Vol.2010-SE-168 No.10
Vol.2010-EMB-17 No.10
2010/6/2

コードパターンの検出に適した C言語前処理系解析器の開発

福原 和哉 [†1]　猪股 俊光 [†1]
新井 義和 [†1]　曽我 正和 [†2]

我々はソースコードレビューに頼って検出を行っていた定式化ができない不具合を検出するために，コードパターンを用いた解析手法を考案し，ANSI-C 言語で記述された組み込みソフトウェア向けの検査ツールとして実装し評価を行ったが，C言語前処理系によって処理が行われる識別子を含むパターンについては正しく処理できない問題があった．

そこで，C言語前処理系の置換動作などを制御可能とすることでこの問題を解決するコードパターンの検出に適したC言語前処理系を提案し，作成・評価した．その結果，従来は検出を行うことが出来なかった不具合をコードパターン化し，検出可能であることがわかった．

Development of C preprocessor analyser that is appropriate for detection of code pattern.

Kazuya Fukuhara ,[†1] Toshimitsu Inomata ,[†1]
Yoshikazu Arai [†1] and Masakazu Soga [†2]

To automate detection of problem that was not able to be formulated that used source code review, we designed method for code pattern analysis. We made inspection-tool for embedded software that is written by C language, and evaluated it. As a result, pattern including the identifier processed in the preprocessor is not correctly treatable. Then, we propose preprocessor that is appropriate for the detection of the code pattern that controlled operation of preprocessor, and evaluate it. The experimental results show that the propose method can trouble that cannot be detected up to now can be detected.

1. はじめに

ソフトウェアの品質を確保するための手法の一つであるコードレビューは，ソフトウェア開発工程で見過ごされた誤りを検出・修正するためにソースコードを検査をする作業である．コードレビューではコーディング規約に準拠しているかどうか，不具合の要因となりそうなコードが含まれているかどうかなどが検査され，ソフトウェア品質の改善に有効とされている．しかしながら，コードレビュー中は担当者が一定時間拘束される，目視による作業が多く，品質にばらつきが生じやすいという問題点がある．

そこで，従来，目視で行われていた不具合の要因となるコードの検出の自動化を目指し，筆者らはコードパターンを用いた静的解析手法を考案し，ANSI-C 言語で記述された組み込みソフトウェアを対象とする検査ツールとして実装した．その結果，不具合の要因となるコードをパターン記述言語で表した後，ソースコードとパターンマッチングすることで不具合の要因となるコードの有無を検査できることが可能であるという結果が得られた[1)2)]．

しかしながら，その過程でマクロ定義や条件付きコンパイルなどの前処理系が原因となって，パターンマッチングによる解析・検出が失敗するケースの存在が明らかになった．Erust らの調査[3)] では，前処理系を用いないプログラムは存在せず，`#define` などの前処理指令は全ステップ数の 8.4%を占めている．そのため，C言語を対象とした検査ツールでは前処理指令やマクロ呼出しを扱う必要がある．

このような背景から，本研究ではこの問題を解決するコードパターンの検出に適したC言語前処理系を考案し，作成・評価を行った．

2. コードパターンの検出方式と前処理系

ソースコードの不具合箇所の検出は例 2 に示すような不具合の要因を**表 1** に示すパターン記述言語で表し，得られたコードパターンとソースコードとのパターンマッチングをすることにより行われる．パターンマッチングは，Santanu ら[4)] の方法を拡張したオートマトンを基に行われる．開発した検査ツールの使用例を次の例 2 を用いて示す．

例 2　不具合事例

タイマー制御レジスタ `TCCR1A` に対して代入を伴う操作を行う際に，事前に割り込み禁止（`cli()`

†1 岩手県立大学
Iwate Prefectural University
†2 岩手県立大学地域連携研究センター
Iwate Prefectural University Regional Cooperative Research Center

Vol.2010-SE-168 No.10
Vol.2010-EMB-17 No.10
2010/6/2

表 1 パターン記述記号
Table 1 Pattern description symbols

構文要素	単体	複数	接頭語	記述例
宣言	$d	$*d	○	$d_Member
型	$t		○	$t_cell
変数	$v	$*v $v_counter	○	
関数	$f		○	$f_printf()
式	$e	$*e	×	i = $e;
文	@	@*	×	if ($e) @;
複合代入演算子	$assignop	×	×	i $assignop $e;
ブロック開始	{{...{	$*{	×	$*{ puts($e) $}
ブロック終了	}...}}	$*}	×	
否定ブロック	$!{ ... $!}		×	$!{ @*; break; $!}

の呼び出し) と事後にタイマカウンターレジスタ OCR1A のリセット，および，割り込み禁止解除 (sei() の呼び出し) を行っていない．

例 2 に対応するコードパターンは次のようにして作成される．

(1) レジスタ TCCR1A と レジスタ OCR1A をそれぞれ変数 $v_TIMER_CR と $v_OCR1A で表す．
(2) レジスタ TCCR1A に対する操作を パターン $v_TIMER_CR $assignop $e; で表す．
(3) 割り込み禁止を行っていないことを否定ブロックを用いて $!{ cli(); @*; $!}と表す．
(4) レジスタ OCR1A のリセットを行っていないことを否定ブロックを用いて $!{ @*; $v_TIMER_IR = 0; @*; $!} と表す．
(5) 割り込み禁止解除を行っていないことを否定ブロックを用いて $!{ @*; sei(); $!} と表す．

これらをまとめると**図 1** のコードパターンが得られる．

開発した検査ツールにこのコードパターンと**図 2** のソースコードを適用すると，**図 3** に示すように図 2 のソースコードの 4 行目から 5 行目にかけて，タイマー制御レジスタのリセット (v_TIMER_IR = 0;) が行われていないことが検出される．

3. C言語解析ツールにおける前処理系の問題

3.1 前処理系の問題点

前処理前のソースコードはC言語の文法を完全に満たしていない場合や，前処理系に依存

```
$v_TIMER_CR="TCCR1A"; /* タイマーの制御レジスタ */
$v_TIMER_IR="OCR1A";  /* タイマーカウンタレジスタ */
%%  /* ラベル記述とパターン記述の区切り */
$!{
    cli();                  /* 割り込み禁止 */
    @*;                     /* 任意の文が来る */
$!}

$v_TIMER_CR $assignop $e;  /* タイマー制御レジスタに対して操作 */

$!{
    @*;                     /* 任意の文が来る */
    $v_TIMER_IR = 0;        /* タイマーカウンタレジスタをゼロリセット */
$!}

$!{
    @*;                     /* 任意の文が来る */
    sei();                  /* 割り込み許可 */
$!}
```

図 1 不具合記述の例
Fig. 1 An example of code pattern

```
void __attribute__((naked)) start_timer() {
  cli();
  tm_run = TRUE;
  TCCR1A = 1 << 3;
  sei();
}
```

図 2 不具合を含むソースコード
Fig. 2 An example of source code include fault

```
c:\> patchk -p fail.pat timer.c
check result: match
*pattern line 17: $v_TIMER_IR is not found in source code. (source line 4--5)
```

図 3 検査結果
Fig. 3 A result of code pattern check processor

Vol.2010-SE-168 No.10
Vol.2010-EMB-17 No.10
2010/6/2

する特別な命令を用いている場合があるため，C 言語の解析ツールでの正確な構文解析は難しい．たとえば，図 2 に示すソースコードを例とした場合，タイマー制御レジスタ TCCR1A がマクロによって

```
#define TCCR1A (*(volatile unsigned char *)(0x2F))
```

と定義されていた場合，前処理によって

```
TCCR1A = 1 << 3;  は (*(volatile unsigned char *)(0x2F)) = 1 << 3;
```

となるため，処理後のソースコードには識別子 TCCR1A は存在しないことになり，パターン検出の大きな障害となる．また，前処理系に依存する特別な命令である __attribute__((naked)) を用いているため C 言語の文法を満たしておらず，正確な構文解析を行うことができない．

3.2 既存研究による解決策

従来の研究では，前処理の問題を解決するために大きく分けて三種類の手法を用いている．

(1) コンパイラの生成する情報を利用

gcc の-dM，-dD オプションなどから得られる情報を元に対応情報を得る．この方式は，容易に正確な情報を得ることができる反面，得られる情報はコンパイラに依存してしまう問題がある．たとえば，例に挙げた gcc の-dM，-dD オプションではマクロの展開情報が得られないため，字句・構文ベースの検査を行う我々の目的には適さない

(2) 前処理系を模倣

G.J.Badros ら[5] は前処理系の動作を模倣する処理系を作成することで，前処理系で失われてしまう情報を得ている． 前処理系の模倣では以下のような問題点に対応しなければならないが，前処理の中で行われる望まれない処理の抑制などといったことが容易に可能となる利点もある．

- 非標準の#pragma の解釈
- 前処理系毎に独自拡張された指令への対応
- 処理系毎に付加される独自マクロへの対応

ただし，前処理系の適用範囲や保守コストの面で課題がある．

(3) 追跡子方式

K.Gondow ら[6] は予め前処理を施す前に，追跡子をソースコードに埋め込み，前処理後のソースコードの対応付けから対応情報を得る手法を提案している．この手法は既存の前処理系に一切手を加えることなく，最小限の手間のみでマクロの展開情報を得ることができるため，(2) の方式の多くの問題点を解決している．しかし，構文木ベースの検査を行う際には，単なる字句の置換状況だけではなく，構文木としての置換状況も必要となる．そのため，常に追跡子を解析して検査パターン中に登場する要素と一致するか否かを検出する必要がある．そして，多くの場合は，要素と追跡子は一致しないため，巨大なソースコードであるほど無駄に追跡子を探索することになってしまう．また，従来のパターンベースの検査ツールに生成した追跡子を用いるための修正が必要となる．

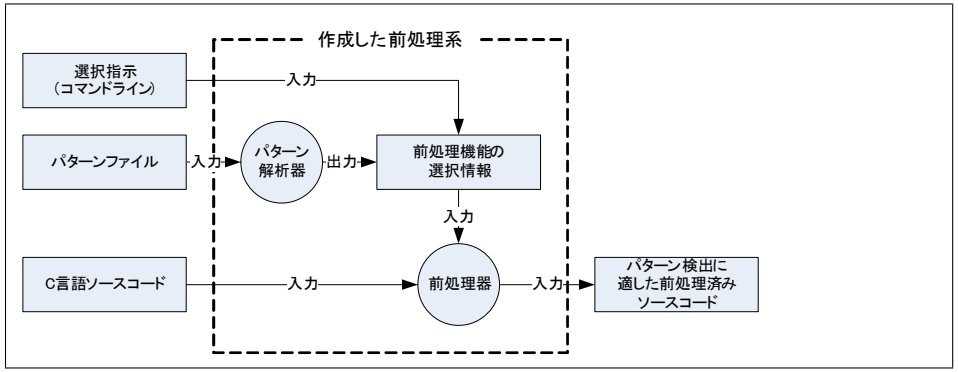

図 4 提案手法の概要
Fig. 4 Outline of proposal technique

4. 考案手法の全体構成

3.2 で述べたように，既存の手法には一長一短があり，本研究のように構文木ベースのパターンマッチを行う際には，前処理の影響を最小限に抑える必要がある．特に，組込みソフトウェア向けの前処理系には特殊な意味を持つ前処理指令などが存在するため，汎用の前処理系では対応できない場合が多い．そのため，本研究は前処理系を模倣し，前処理を選択可能とすることで従来の構文木ベースのパターンマッチを可能とする手法を考案した．

提案する手法の概要を**図 4** に示す．

この手法では，前処理時にマクロの展開，#include，コメント文，#pragma に対する処理をユーザーが自由に選択可能となるため，追跡子方式のように検査に不要な箇所の展開を抑制することができる．

Vol.2010-SE-168 No.10
Vol.2010-EMB-17 No.10
2010/6/2

```
void start_timer() {
  cli();
  tm_run = TRUE;
  (*(volatile unsigned char *)(0x2F)) = 1 << 3;
  sei();
}
```

図 5 表 1 の通常のプリプロセス結果
Fig. 5 Usual preprocessed result of 表 1.

```
void start_timer() {
  cli();
  tm_run = 1;
  TCCR1A = 1 << 3;
  sei();
}
```

図 6 提案手法を用いた図 1 の前処理結果
Fig. 6 Preprocessing result of 図 1 that uses proposal technique.

前処理時の処理の選択指示は，外部から与えるパターンファイル中の記述，もしくは，コマンドラインオプションとして与える．

考案手法を用いた検出の流れを図 2 に示すソースコード中から表 1 に示すパターンを探す場合を例に挙げて説明する．

以下のが定義されている場合，通常の前処理系を用いた場合は，**図 5** に示す結果が得られる．この結果に対して開発したパターン検出ツールを適用した場合，TCCR1A と一致する場所が存在しないため，不具合箇所は存在しないという結果が得られてしまう．

```
#define TCCR1A (*(volatile unsigned char *)(0x2F))
#define __attribute__((x))
#define TRUE 1
```

一方，マクロ TCCR1A の展開を抑制するように動作を選択して前処理を行った場合，**図 6** に示す結果が得られる．この結果に対して開発したパターン検出ツールを適用した場合，通常の前処理系を用いた場合と違い，4 行目に TCCR1A への代入が存在し，それ以降に OCR1A への代入が存在しないため，不具合箇所が存在するという結果を得ることができる．

5. 評 価

考案した手法にもとづいて C 言語を対象として動作する処理系を試作しある種の組込みソフトに適用し，評価を行った．

表 2 評価に用いた組込みソフトウェアの詳細
Table 2 Spec of automotive software used for evaluation

*.c ファイル数	141
*.h ファイル数	158
総ステップ数	82298
前処理後ステップ数	209146
総 include 数	5827
総 define 数	135352

表 3 前処理時間の比較
Table 3 Evaluation of processing time

既存の前処理系	11846 ms
試作した前処理系	39239 ms
１０個の複文マクロの展開を抑制時	36131 ms

表 4 パターン検出に要した時間の比較
Table 4 Comparison of pattern detection time

	複文マクロを展開	複文マクロの展開を抑制
プリプロセス時間	591ms	578ms
検出時間	1433ms	985ms

適用したソフトウェアの総ファイル数，総ステップ数は**表 2** に示す．

5.1 前処理に要する処理時間の評価

既存の前処理系を用いた場合と，評価のために試作した前処理系による前処理時間の比較結果を**表 3** に示す．

この結果，試作した前処理系は純正開発環境の前処理系と比べて約 3 倍ほど遅く，さらに複文マクロの展開を抑制した場合は実行速度の差がより大きくなることがわかった．

また，この中から一つファイルを選び，複文マクロの展開を選択した場合と，選択しない場合でのパターン検出に要した時間の比較結果行ったところ，**表 4** に示す結果が得られた．この結果より，マクロの展開を選択しない場合はより高速に検出を行うことができる．

6. 考案手法の課題

我々の考案した手法には以下のような制約がある．

特に前処理系での利用頻度の高い，次の（１）と（３）の用途に対応することは今後必要

Vol.2010-SE-168 No.10
Vol.2010-EMB-17 No.10
2010/6/2

である．

(1) 展開を抑制したマクロを含む前処理指令 (`#if` 等) があるとエラーとなる．この制約は条件付きコンパイルなどの障害となる．

(2) 特定のファイル中でのみマクロの展開を抑制することができない．同名のマクロであれば全て展開が抑制される．

(3) `##`で結合される箇所のマクロの展開はエラーとなる．これは抑制してしまった場合に，正しい結果が得られないためである．

7. 結 論

本手法を用いてC言語前処理系の各種の機能（置換など）を選択可能とすることで，従来は検出を行うことができなかったソースコードに対しても、コードパターンを用いて不具合箇所の検出が可能となり，コードパターンの検出速度も向上することを示した．

今後は，6章の制約を緩和しつつ，より多くの不具合コード検出への応用を行いたい．

参 考 文 献

1) 福原和哉，猪股俊光，新井義和，曽我正和：
ソフトウェア製品の不具合原因となるコードのパターンを用いた検証手法，第4回システム検証の科学技術シンポジウム，2007.

2) 福原和哉，猪股俊光，新井義和，曽我正和：
ソフトウェア製品の不具合原因コードパターン検出のための静的解析法，電子情報通信学会 2008 年総合大会，2008.

3) 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.

4) Santanu Paul，Atul Prakash：A Framework for
Source Code Search Using Program Patterns,
IEEE Transactions on Software Engineering,
Vol.20. pp.463-475 (1994).

5) G.J.Badros: PCp3: A C Front End for Preprocessor Analysis and Transformation, Masters Thesis, 1997.

6) K.Gondow, H.Kawashima, T.Imaizumi, TBCppA: a Tracer Approach for Automatic Accurate Analysis of C Preprocessor's Behaviors, 8th IEEE Int. Working Conf. on Source Code Analysis and Manipulation (SCAM2008), pp.35–44, (2008).