Vol.2017-SE-195 No.29
2017/3/13

マクロ展開処理の追跡を行うC言語前処理系解析器

福原 和哉[1]　猪股 俊光[2]　杉野 栄二[2]　新井 義和[2]　今井 信太郎[2]

概要：C 言語を対象としたプログラム解析ツールの多くは前処理後のプログラムを解析の対象としているが，Erust らの調査では前処理指令を用いない C 言語プログラムは存在せず，前処理指令は全ステップ数の 8.4%を占めていることから，C 言語プログラムの正確な解析には前処理によって置換・挿入される要素の対応付けが必要である．そこで，筆者らは前処理前後での要素の対応関係と前処理過程をモデル化して出力するC言語前処理系解析器を提案し，作成・評価した．その結果，得られたモデルを用いることで，マクロ展開前後のプログラムの要素の対応付けと，前処理過程の可視化が可能であることがわかった．

A C preprocessor analyzer that traces macro expansion processing.

Kazuya FUKUHARA[1]　Toshimitsu INOMATA[2]　Sugino Eiji[2]　Yoshikazu ARAI[2]
Shintaro IMAI[2]

1. はじめに

ソフトウェアに対する品質・コストの要求が高まっており [1]，それに対応すべく，品質の確保や工数削減，保守性の向上を目的としたプログラム解析ツール [2], [3], [4], [5], [6] の開発も多く行われている．

しかし，既存の C 言語を対象としたプログラム解析ツールは前処理済み C プログラムを入力として要求しており，前処理済みプログラムの取得には解析ツールとは無関係の既存の前処理系の出力を用いている．そのため，前処理によって失われるマクロや前処理指令などの情報を得ることができず，前処理前後でのプログラムの対応関係を取得することができない．

Erust らの調査 [7] によると前処理指令を用いない C 言語プログラムは存在せず，また，前処理指令は全ステップ数の 8.4%を占めていることから，C 言語プログラムの正確な解析には前処理命令も含めた解析が必要である．

そこで，筆者らは前処理結果の出力および要素の対応関係を含む，前処理過程をモデル化した形式 CPPXML と CPPXML 形式の出力に対応したC言語前処理系解析器を提案し，作成・評価した．

その結果，得られたモデルを用いることで，前処理前後のプログラムの要素の対応付けと，前処理過程の可視化が可能であることがわかった．

2. 前処理解析の課題

2.1 C 言語前処理の問題点

C プログラム中で前処理命令の出現位置は C 言語の文法とは独立しており，記述の自由度が非常に高く，マクロによる予約や要素の置換・結合が可能，条件付きコンパイル命令を用いてプログラムの一部の書き換えが可能であるが，図 1 の引数の選択が問題となる例や括弧の対応付けが問題となる例が示すように，前処理を無視すると引数の数や，括弧の対応関係などが定まらないプログラムも記述可能なため，前処理前のプログラムの構文解析は非常に難しく，正確な解析が困難である．

一方，前処理系の行う動作を模倣する場合は次に挙げる点が問題となる．

(2.1-a)マクロ展開の仕様には明文化されていない事項が存在し，処理系が独自に対応を行っている

(2.1-b)既存の処理系で施されている独自拡張の存在

1　岩手県立大学いわてものづくりソフトウェア融合テクノロジーセンター
Iwate Monodukuri and Software Integration Technology Center
2　岩手県立大学大学院ソフトウェア情報学研究科
Graduate School of Software and Information Science, Iwate Prefectural University

ⓒ 2017 Information Processing Society of Japan

```
// 引数の選択が問題となる
int write_to(
#ifdef _HAS_STDIO
  FILE fp,
#else
  int fd,
#endif
  unsigned char *data,  size_t size) {
     （略）
    // 括弧の対応付けが問題となる例
#ifdef _HAS_STDIO
    if (fwrite(data, 1, size, fp) != size) {
#else
    if (write(fd, data, size) != size) {
#endif
     （略）
   }
}
```

図 1 前処理前のコードの構文解析が難しい例
Fig. 1 An example where it is difficult to pase code before preprocessing.

```
/* 元コード */
#define STR(s) STR_(s)
#define STR_(s) #s
#define PASS(x) x
PASS(a)b;
STR(PASS(a)b);

/* cpp(gcc version 4.8.5) */
a b;
"ab";

/* MCPP V.2.7.2 */
a  b ;
"ab";

/* Microsoft(R) C/C++ Optimizing Compiler
 * Version 19.00.24215.1
 */
ab; // 他の処理系と意味的に異なる結果
"ab";
```

図 2 マクロ展開で意味的に異なる結果が得られる例
Fig. 2 An example in which semantically different results are obtained by macro expansion.

(2.1-c)処理系やシステムが提供するヘッダファイルで用いられているマクロ

(2.1-a) は，前処理系の模倣を行う際の大きな問題であり，既知の仕様に従って処理系を実装しても，これが原因で既存の処理系と意味的に異なる展開結果となることがある．図 2 に GCC [8], mcpp [9]，VisualC で意味的に異なる結果が得られる例を示す．

(2.1-b) には次に示すものがある．

- GCC に存在する`#include_next` マクロ
- `clang` に存在する`#import` マクロ
- GCC でサポートされている引数 0 個をサポートする可変長引数マクロの書式
- 非標準の`#pragma` 指令

これらは，処理系に依存した動作を行うものであり，標準の前処理命令ではない．そのため，模倣を行うためには対象処理系の仕様を調査する必要があり，多くのコストが必要となる．

(2.1-c) には，処理系内部で定義されるマクロ (`_msc_var` や`__i686` など) がある．gcc[8] であれば `gcc -E -dM -xc /dev/null` などで調査することが可能ではあるが，設定される内容や条件については対象処理系の仕様を調査する必要があり，多くのコストが必要となる．また，処理系から提供されるシステムヘッダファイル内では，独自マクロや独自命令が多用されており，システムヘッダファイルによって提供されるマクロ (`stdin` や `va_arg` 等) の内容も処理系依存である．そのため，システムヘッダファイルを用いるためには，独自マクロや独自命令への対応が必要となることも動作模倣の課題である。

2.2 既存手法

前処理過程の追跡を行っている先行研究 [10], [11], [12], [13], [9] では，次に示す手法を用いて対応関係などを得ている．

(2.2-a)前処理前のプログラムを受理できるようにツールを拡張 [10], [11]

(2.2-b)既存の前処理系のオプションで得られる出力や，コンパイル時に得られるデバッグ情報から情報を得る [9]

(2.2-c)既存の前処理系を改造して必要な情報を出力させる [12]

(2.2-d)前処理前のプログラムに追跡子を埋め込み，前処理後の結果と対応づける [13]

(2.2-a) は GNU GLOBAL [14] や Doxygen [15] などの厳密なコード解析が不要なツールで採用されているが，そのため，前処理命令を挿入できる位置に強い制約を設けることで，構文解析を可能にしているものもあるが，適用可能なソースコードの範囲が狭く限定されたる欠点がある．

(2.2-b) では前処理結果に冗長な情報を含ませるオプションの有効化 (gcc では`-M`, `-dM`, `-dI` など) や，デバッグ情報を有効化してコンパイルを行うなどして，各処理の結果に含まれる情報を収集して解析を行うものである．(2-a) と違い前処理系を用いるため，正確な前処理結果を容易に得ることが可能だが，マクロ展開についての情報が得られず，前処理前後のプログラムの要素間の対応付けが困難である．

(2.2-c) は既存の前処理系に対して拡張・改造を行い，マクロ展開等の情報を収集するものであり，gcc の前処理機能をライブラリ化した Cpplib[16] や clang::Preprocessor[17] を用いる研究 [12] などがある．この方式は (2.2-b) と同様に正確な前処理結果を容易に得ることが可能であるが，改造元となった前処理系への動作環境や機能の依存度が強く，機能拡張，移植，前処理系のバージョンアップなどに必要なコストも大きいものとなる．実際に PCp3[12] は cpplib2

系が用いられており，現在の gcc が提供する cpplib のバージョンとはかけ離れているが，更新などがされていないため，現状のままでの利用は困難である．

(2.2-d) は前処理前の C プログラム中で前処理の対象となるトークン列に対し追跡子と呼ばれるトークンを付加した後，既存の前処理系を用いて前処理を行い，前処理前後のプログラム中に含まれる追跡子から展開結果の対応関係を取得するものである．他の方式と比べて，既存の前処理系をそのまま利用できるため移植性の高さや保守の低コストが利点であるが，行を跨ぐマクロや，可変長マクロの引数の扱いに問題があること，追跡子を埋め込むことにより一部のマクロ形式の解析が不可能になることなど利用には制限がある．また，既存の前処理系が対応していない非標準の`#pragma` 指令などは処理を行うことができず，実際のビルド時に用いる処理系とは違う処理系を用いるケースには適用ができない．

このように既存の研究が用いる手法は一長一短である．

3. 提案する前処理追跡手法

3.1 前処理に要求する機能

前処理系に要求する機能は次の通りである．

(3-a)前処理前後での対応関係の取得
(3-b)前処理での展開過程の取得
(3-c)未対応の`#pragma` 指令の保持
(3-d)特定の処理系に依存しない
(3-e)前処理で得られた展開過程と結果の出力をコード解析ツールへの入力に利用

(3-a) は多くの既存手法で取得可能であることが示されている．しかし，(3-b) については，既存手法 (2.2-d) 以外では示されていない．

また，(3-d) と (3-c) は，既存の前処理系を用いる既存手法 (2.2-b, c, d) では満たすことができない．

また，いずれの方式も出力はプリプロセス後の C ソースコードと対応付け情報となっており，(3-e) には前処理後のソースコードと対応付け情報を読み取り，C プログラム解析時のトークン読み取り時に再度の対応付けが必要となるため，出力結果を用いたツールを実装する際にコストが発生する．

これらの要求を満たすモデル CPPXML の提案と出力用の C 言語前処理系解析器 CSCPP の実装を行い，評価した．

3.2 前処理の展開過程の表現

かつて，K&R や C89 では C 言語の前処理は手法が定義されておらず，ISO/IEC C90 で初めて “Phases of Translation” として定義された．また，複雑なマクロ展開の部分については アルゴリズムが Dave Prosser's C Preprocessing Algorithm [18] として公開されている．これに従った前処理の流れを次に示す．

(3.1-a)前処理前ソースコード中のトリグラフの置換を行う．
(3.1-b)前処理前ソースコード中のバックスラッシュに改行文字が続く並びを削除し，行を連結する．
(3.1-c)前処理前ソースコードをトークン列に変換する．コメントは 1 文字の空白トークンに置換する．改行は保持する．
(3.1-d)トークン列からトークンを順次読み取り，前処理指令の処理と Dave Prosser's C Preprocessing Algorithm [18] に従ったマクロ展開を行い，前処理後のトークン列を得る．
(3.1-e)前処理後のトークン列から前処理後のプログラムを出力する．

(3.1-c), (3.1-d) より，前処理系は入出力を文字列ではなくトークンで扱うことが必要となる．これに習い，CPPXML では展開される要素をトークン単位で扱うこととし，ソースコード上，違う位置に登場する同一の字面の要素を区別するためにトークン毎にユニークな ID を割り当てることとした．

また，前処理系によってトークンは表 1 に示す種別情報のうち TokenRef を除くいずれかに分類されるため，モデル上のトークン情報にはこれらも含まれる．

表 1 トークンの種別と意味
Table 1 Token type and meaning

トークン種別	意味
Ident	識別子もしくは予約語を示す
Keyword	演算子もしくは記号を示す
Number	整数値定数を示す
String	文字列定数を示す
Space	空白文字，コメント，条件付きコンパイルなどで読み飛ばされたコードを示す
NewLine	改行を示す
TokenRef	既存のトークンへの参照を示す

表 1 に示す種別情報のうち，TokenRef は通常の前処理系では必要のない種別である．図 3 の例のように，複数箇所で定数マクロが利用されるコードにおいて，前処理前のトークンを複製して前処理後のトークン列に挿入してしまうと出力トークン列上では同一の ID を持つトークンが複数登場することになり，展開の対応関係を用いた逆変換の際，置換対象となる前処理後トークン部分列が一意に定まらない問題が発生する．この対策として，CPPXML 形式においてトークンの複製は，複製対象のトークンへの参照として TokenRef を用いて表現する．TokenRef 型のトークンは参照先トークンの ID とは別に自身を示すユニークな ID を持つため，前処理後トークン列において同一の ID を持つトークンが複数登場する問題が起きない．

```
// 展開前コードで出現するトークン num は 1 つ
#define NUM num
int n1 = NUM;
int n2 = NUM;

// 展開後コードにはトークン num が複製される
(空行)
int n1 = num;
int n2 = num;
```

図 3 マクロ展開時にトークンのコピーが発生する例
Fig. 3 An example in which a copy of a token occurs at macro expansion.

3.3 前処理命令とマクロ処理

(3.1-d) で行われる前処理命令とマクロ展開の処理の大まかな流れを次に示す.

(3.1-1)出力トークン列を空列とする.
(3.1-2)読み戻しトークンスタックを空とする.
(3.1-3)トークンを一つ取得する．取得するトークンは次の方法で選択される
- (3.1-3-i)読み戻しトークンスタックが空で無い場合，読み戻しトークンスタックの最上位要素を取り出す.
- (3.1-3-ii)読み戻しトークンスタックが空の場合，入力ソースコードからトークンを一つ読み取る.
- (3.1-3-iii)入力ソースコードから読み取れなければマクロ展開終了.

(3.1-4)取得したトークンが前処理命令，もしくは展開可能なマクロであるか調べる
- (3.1-4-i)合致する場合，取得したトークンに加え展開に必要な数のトークンを追加で読み取り，展開前トークン列を作成して前処理を適用．その結果得られた展開後トークン列を読み戻しトークンスタックに積む.
- (3.1-4-ii)合致しない場合，読み取ったトークンを出力トークン列に追加する

(3.1-5)(3.1-3) に戻る

このことより，前処理系が行うマクロ展開などの動作は次のように表現できる.

- 前処理とは入力ソースコードから得られた入力トークン列 S_{input} に対して，有限個の書き換え操作 Log を適用して出力トークン列 S_{output} を得る.
- 有限個の展開操作は入力トークン列の並びから一意に定まる.
- 書き換え操作 r とは，入力トークン列中のあるパターンに一致する部分列（展開前トークン列 S_{src}）を，部分列に書き換え操作を適用して得られたトークン列（展開後トークン列 S_{dst}）で置き換える.

展開処理の追跡で必要な情報は，この展開前トークン列 S_{src} と展開後トークン列 S_{dst}，そして，対応関係を示す前処理の種別を組とした情報である.

これらより筆者らが提案する展開処理を記録した前処理追跡モデル $Model$ の定義を次に示す.

$Model = \{T_{\mathbf{all}}, T_{\mathbf{input}}, T_{\mathbf{output}}, Log\}$
  前処理追跡モデル

$t = (TokenKind, String, Id)$
  トークンを構成する情報の組.

$T_{\mathbf{all}}$
  前処理中に登場する全てのトークン t の集合.

$S_{\mathbf{input}} = (t_1, t_2, \cdots, t_n | t_i \in T_{\mathbf{all}}, 1 \leq i \leq n)$
  前処理前のソースコードから得られたトークン列.

$S_{\mathbf{output}} = (t_1, t_2, \cdots, t_m | t_i \in T_{\mathbf{all}}, 1 \leq i \leq m)$
  前処理後のソースコードと対応するトークン列.

$r = (ReplaceKind, S_{\mathbf{src}}, S_{\mathbf{dst}})$
  トークン列中の部分列 S_{src} から 部分列 S_{dst} への書き換え操作の記録.

$S_{\mathbf{src}} = (t_1, t_2, \cdots, t_o | t_i \in T_{\mathbf{all}}, 1 \leq i \leq o)$
  展開前トークン列

$S_{\mathbf{dst}} = (t_1, t_2, \cdots, t_p | t_i \in T_{\mathbf{all}}, 1 \leq i \leq p)$
  展開後トークン列

$R_{\mathbf{all}}$
  前処理によって行われる書き換え操作 r の集合

$Log = (r_1, r_2, \cdots, r_q | r_i \in R_{\mathbf{all}}, 1 \leq i \leq q)$
  トークン列 T_{input} から トークン列 T_{output} を得るまでに適用した書き換え操作の記録を示す 書き換え操作 r の操作系列.

書き換え操作 r の順列 Log は入力トークン列 S_{input} への適用順序に並んでおり，順に適用していくことで出力トークン列 S_{output} が得られるうなっている.

さらに，トークンの ID には同名であっても出現箇所に依存した ID が割り当てられ，マクロ展開時に与えられたパラーメタを構成するトークンは新たにユニークな ID が割り当てられた TokenRef による参照に置き換えられる．同じパラーメータのマクロ展開であっても，出現箇所が異なれば展開前，展開後どちらもトークン列を構成するトークンの ID は一致しない．そのため，本モデルでは，前処理前後での対応関係取得だけではなく，前処理内で行われるマクロ展開処理の追跡や，前処理後のトークン列から前処理前への復元が可能となる.

4. CSCPP と CPPXML の設計と実装

前章で述べた前処理追跡モデルの生成を行う機能を持つ前処理系 CSCPP と，モデル表現形式 CPPXML を実装した.

4.1 CPPXML

CPPXML は 3.3 で示したモデル定義を元に，トークンのユニークな識別番号，ファイルとの対応関係を加えて，XML 形式で表現したものである.

4.1.1 要素の説明

ファイル名表: Files

```
// header.h
01:  #define add(x,y)    x + y
02:  #define sum         add

// code.c
01:  #include "header.h"
02:  #define ONE 1
03:  sum(ONE,ONE);
```

図 4　解析コード例

Fig. 4　An example of code to parse.

```
<Files>
  <File Id="F1" Path="code.c" />
  <File Id="F2" Path="header.h" />
</Files>
```

図 5　ファイル名表

Fig. 5　A file name table.

表 2　入力コードと対応している例

Table 2　Example corresponding to input code

ID	トークン種別	キーワード
_61	Ident	sum
_65	Keyword	(
_67	Ident	ONE
_69	Keyword	,
_71	Ident	ONE
_73	Keyword	)
_87	Ident	;

トークンに対応づけられるファイル名を File ノードの Path 属性に格納した表．ファイル名とトークンの対応はトークンの File 属性に対応する File ノードの ID を格納することでする．ID は'F' に続けて整数値の形式で割り当てる．

図 4 に示すコードから得られるファイル名表を図 5 に示す．

トークン表: Tokens

後述の入力列，出力列，置換情報中に出現するトークンを格納した表．一つのトークンを一つのノードが示し，ノード名が表 1 で示したトークン種別と対応している．

図 4 に示すコードから得られるトークン表を図 6 に示す[*1]．

入力列: Input

Tokens 属性に前処理器が入力として読み取ったトークンを ID 番号で出現順に並べたもの．前処理前の C コードと等価である．図 4 に示すコードから得られるトークン表を図 7 に，部分列_61 _65 _67 _69 _71 _73 _87 が入力コード sum(ONE,ONE); と対応している例を表 2 に示す．

出力列: Output

[*1] 掲載スペースの関係で，カラム位置情報 Column と空白情報 Space は省略した．

```
<Tokens>
  <Keyword Value="#" Id="_1" File="F1" />
  <Ident Value="include" Id="_2" File="F1" />
  <UserPath Value="header.h" Id="_3" File="F1" />
  <NewLine Id="_4" File="F1" />
  <Keyword Value="#" Id="_5" File="F2" />
  <Ident Value="define" Id="_6" File="F2" />
  <NewLine Id="_7" File="F2" />
  <Ident Value="add" Id="_10" File="F2" />
  <Keyword Value="(" Id="_12" File="F2" />
  <Ident Value="x" Id="_14" File="F2" />
  <Keyword Value="," Id="_16" File="F2" />
  <Ident Value="y" Id="_19" File="F2" />
  <Keyword Value=")" Id="_21" File="F2" />
  <Ident Value="x" Id="_25" File="F2" />
  <Keyword Value="+" Id="_29" File="F2" />
  <Ident Value="y" Id="_32" File="F2" />
  <NewLine Id="_34" File="F2" />
  <Keyword Value="#" Id="_37" File="F2" />
  <Ident Value="define" Id="_39" File="F2" />
  <NewLine Id="_40" File="F2" />
  <Ident Value="sum" Id="_43" File="F2" />
  <Ident Value="add" Id="_46" File="F2" />
  <NewLine Id="_47" File="F2" />
  <Keyword Value="#" Id="_49" File="F1" />
  <Ident Value="define" Id="_51" File="F1" />
  <NewLine Id="_52" File="F1" />
  <Ident Value="ONE" Id="_55" File="F1" />
  <Number Value="1" Id="_58" File="F1" />
  <NewLine Id="_59" File="F1" />
  <Ident Value="sum" Id="_61" File="F1" />
  <TokenRef Id="_63" Ref="_46" />
  <Keyword Value="(" Id="_65" File="F1" />
  <Ident Value="ONE" Id="_67" File="F1" />
  <Keyword Value="," Id="_69" File="F1" />
  <Ident Value="ONE" Id="_71" File="F1" />
  <Keyword Value=")" Id="_73" File="F1" />
  <TokenRef Id="_75" Ref="_58" />
  <TokenRef Id="_78" Ref="_58" />
  <TokenRef Id="_83" Ref="_75" />
  <TokenRef Id="_84" Ref="_29" />
  <TokenRef Id="_85" Ref="_78" />
  <Keyword Value=";" Id="_87" File="F1" />
  <NewLine Id="_88" File="F1" />
</Tokens>
```

図 6　トークン表

Fig. 6　A token table.

```
<Input Tokens="_1 _2 _3 _4 _49 _51 _55 _58 _59 _61
               _65 _67 _69 _71 _73 _87 _88"
/>
```

図 7　入力列

Fig. 7　A sequence of input tokens.

```
<Output Tokens="_4 _7 _40 _52 _83 _84 _85 _87 _88"
/>
```

図 8　出力列

Fig. 8　A sequence of output tokens.

Tokens 属性に前処理器が出力として書き出したトークンを ID 番号で出現順に並べたもの．前処理後の C コードと等価である．図 4 に示すコードから得られるトークン表を図 8 に，部分列_83 _84 _85 _87 _88 が出力コードと対応している例を表 3 に示す．

置換情報: Deriverd

表 3　出力コードと対応している例
Table 3　Example corresponding to output code

ID	トークン種別	参照先	キーワード
_83	TokenRef	_75	
_75	TokenRef	_58	
_58	Number		1
_84	TokenRef	_29	
_29	Keyword		+
_85	TokenRef	_78	
_58	Number		1
_87	Keyword		;
_88	NewLine		\n

表 4　置換情報の種別と意味
Table 4　Type and meaning of replacement information

置換情報のノード名	意味
FromDefMacro	マクロ宣言の出現
FromBuildinMacro	組込みマクロを置換
FromObjectMacro	定数マクロを置換
FromFuncMacro	関数マクロを置換
FromConcat	トークン連結演算子の適用
FromStringize	文字列化演算子の適用
FromIf	文字列化演算子の適用
FromInclude	`#include` 指令の出現
FromIfdef	`#ifdef` 指令の出現
FromElif	`#elif` 指令の出現
FromElse	`#else` 指令の出現
FromUndef	`#undef` 指令の出現
FromError	`#error` 指令の出現
FromWarning	`#warning` 指令の出現
FromLineDirective	`#line` 指令の出現

前処理で適用されたトークン列の置換処理を適用順に並べたもの．置換情報の種別に応じたノード名が割り当てられる以外は共通で，ID は'D' に続けて整数値の形式で割り当てられ，子要素として From と To を持つ．前処理によって，トークン列中の From と一致する部分列が To で示されるトークン列に書き換えられたことを示しており，子要素は前処理による適用順で並ぶ．

置換情報のノード名と対応する意味を表 4 に，図 4 に示すコードから得られる置換情報を図 9 に示す．

4.2　C 言語前処理系解析器 CSCPP

作成した CSCPP は提案した前処理追跡形式である CPPXML を出力可能な前処理系で次に示す特徴を持つ．

- ISO/IEC C90 に準拠した展開処理
  – で示した手順に従って前処理を実行．
  – マクロ展開アルゴリズムには Dave Prosser's C Preprocessing Algorithm [18] を利用．
- 非標準指令・未対応プラグマの扱い
  – ISO/IEC C90 標準の前処理指令のみに対応．gcc 拡張など特定の処理系に依存する指令 (`#include_next` 等) には対応しない．
  – 未対応の`#pragma` 指令についても同様．
  – これらが出現した場合，エラーとするか，エラーとせずそのまま前処理結果に出力するかを選択可能．
- 前処理結果の出力
  – 通常の前処理系と同様の C コード形式の出力．
  – 提案した前処理追跡形式である CPPXML の出力．
- ポータブルな実装
  – C#言語のみを用いて，.NET Framework 4.5.2 の標準クラスライブラリの範囲のみで実装した．

```
<Deriverd>
<FromInclude Id="D1">
  <From TokenRefs="_1 _2 _3 _4" />
  <To TokenRefs="_4 _5 _6 _10 _12 _14 _16 _19 _21
      _25 _29 _32 _34 _37 _39 _43 _46 _47" />
</FromDefMacro>
<FromDefMacro Id="D2">
  <From TokenRefs="_5 _6 _10 _12 _14 _16 _19 _21
      _25 _29 _32 _34" />
  <To TokenRefs="_7" />
</FromDefMacro>
<FromDefMacro Id="D3">
  <From TokenRefs="_37 _39 _43 _46 _47" />
  <To TokenRefs="_40" />
</FromDefMacro>
<FromDefMacro Id="D4">
  <From TokenRefs="_49 _51 _55 _58 _59" />
  <To TokenRefs="_52" />
</FromDefMacro>
<FromObjectMacro Id="D5" MacroRef="M5">
  <From TokenRefs="_61" />
  <To TokenRefs="_63" />
</FromObjectMacro>
<FromObjectMacro Id="D6" MacroRef="M6">
  <From TokenRefs="_67" />
  <To TokenRefs="_75" />
</FromObjectMacro>
<FromFuncMacro Id="D10" MacroRef="4">
  <From TokenRefs="_63 _65 _75 _69 _78 _73" />
  <To TokenRefs="_83 _84 _85" />
</FromFuncMacro>
</Deriverd>
```

図 9　置換情報
Fig. 9　Replacement information

5.　評　価

開発した CSCPP の性能評価と，解析で得られる CPPXML 形式が実用的であることを確認するために CPPXML を用いて前処理展開の可視化ツールを開発した．その際の，入力には標準ライブラリを用いていない組み込みシステム向けソースコードの一部を用いて行った．

評価に用いた実行環境を次に示す．

CPU　Intel Core i5-3320M
RAM　16GB
OS　Windows 10 Pro

表 5　実行時間とファイルサイズの比較

Table 5　Comparison of execution time and file size

	ファイルサイズ	ステップ	処理時間
ソースコード 1 (#include ファイル数:26)			
	バイト数 / 比率	行数 / 比率	ミリ秒
元コード	83981 / 1.00	1784 / 1.00	-
C コード	313654 / 3.73	8787 / 4.92	556
CPPXML	4242767 / 50.52	52093 / 29.20	1647
cl.exe	335659 / 4.00	8231 / 4.61	119
ソースコード 2 (#include ファイル数:46)			
	バイト数 / 比率	行数 / 比率	ミリ秒
元コード	115023 / 1.00	2535 / 1.00	-
C コード	452608 / 3.93	15250 / 6.02	913
CPPXML	6360224 / 55.30	73989 / 29.19	1976
cl.exe	490573 / 4.26	14554 / 5.74	134

5.1　実行時間とファイルサイズの比較

CSCPP による前処理時間，得られる出力のステップ数とファイルサイズについて，通常の前処理実行時，CPPXML 用の実行時で測定した．また，比較用として Visual Studio 2015 の C/C++ コンパイラである cl.exe *2 での実行結果も測定した．

時間計測には Windows の elapsedtime コマンドを用いた

実行結果より，CPPXML 形式での出力では，ファイルサイズが元ソースコードの約 50 倍，プリプロセス結果同士で比べると約 14 倍増加している．これは通常の前処理情報に次に示す情報が XML 形式の利便性形式として追加されたためである．

- 書式がタグ形式に変換される．
- 出力するトークンには位置情報や空白要素などの要素が含まれる．
- 前処理によって行われた置換処理の情報含まれる．

たとえば，追跡子方式では得られる最終結果のサイズは前処理後の C プログラムのサイズの 7～13 倍であるが十分許容範囲であるとのことから，CPPXML 形式化によるファイルのサイズ増加も同様に十分許容範囲であるといえる．

処理時間ではテキスト形式と比べて 2～3 倍の実行時間を要している．これはトークンが確定すればそのまま出力することができる通常形式に対して，トークンの XML 形式への整形と CPPXML に含まれる情報の構成が主な要因である．追跡子方式では 946 行のファイルに対して 1.4 秒の時間を要しており，CSCPP は 1784 行のファイルに対して 1.64 秒である．そのため，本方式は十分に高速であるといえる．

*2 Microsoft(R) C/C++ Optimizing Compiler Version 19.00.24215.1 for x64

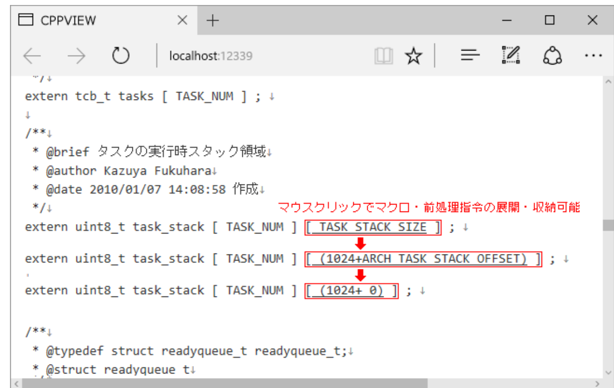

図 10　マクロ展開可視化ツール CppView

Fig. 10　Macro expansion visualization tool CppView

5.2　マクロ展開処理の可視化ツール

解析で得られた CPPXML 形式が有用性を有することを確認するために，CPPXML を読み込みマクロ展開を可視化する対話ツール CppView を開発した．ツールの実行例を図 10 に示す．

CppView は HTML と TypeScript を用いて作成したブラウザ上で動作するコード可視化ツールで，ソースコード上の任意のマクロや前処理命令をクリックすることで前処理の適用や巻き戻しを対話的に実行できる．また，ツールの実装規模は TypeScript 部分が 372 行，HTML 部分が 36 行と非常に小さく，開発も 1 人が約 6 時間の作業時間で作成できた．

このことから，提案する前処理モデルとそれを XML 形式化した CPPXML 形式はマクロ展開等の前処理の追跡を行うに十分な情報を持っており，また利便性も高いといえる．

6.　まとめ

本研究では，提案するモデルによって前処理前後での対応関係と展開過程を正確に追跡することが可能であることを示し，提案モデルを XML 記法で表現した記述方式 CPPXML と CPPXML を生成する前処理系 CSCPP を作成した．そして，CPPXML と CSCPP を用いた評価によって，提案手法は実用的であることと，前処理の展開過程が必要となるツールの実装において CPPXML 形式が有用であることを確認できた．

今後は，筆者らの開発している C 言語コーディング規約チェッカ [19] および，C 言語解析フレームワークにて CPPXML 形式を扱えるよう拡張し，前処理の展開情報を扱うことが可能なコード検査器への応用を行う．

参考文献

[1] 独立行政法人情報処理推進機構（IPA）技術本部 ソフトウェア高信頼化センター（SEC）：
ソフトウェア開発データ白書 2016-2017, 2016/10/01.

[2] David Evans, David Larochelle:

Improving Security Using Extensible Lightweight Static Analysis.
IEEE Software January/Feburary 2002, pp. 42-51, 2002.

[3] Programming Research:
QA Static Analyzers,
http://www.programmingresearch.com/static-analysis-software/qac-qacpp-static-analyzers/, 2017/01/15

[4] Renesas Electronics:
MISRA C ルールチェッカ SQMlint,
https://www.renesas.com/ja-jp/products/software-tools/tools/compiler-assembler/misra-c-rule-checker-sqmlint.html, 2017/01/15

[5] GrammaTech:
CodeSonar - Static Analysis SAST Software,
https://www.grammatech.com/products/codesonar, 2017/01/15

[6] Raincode:
Raincode Checkerr,
https://www.raincode.com/, 2017/01/15

[7] 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.11461170, 2002.

[8] GNU Project: The C Preprocessor,
https://gcc.gnu.org/onlinedocs/cpp/, 2017/01/15

[9] 松井 潔:
MCPP: A C Preprocessor with the Highest Conformance, 情報処理学会論文誌プログラミング Vol. 46, SIG1(PRO24), pp. 28-39, 2005

[10] 福安 直樹，山本 晋一郎，阿草 清滋:
細粒度ソフトウェア・リポジトリに基づいた CASE ツール・プラットフォーム Sapid,
情報処理学会論文誌, Vol.39, No.6, pp.1990-1998 (1998)

[11] 椴山嘉人，山本晋一郎：
Sapid P-model ver. 1.0 仕様書，
http://www.sapid.org/html2/P-model/P-model.pdf, 2017/01/15

[12] Greg J. Badros:
PCp3: A C Front End for Preprocessor Analysis and Transformation,
Technical report, University of Washington, 1997

[13] 権藤克彦, 川島勇人:
TBCppA: 追跡子を用いた C 前処理系解析器, コンピュータサイエンス, Vol. 25, No. 1(2008), pp. 105-123.

[14] GNU Project:
GNU GLOBAL source code tagging system,
https://www.gnu.org/software/global/, 2017/01/15

[15] Dimitri van Heesch:
Doxygen,
http://www.stack.nl/ dimitri/doxygen/index.html, 2017/01/15

[16] GNU Project:
The GNU C Preprocessor Internals,
https://gcc.gnu.org/onlinedocs/cppinternals/index.html, 2017/01/15

[17] Clang:
Clang: a C language family frontend for LLVM,
http://clang.llvm.org/, 2017/01/15

[18] Dave Prosser:
A corrected and annotated version of the X4J11/86-196 document,
http://www.spinellis.gr/blog/20060626/cpp.algo.pdf, 2017/01/15

[19] 福原 和哉, 高橋 耶真人, 猪股 俊光, 新井 義和, 今井 信太郎
組込みソフトウェア向けコーディング規約チェッカのためのカスタマイズの一方式：
FIT2012 第 11 回情報科学技術フォーラム.