] GNU Project: Bison, Free Software Foundation, http://www.gnu.org/.

[11] P. E. Livadas and D. T. Small: Understanding Code Containing Preprocessor Constructs, Proc. IEEE Third Workshop Program Comprehension, pp.89–97, 1994.

[12] E. Mamas and K. Kontogiannis: Towards Portable Source Code Representations Using XML, Proc. IEEE Working Conf. on Reverse Engineering (WCRE'00), pp.172–182, 2000.

[13] cppML,http://swen.uwaterloo.ca/~evan/cppml.htm

[14] B. McCloskey and E. Brewer: ASTEC: a new approach to refactoring C, Proc. 10th European Software Engineering Conf, pp.21–30, 2005.

[15] C. A. Mennie and C. L. A. Clarke: Giving Meaning to Macros, 12th IEEE Int. Workshop on Program Comprehension, p.79, 2004.

[16] G. C. Murphy, D. Notkin, and E. S. C. Lan: An Empirical Study of Static Call Graph Extractors, 18th Int. Conf. on Software Engineering (ICSE), pp.90-99, 1996.

[17] L. Vidacs and A. Beszedes: Opening Up The C/C++ Preprocessor Black Box: Proc. 8th Sympo. on Programming Languages and Software Tools (SPLST'03), pp.45–57, 2003.

[18] L. Vidacs, A. Beszedes and R. Ferenc: Columbus Schema for C/C++ Preprocessing, 8th European Conf. on Software Maintenance and Reengineering, pp.75–84, 2004.

[19] M. Vittek: Refactoring browser with preprocessor, 7th European Conf. on Software Maintenance and Reengineering, 2003.

[20] D. Spinellis: Global analysis and transformations in preprocessed languages, IEEE Trans. on Softw. Eng., 29(11):1019–1030, 2003.

[21] D. Spinellis: Browsing and refactoring program collections written in C, WP 2004-006, Eltrun Working Paper Series, 2004.

[22] Programming languages–C: ISO/IEC 9899:1990.

[23] Programming languages–C: ISO/IEC 9899:1999.

[24] comp.lang.c Frequently Asked Questions, http://c-faq.com/

[25] K. Gondow, T. Suzuki, H. Kawashima: Binary-Level Lightweight Data Integration to Develop Program Understanding Tools for Embedded Software in C, 11th Asia-Pacific Software Engineering Conference (APSEC), pp.336-345, 2004.

[26] K. Gondow: TBCppA Homepage, http://www.sde.cs.titech.ac.jp/~gondow/TBCppA/.

[27] Y. Momiyama and S. Yamamoto: Sapid P-model Specification ver. 1.0, http://www.sapid.org/html2/P-model/P-model.pdf, 2000. (in Japanese)

[28] T. Mimura: Relating source program and C preprocessor's output for program understanding, Bachelor's Thesis in Nagoya Univ., 1993. (in Japanese)