特別研究報告

題目

パターンマイニング技術を用いた実時間プログラムのコーディングパターン検出

指導教員
井上 克郎 教授

報告者
中村 勇太

平成 27 年 2 月 13 日

大阪大学 基礎工学部 情報科学科

平成 26 年度 特別研究報告

パターンマイニング技術を用いた実時間プログラムのコーディングパターン検出

中村 勇太

内容梗概

コーディングパターンとは複数のモジュールに分散する定型的なコードである．ファイルのオープン · クローズや例外処理などのコーディングパターンはプログラムの振る舞いや守らなければいけないルールを表すため，再利用やバグの検出に利用できる．コーディングパターンが発生しやすいプログラムとして実時間プログラムがある．

実時間プログラムとは決められた時間内に計算を完了させる必要がある組込みプログラムのことである．実時間プログラムが起こす不具合が与える影響は非常に大きいため，高い信頼性が求められる．例えば，車載制御プログラムの不具合による事故や販売した製品の回収 · 修正による経済的損害が起きる．実時間プログラムからコーディングパターンを検出し，再利用やバグ検出に用いることは実時間プログラムの信頼性向上に繋がる．

既存のコーディングパターン検出の研究では Java プログラムからコーディングパターンの検出を行った．しかし，C 言語で記述された実時間プログラムにおけるコーディングパターン検出はまだ行われていない．

そこで本研究では C 言語で記述された実時間プログラムの特性を考慮したうえでコーディングパターンを検出した．本研究ではまず，実時間 OS(オペレーティングシステム) のプログラムからコーディングパターンの構成要素として関数呼び出し，制御構造，goto 文，ラベル，return 文を抽出した．その後，抽出された要素に基づいてシーケンシャルパターンマイニングを適用した．シーケンシャルパターンマイニングは順番を考慮した，頻出する要素の組合せを検出するマイニング技術である．その後，検出されたコーディングパターンの絞り込みを行い，ラベルや制御構造の対応が取れており，関数呼び出しが 2 つ以上含まれているコーディングパターンのみを選択した．さらに，絞り込み後のコーディングパターンの有用性を評価した結果，仕様書の情報を含んだ意味のあるコーディングパターンを検出できたことを確認した．このことから本研究では，今後これらのコーディングパターンを再利用やバグの検出に利用していくことができると判断した．

主な用語

コーディングパターン
パターンマイニング技術
実時間プログラム

目 次

1　まえがき

コーディングパターンとは複数のモジュールに分散する定型的なコードである [1]．例えばC言語では，ファイルのオープン `fopen()` を行った後は必ずファイルをクローズ `fclose()` する必要がある．この場合の `fopen()` と `fclose()` の関数呼び出しの対応はコーディングパターンである．また，特定の処理を行う際に発生する例外を処理するための定型的なコードもコーディングパターンの 1 つである．

コーディングパターンはプログラムの振る舞いや実装において守らなければいけないルールを表している．そのため，コーディングパターンは再利用やバグの検出に利用することができる．具体的には，あるプロダクトから検出されたコーディングパターンをプログラムの部品として別プロダクトの開発に再利用することができる．また，コーディングパターンに基づいて守らなければいけないルール通りの記述ができているかチェックをすることでバグの検出を行うことができる．

実時間プログラムとは正常な結果が求められるだけでなく，決められた時間内に計算を完了させる必要がある組込みプログラムのことであり，自動車のエンジン部，原子力発電所，航空制御システムなど社会において重要な役割を担っている [2]．

実時間プログラムの不具合が与える影響は大きいため，実時間プログラムでは高い信頼性が要求される．例えば，自動車制御で不具合が発生した場合は人命にかかわる事故に繋がる恐れがある．さらに，販売した製品を回収して修正する必要があるため高いコストが掛かる．このように実時間プログラムの不具合によって人的な被害を与えるだけでなく経済的損失をこうむる可能性があるため，不具合を減らし信頼性が高いプログラムを開発する必要がある．

プログラムの信頼性を向上させる方法として部品の再利用やバグの検出がある．Mohagheghiらは，再利用されたコンポーネントとそうではないコンポーネントを比較し，再利用されたコンポーネントの方が欠陥密度が低かったことを実験により確かめた [3]．さらに，再利用されたコンポーネントの方が重要度が高い欠陥であった．このことから，再利用が信頼性を向上させる要因であると述べている [3]．また，プログラムとしての不具合を発生させるコーディング上のバグを発見，修正することは信頼性を高める．そこで本研究では，実時間プログラムの信頼性の向上を目的に，再利用やバグ検出のためのコーディングパターン検出を行った．

石尾らは Java プログラムに対してのコーディングパターン検出を行った [1]．彼らはプログラムの各処理の振る舞いを理解するためにはメソッド呼び出しパターンが有用だと考え，対象プログラムから関数呼び出しや制御構造を要素として抽出し，その要素列をもとにコーディングパターンの検出を行った．

しかし，彼らの研究では実時間プログラムが持つ特性を考慮していないため実時間プログラムに対して同じ手法を用いることはできない．そこで本研究では実時間プログラムが主にC 言語で実装されること [4]，ジャンプ命令やプリプロセッサ命令が多用されることを踏まえ，コーディングパターンの検出を行った．コーディングパターンの検出には順番を考慮した，頻出する要素の組合せを求めるシーケンシャルパターンマイニングを用いた．さらに，得られたコーディングパターンの有用性について評価した．

以降，2 章では本研究の背景を説明し，3 章では本研究におけるコーディングパターン検出の手法について述べる．4 章では手法の適用実験について説明し，最後に 5 章で本研究のまとめと今後の課題を述べる．

2　背景

この章では，まずコーディングパターンについて説明する．次に，コーディングパターンを検出するための技術としてパターンマイニング技術について説明する．そして，今回の研究対象である実時間プログラムについて述べる．

2.1　コーディングパターン

あるモジュールで使われているコードをそのまま，あるいは必要に応じて一部を修正して再利用することはコーディングにおいて頻繁に行われている [5]．そして複数モジュールでコードの再利用が行われる場合，それらのコードは保守性の向上を目的として 1 つのモジュールにまとめるべきである [6, 7]．しかし，横断的関心事と呼ばれる，複数モジュールに分散しているコードが存在する場合もある [8]．本研究では，こうした複数のモジュールに存在する定型的なコードをコーディングパターンと呼ぶ．

Java プログラムを対象としたコーディングパターン検出の研究では，プログラムの各機能がどのように振る舞うか理解するためにメソッド呼び出しのパターンを知ることが有用であるとしている [1]．また，定型的なメソッド呼び出し列はしばしば，特定の制御構造を伴うとし，コーディングパターンをメソッド呼び出しとそれに付随する制御構造要素の列として捉えている．

2.2　パターンマイニング技術

この節では，コーディングパターンを検出するための技術として用いるパターンマイニング技術について述べる．

膨大なデータの中から情報，知識を取り出す技術がデータマイニング技術であり [9]，パターンマイニング技術はその中でも特に，データ中に高頻度で出現する特徴的なパターンを見つける技術である．

パターンマイニング技術には頻出パターンマイニングとシーケンシャルパターンマイニングがある．頻出パターンマイニングは，例えば A が発生する場合 B も発生する，といったルールを発見するための相関ルールマイニングや単純に頻出するアイテムを見つけるための頻出アイテムマイニングなどがある．スーパーで客ごとの購入リストを分析して購入されやすい商品の組み合わせ (パターン) を調査することにこの頻出パターンマイニングは適している．

頻出パターンマイニングは発見されるパターンに関して，その順番は考慮しない．先の例で言えば，商品 A と商品 B が購入される場合には商品 C が同時に購入されることが分かっても，商品 A と商品 B のどちらが先に購入されたのかまでは分からない．その一方，順番

表 1: 系列データベース ([13] から引用)

SID	EID(時間)	Items
1	10	C D
1	15	A B C
1	20	A B F
1	25	A C D F
2	15	A B F
2	20	E
3	10	A B F
4	10	D G H
4	20	B F
4	25	A G H

を考慮したパターンマイニングがシーケンシャルパターンマイニングである．パターンの順番性を重視するため，WEB 利用者のアクセス履歴をもとにどのような順番で WEB サイトが閲覧されているかを調査することに適している [10, 11].

Kagdi らは，プログラムに対してパターンマイニングを適用する場合，頻出パターンマイニングよりも順番性を持つシーケンシャルパターンマイニングのほうが優れると述べている [12]．本研究ではこの研究に基づいて，順番を考慮したマイニング手法であるシーケンシャルパターンマイニングを用いてプログラムからコーディングパターンの検出を行う．

本研究では，シーケンシャルパターンマイニングのアルゴリズムの 1 つである Zaki の考案した SPADE(Sequential PAttern Discovery using Equivalence classes) アルゴリズムを使用する [13]．SPADE は入力として表 1 のような系列データベースと最小支持度を受け取る．系列データベースは複数の系列データから構成されており，各系列データは系列 ID(SID)，時間もしくは出現順序 (EID) そしてアイテム集合 (Items) で構成される．SPADE は系列データベースから表 2 のようにアイテムごとに SID と EID をまとめたリストを作成し，そのリスト同士を結合させることによって新たな系列パターンを検出していく．例として，A と B という長さ 1 の系列パターンから A-B という長さ 2 の系列パターンを得る際に作成されるリストを表 3 に示した．また，支持度は系列パターンの出現頻度を表す値である．例えば表 1 において `D B A` という系列パターンは SID=1，SID=4 でみられるため，この系列パターンの支持度は 2/4=0.5 となる．SPADE はリストの結合によって新たな系列パターンを検出した際に，最小支持度未満の系列パターンをその時点で除外する．

表 2: 表 1 から作成される，SID と EID をまとめたリストの一部

A		B		...	...
SID	EID	SID	EID		
1	15	1	15		
1	20	1	20		
1	25	2	15		
2	15	3	10		
3	10	4	20		
4	25				

表 3: 表 2 の A，B に関するリストを結合して得られるリスト．

A-B	
SID	EID
1	20

2.3 実時間プログラム

我々が普段接している機器はプログラムでその動作を制御されており，機器に組込まれているプログラムは組込みプログラムと呼ばれている．組込みプログラムの中でも決められた時間内にその処理 (計算) を完了させる必要がある，時間的制約を持つプログラムを実時間プログラムと呼ぶ [2]．また，実時間プログラムは時間的制約だけではなく厳しいリソース制約が課せられるためオブジェクト指向の構造を崩して類似コードを複数個所に記述する傾向にある [14]．このことから，実時間プログラムではコーディングパターンが発生しやすくなると考えられる．

実時間プログラムに課せられた時間制約とリソース制約を守るためにプログラム上で行われる工夫として次の 2 点を挙げる．

- ジャンプ命令の多用
- プリプロセッサ命令の多用

ジャンプ命令の多用は時間制約を守るために行われる．ある処理から別の処理へ間に一切の処理を挟まず直接ジャンプすることは実行時間の短縮に繋がる．また，プリプロセッサ命

令を用いて実行コードを切り替えることにより動的に実行コードを変更することがなくなるため使用するリソースを抑えることができる．

2.1 節で述べた通り，石尾らの研究はコーディングパターンは関数呼び出しの列とそれに付随する制御構造から構成されるとしている．しかし，実時間プログラムではジャンプ命令とプリプロセッサ命令を多用する特性があるため既存研究の手法のままでは本研究の目的である，C 言語で記述された実時間プログラムのコーディングパターンを検出することはできない．

本研究では手法を改善し，コーディングパターンを構成する要素を追加することで実時間プログラムからコーディングパターンの検出を行った．本研究で行ったコーディングパターン検出の手法については次章で説明する．

3　コーディングパターン検出手法

本研究では，SPADE アルゴリズムを用いて C 言語で記述された実時間プログラムからコーディングパターンの検出を行う．この章ではその手法について述べる．

手法は以下の 3 つのステップから構成されており，その概要を図 3 に示した．

ステップ 1 コーディングパターン構成要素の抽出及び系列データベースの作成

ステップ 2 コーディングパターン検出

ステップ 3 マイニング結果の絞り込み

ステップ 1 では対象プログラムからコーディングパターンの構成要素を抽出し，系列データベースを作成する．前章で，実時間プログラムが持つ特性により石尾らの研究の手法のままでは実時間プログラムにおけるコーディングパターンが検出できないことを述べた．本研究では，彼らの研究で抽出されていた関数呼び出しと制御構造だけではなく実時間プログラムの特性を考慮した要素を抽出した．詳細は 3.1 節で述べる．

ステップ 2 では SPADE アルゴリズムを用いて系列データベースと最小支持度からコーディングパターンの検出を行う．今回の研究では，統計ソフト R と R が提供するパッケージを用いてコーディングパターンを検出した．ステップ 2 については 3.2 節で説明する．

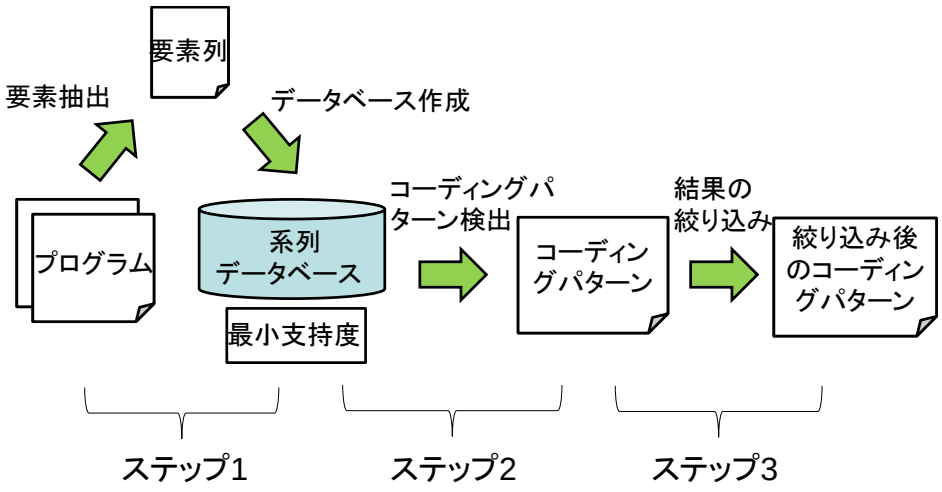


図 1: コーディングパターン検出手法の概要図

パターンマイニングの結果として出力されるコーディングパターンは膨大な数になる．そのままでは開発者が手作業でコーディングパターンの確認を行うことが困難なためステップ3ではマイニング結果の絞り込みを行う．その詳細については3.3節で述べる．

3.1 コーディングパターン構成要素の抽出及び系列データベースの作成

ステップ1ではコーディングパターンを構成している要素の抽出と系列データベースの作成を行う．

3.1.1 コーディングパターン構成要素の抽出

本研究の手法ではコーディングパターンを構成する要素として以下の要素を抽出した．

- 関数呼び出し
- 制御構造
- goto文
- return文

関数呼び出し

本研究では関数呼び出しについてはその関数名を要素とする．例えばプログラム上で関数呼び出し文`A(a+b,c)`が存在したとき，ここから得られる要素はA()となる．プログラムの振る舞いを知るためには関数名の列だけで十分と考え，本研究では関数呼び出しの引数については考慮していない．

C言語で記述されたプログラムではマクロで定義された関数を呼び出す場合がある．マクロ関数の呼び出しについては，マクロを展開せずに関数名をそのまま要素としている．マクロ関数の定義名は，開発者がその名前から処理内容を把握しやすいように名付けていると考えられる．図3.1.1もその例で，展開後のコードから要素を抽出するよりもマクロ関数名をそのまま抽出する方が処理内容はわかり易い．したがってここではマクロ関数の名前を要素として抽出している．

制御構造

本研究の手法では制御構造としてif文，while文，for文を考えた．これらの制御構造からどのように要素が抽出されるかは図3.1.1で示されている．

if 文は IF，ELSE，END-IF を基本構造としており，条件式で関数呼び出しが用いられていればその関数名を IF の直前の要素とする．また，if 文中に現れる他の要素はすべて IF と END-IF で挟む (図 3.1.1(a))．

while 文や for 文は LOOP，END-LOOP を基本構造しており，while 文の条件式中の関数名や for 文の初期化式，条件式，変更式中の関数名を適切な位置に記述している（図 3.1.1(b)(c)）．これらの位置は実行される順番に依存している．

do-while 文は条件式によらず 1 回はループの中身を実行する while 文と捉えることができるので，while 文の最初の条件式に関する部分を取り除いた形になる．

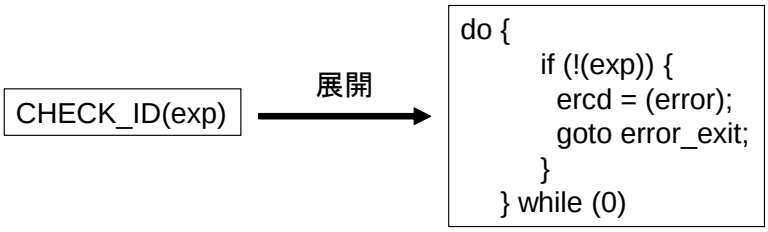


図 2: マクロ関数と展開後のコード

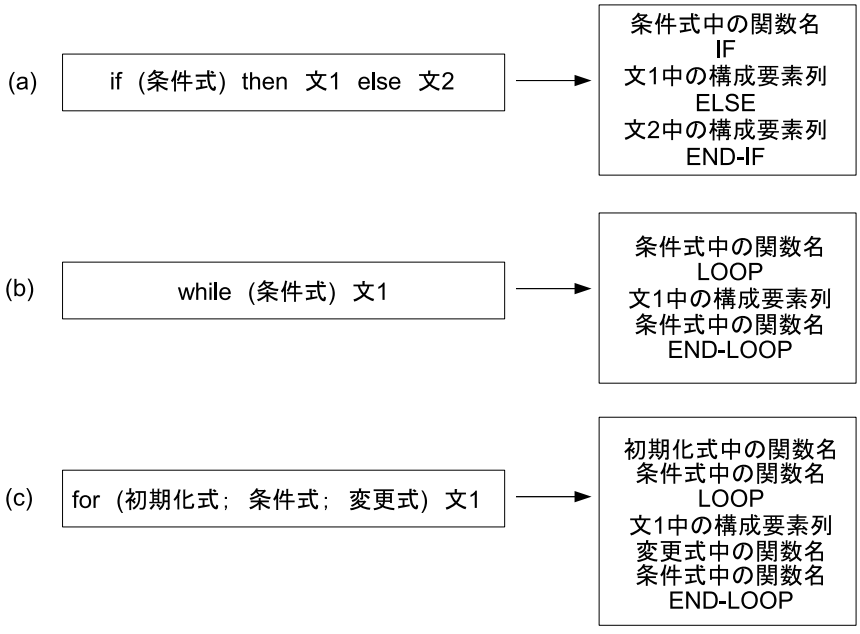


図 3: 制御構造から取り出される要素

goto 文

2.3 節で説明したように，実時間プログラムではジャンプ命令が多用されるため本研究ではC 言語におけるジャンプ命令である goto 文を要素として抽出することにした．また，goto 文は単体では意味を持たず，ジャンプ先のラベル名が後に続くためそのラベル名も要素として取り出した．

return 文

ジャンプ命令が多用されることによって実時間プログラムの実行時にプログラム上を行き来することになる．したがって，処理の流れを掴むために return も要素とするべきである．図 3.1.1 はプログラムから要素を取り出した結果の例である．図 3.1.1(a) は return を要素として含めない場合の例を示している．この場合，label1 にジャンプしてきた後にそのまま処理 2 を実行するのかそれとも処理 1 を実行した段階でその関数を抜け，処理 2 は実行されるべきではない処理なのか分からない．一方図 3.1.1(b) のように return が含まれている場合は，label1 にジャンプした場合は処理 2 は決して実行されないという処理の流れが明確に判断できる．よって本研究では return 文を要素として抽出し，さらに文中に含まれるラベルも考慮した．

図 4: return を要素として抽出した場合と抽出しなかった場合

図 3.2 は実際のプログラムで定義されている関数から要素を抽出した様子を表している．関数呼び出しはその名前のみを要素として抽出され，その他の制御構造やジャンプ文も同様に抽出されていることが分かる．

3.1.2 系列データベースの作成

続いて，得られた要素列から次のステップの入力となる系列データベースを作成する．作成されるデータベースの一部を図 3.2 に示した．データベースは表 1 と類似した構成であり，系列 ID(SID)，時間や出現順序を表す ID(EID)，アイテム数，アイテム集合から成る．ここでは SID として関数番号を指定している．すなわち，n 番目に定義されている関数から抽出される要素はすべて SID=n となる．EID はその要素が関数内で何番目に出現した要素であるかを表す値を指定している．図 3.2 で示した抽出される要素を例にとると，最上段の LOG_ACTTSK_ENTER() は EID=1 となり，以降の要素は順番に 2，3，…と続く．アイテム数は要素数のことでありその値は常に 1 である．アイテムは抽出された要素そのままである．

3.2 コーディングパターン検出

STEP2 では 3.1.2 で作成した系列データベースからシーケンシャルパターンマイニングを用いてコーディングパターンの検出を行う．

コーディングパターン検出には R を用いる．R とはフリーの統計解析用の言語及び開発環境である [15]．R ではユーザが作成したプログラムを自由にインストールすることでその機能を拡張することができる．このプログラムはパッケージと呼ばれており，その内容はモデリングや各種統計解析，描画処理など多岐に渡る．本研究では，arulesSequences パッケージを利用してコーディングパターンの検出を行った．arulesSequences はシーケンシャルパターンマイニングを行うためのパッケージで，Zaki の考案した SPADE アルゴリズムの実装を提供する．SPADE アルゴリズムを実行する R には 3.2 節で作成した系列データベースと最少支持を入力として渡す．本研究では最低 10 個の関数に出現するコーディングパターンのみを検出するように最小支持度を指定している．つまり，総数 100 個の関数が存在する場合，指定する最小支持度は 0.1 となる．

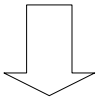
R によるパターンマイニングの結果として出力されるコーディングパターンを図 3.2 に示した．コーディングパターンは支持度とセットになって出力される．

```
StatusType
ActivateTask(TaskType TaskID) {

        LOG_ACTTSK_ENTER(TaskID);
        CHECK_DISABLEDINT();
                :
        x_nested_lock_os_int();
        if (p_tcb->tstat == SUSPENDED) {
                :
        }

  d_exit_no_errorhook:
        x_nested_unlock_os_int();
  exit_no_errorhook:
        LOG_ACTTSK_LEAVE(ercd);
        return(ercd);

#ifdef CFG_USE_ERRORHOOK
  exit_errorhook:
        x_nested_lock_os_int();
  d_exit_errorhook:
#ifdef CFG_USE_PARAMETERACCESS
        _errorhook_par1.tskid = TaskID;
#endif /* CFG_USE_PARAMETERACCESS */
        call_errorhook(ercd, OSServiceId_ActivateTask);
        goto d_exit_no_errorhook;
#endif
}
```



```
LOG_ACTTSK_ENTER()
CHECK_DISABLEDINT()
        :
x_nested_lock_os_int()
IF
make_active()
IF
        :
END-IF
d_exit_no_errorhook:
x_nested_unlock_os_int()
exit_no_errorhook:
LOG_ACTTSK_LEAVE()
return
exit_errorhook:
x_nested_lock_os_int()
d_exit_errorhook:
call_errorhook()
goto
d_exit_no_errorhook
```

図 5: プログラムから抽出される要素

```
…
15  4  1  CHECK_ID()
15  5  1  x_nested_lock_os_int()
15  6  1  incr_counter_action()
15  7  1  d_exit_no_errorhook:
15  8  1  x_nested_unlock_os_int()
15  9  1  exit_no_errorhook:
15 10  1  return
15 11  1  exit_errorhook:
15 12  1  x_nested_lock_os_int()
15 13  1  call_errorhook()
15 14  1  goto
15 15  1  d_exit_no_errorhook
16  1  1  LOG_GETCNT_ENTER()
16  2  1  CHECK_DISABLEDINT()
16  3  1  CHECK_CALLEVEL()
16  4  1  CHECK_ID()
…
```

図 6: マイニングアルゴリズムに渡す系列データベース

```
:
<{CHECK_ID()},{x_nested_unlock_os_int()}> 0.2469
<{CHECK_PARAM_POINTER()},{x_nested_unlock_os_int()}> 0.0987
<{CHECK_RESOURCE()},{x_nested_unlock_os_int()}> 0.0493
<{CHECK_VALUE()},{x_nested_unlock_os_int()}> 0.0617
<{d_exit_no_errorhook:},{x_nested_unlock_os_int()}> 0.2222
:
```

図 7: パターンマイニングアルゴリズムから出力される結果

3.3 マイニング結果の絞り込み

goto 文やラベル文を要素として追加したため，出力されるコーディングパターンは膨大な数となる．そこでステップ 3 としてまず，以下の 3 つの条件で結果の絞り込みを行った．

- 関数呼び出しの数
- 制御文の対応関係
- ラベルの対応関係

関数呼び出しの数

関数呼び出しの数が少ないコーディングパターンは開発者にとって重要ではない．そこで本研究では関数呼び出しの数が 1 つ以下のコーディングパターンを除外した．

制御文の対応関係.

制御構造から抽出される要素 IF と END-IF，LOOP と END-LOOP のそれぞれの対応関係が取れていないコーディングパターンは意味をなさないと考え除外した．

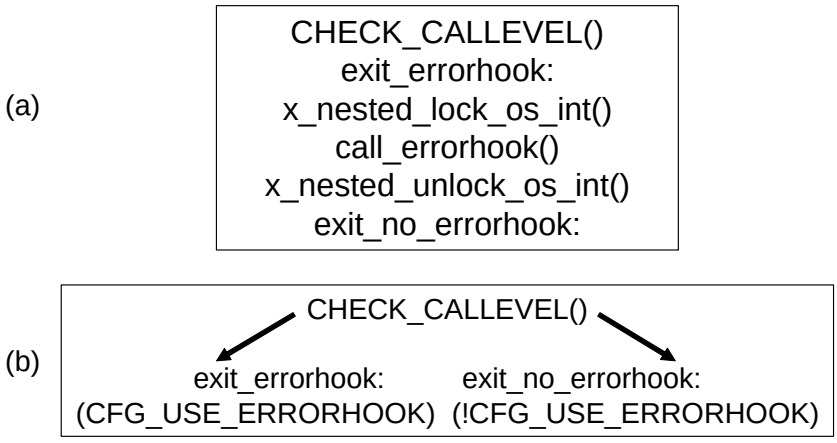


図 8: ステップ 2 から出力されたコーディングパターン (a) と goto 文を含むマクロ関数と対応するジャンプ先 (b)

ラベルの対応関係.

ラベルの対応関係が取れていないコーディングパターンは開発者にとって有益な情報とはいえないため除外した．具体的には，goto 文は存在するにもかかわらずジャンプ先のラベル文がコーディングパターンに含まれていない場合やその逆，ラベル文は存在するがどこからジャンプしてくるかを表す goto 文が存在しない場合である．

しかし，ステップ 2 で検出されたコーディングパターンを見るだけでは対応関係が取れているか分からない場合がある．図 3.3(a) はステップ 2 で検出されたコーディングパターンであり，一見すると対応関係が取れていないように見える．しかし実際はマクロ関数の中に goto 文が含まれており，マクロが展開されて初めてラベルの対応関係が取れているかどうかの判断が可能になる．そのため，ラベルの対応関係が取れているにもかかわらず除外されるコーディングパターンが出るので，本研究の手法ではマクロ関数が存在する場合は対象ソースファイルと同一ディレクトリに存在するヘッダファイルからマクロ関数の定義を調べて goto 文が含まれている場合はその情報も出力した．さらに，goto 文に続くラベルが同一ヘッダファイル内で再定義されている場合にも対応した．その結果が図 3.3(b) である．左側にはマクロ関数が，その横にジャンプ先が記されている．ジャンプ先のラベル名の後ろに付く括弧の中はラベルが`#ifdef` プリプロセッサ命令で囲まれていたことを表している．図 3.3(b) の例でいえば，CFG_USE_ERRORHOOK というマクロが定義されていた場合 exit_errorhook というラベルにジャンプするが，そのマクロが定義されていない場合は exit_no_errorhook にジャンプする．

SPADE アルゴリズムによるパターンマイニングでは図 3.3 のように，出力されるパターンはすべての部分パターンも含まれる．部分パターンは開発者にとって重要ではないので最長のパターン 1 つにまとめる操作を行った．これをパターンの極大化と呼ぶ．極大化が行われる条件は次の通りである．

パターンを α，β とする．
ただし，$\alpha = a_1, a_2, ..., a_n$,$\beta = b_1, _2, ..., b_m$, $n \leq m$
このとき， $a_1, a_2, ..., a_n$ がこの順番で $b_1, b_2, ..., b_m$ に含まれる場合かつ，
α，β の支持度が同じ場合に限り α は β に含まれ，極大化を行うことができる．

以上が本研究のコーディングパターン検出手法である．4 章ではこの手法の適用実験について述べる．

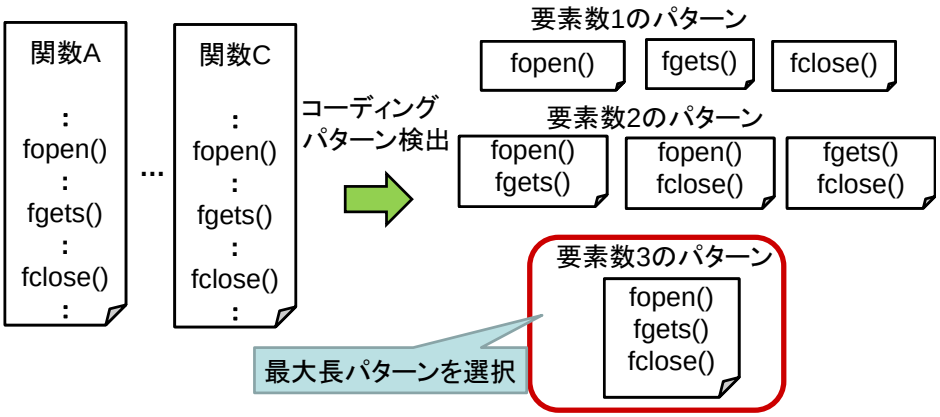


図 9: コーディングパターンの極大化

4　適用実験

4.1　実験データ

本研究では提案手法を TOPPERS/ATK2 に適用した．ATK2 は自動車制御用実時間 OS で，そのプログラムは多重割込みに関する処理など時間的制約の厳しい処理を含んでいるため，手法適用対象として相応しいと考えた．

ATK2 ではいくつかの機能セットが公開されておりそれぞれは複数のディレクトリ，プログラムから構成される．その中でも本研究では kernel ディレクトリ及びその中に含まれる C ファイルに着目した．利用した機能セットとその中の kernel ディレクトリに含まれる C ファイル及び C ファイル中の関数の数，コードの行数 (LOC) を表 4 に示した．

ATK2-SC1 はすべての機能セットのベースとなる機能セットで AUTOSAR OS 仕様の SC1 をベースとしている．ATK2-SC3 は SC1 をベースに，メモリ保護や OS アプリケーション等の機能を追加した機能セットである．また，MC と後ろに付く場合はベースとなっている機能セットをマルチコアに拡張したことを表している．すべての機能セットのベースとなっている ATK2-SC1 よりも機能拡張された機能セットのほうが複雑化しており最も複雑な機能セットは ATK2-SC3-MC である．以降，各機能セット名は先頭の ATK2 を省略する．

4.2　実行環境及び実行速度

本研究における実験環境は以下の通りである．

- OS：Microsoft Windows 8.1 Pro (64bit)
- CPU：Intel(R) Xeon(R) CPU E5-2690 0 @ 2.90GHz 2.90GHz (2 プロセッサ)
- RAM：256GB

手法の適用の際にかかった時間を表 5 に示す．3 章で述べた手法の 3 つのステップそれぞれに要した時間とその合計時間を表にした．

要素抽出や結果の絞り込みを行うステップに掛かる時間が全体の実行時間に対して占める割合は非常に小さい．さらに実行時間に関して，各ステップの中で最も支配的なステップはパターンマイニングを行うステップ 2 である．

また，ステップ 3 の絞り込みに関して，コーディングパターンがどれほど減ったかを表 6 に示した．この表から，約 99%のコーディングパターンが削減されていることが分かり，開発者が手作業でコーディングパターンを確認するコストを削減できたといえる．

表 4: 本研究の適用実験で用いた ATK2 の機能セットの kernel ディレクトリに含まれる C ファイルの数と関数の数及び LOC

機能セット名	C ファイルの数	関数の数	LOC
SC1	12	81	4,620
SC1-MC	17	131	7,726
SC3	16	135	7,698
SC3-MC	19	172	10,244

表 5: 各機能セットに対する適用実験に要した時間

	ステップ 1 (要素抽出)	ステップ 2 (パターンマイニング)	ステップ 3 (結果の絞り込み)	合計時間
SC1	0.094 秒	3.1 秒	2.1 秒	5.3 秒
SC3	0.078 秒	81 秒	1.1 秒	83 秒
SC1-MC	0.108 秒	4 時間 46 分	74 秒	4 時間 47 分
SC3-MC	0.078 秒	11 時間 1 分	148 秒	11 時間 3 分

表 6: 各絞り込みを適用後のコーディングパターンの数

	絞り込み前	関数呼び出しの数で絞り込み後	制御文の対応で絞り込み後	ラベルの対応で絞り込み後	極大化による絞り込み後	削減割合 (%)
SC1	408,613	392,648	258,441	9,026	29	99.993
SC3	157,148	150,259	97,909	3,639	64	99.959
SC1-MC	17,561,490	17,499,108	17,204,504	154,253	240	99.999
SC3-MC	34,279,185	34,246,413	34,138,960	274,025	511	99.999

4.3 検出されたコーディングパターン

各機能セットから手法を用いて検出したコーディングパターンのうち，支持度の上位 5 つを表 7 から表 10 示した．また，表 4.3 から表 4.3 は長さでソートした中から選択された上位 5 つを示している．各表は 5 つのコーディングパターンとその支持度，及び goto 文を含むマクロ関数のジャンプ先を示した表から構成されている．さらに，支持度の括弧内の数字はそ

表 7: ATK2-SC1 から検出されたコーディングパターンを支持度でソートした中の上位 5 つ及び goto 文を含むマクロ関数のジャンプ先

マクロ関数	ジャンプ先
CHECK_CALLEVEL()	exit_errorhook (CFG_USE_ERRORHOOK) exit_no_errorhook (!CFG_USE_ERRORHOOK)
コーディングパターン 1	コーディングパターン 2
x_nested_lock_os_int() x_nested_unlock_os_int() 0.419753(34)	x_nested_lock_os_int() call_errorhook() 0.407407(33)
コーディングパターン 3	コーディングパターン 4
return x_nested_lock_os_int() call_errorhook() 0.358025(29)	CHECK_CALLEVEL() exit_no_errorhook: return exit_errorhook: x_nested_lock_os_int() call_errorhokk() 0.283951(23)
コーディングパターン 5	
x_nested_lock_os_int() x_nested_lock_os_int() 0.271605(22)	

のコーディングパターンがいくつの関数に存在するかを示している.

表 8: ATK2-SC3 から検出されたコーディングパターンを支持度でソートした中の上位 5 つ及び goto 文を含むマクロ関数のジャンプ先

マクロ関数	ジャンプ先
コーディングパターン 1	コーディングパターン 2
x_nested_lock_os_int() x_nested_unlock_os_int() 0.447761(60)	x_nested_lock_os_int() call_errorhook() 0.440299(59)
コーディングパターン 3	コーディングパターン 4
return x_nested_lock_os_int() call_errorhook() 0.402985(54)	x_nested_lock_os_int() call_errorhokk() x_nested_unlock_os_int() 0.276119(37)
コーディングパターン 5	
return x_nested_lock_os_int() call_errorhook() x_nested_unlock_os_int() 0.246269(33)	

表 9: ATK2-SC1-MC から検出されたコーディングパターンを支持度でソートした中の上位 5 つ及び goto 文を含むマクロ関数のジャンプ先

マクロ関数	ジャンプ先
コーディングパターン 1	コーディングパターン 2
get_my_p_ccb() call_errorhook() 0.412214(54)	x_nested_lock_os_int() x_nested_unlock_os_int() 0.396947(52)
コーディングパターン 3	コーディングパターン 4
get_my_p_ccb() x_nested_unlock_os_int() 0.381679(50)	x_nested_lock_os_int() call_errorhokk() 0.381679(50)
コーディングパターン 5	
return x_nested_lock_os_int() call_errorhook() 0.335878(44)	

表 10: ATK2-SC3-MC から検出されたコーディングパターンを支持度でソートした中の上位 5 つ及び goto 文を含むマクロ関数のジャンプ先

マクロ関数	ジャンプ先
コーディングパターン 1	コーディングパターン 2
get_my_p_ccb() call_errorhook() 0.412791(71)	x_nested_lock_os_int() x_nested_unlock_os_int() 0.395349(68)
コーディングパターン 3	コーディングパターン 4
x_nested_lock_os_int() call_errorhokk() 0.383721(66)	get_my_p_ccb() x_nested_unlock_os_int() 0.383721(66)
コーディングパターン 5	
return x_nested_lock_os_int() call_errorhook() 0.343023(59)	

表 11: ATK2-SC1 から検出されたコーディングパターンを長さでソートした中の上位 5 つ及び goto 文を含むマクロ関数のジャンプ先

マクロ関数	ジャンプ先
CHECK_DISABLEDINT()	exit_errorhook (CFG_USE_ERRORHOOK) exit_no_errorhook (!CFG_USE_ERRORHOOK)
CHECK_CALLEVEL()	exit_errorhook (CFG_USE_ERRORHOOK) exit_no_errorhook (!CFG_USE_ERRORHOOK)
CHECK_ID()	exit_errorhook (CFG_USE_ERRORHOOK) exit_no_errorhook (!CFG_USE_ERRORHOOK)
コーディングパターン 1	コーディングパターン 2
CHECK_DISABLEDINT() CHECK_CALLEVEL() CHECK_ID() x_nested_lock_os_int() d_exit_no_errorhook: x_nested_unlock_os_int() exit_no_errorhook: return exit_errorhook: x_nested_lock_os_int() call_errorhook() goto d_exit_no_errorhook 0.17284(14)	CHECK_DISABLEDINT() CHECK_CALLEVEL() x_nested_lock_os_int() d_exit_no_errorhook: x_nested_unlock_os_int() exit_no_errorhook: return exit_errorhook: x_nested_lock_os_int() call_errorhook() goto d_exit_no_errorhook 0.209877(17)

コーディングパターン 3	コーディングパターン 4
CHECK_DISABLEDINT() CHECK_CALLEVEL() CHECK_ID() x_nested_lock_os_int() x_nested_unlock_os_int() exit_no_errorhook: return exit_errorhook: x_nested_lock_os_int() call_errorhook() 0.197531(16)	CHECK_DISABLEDINT() CHECK_CALLEVEL() IF END-IF exit_no_errorhook: return exit_errorhook: x_nested_lock_os_int() call_errorhook() 0.135802(11)
コーディングパターン 5	
CHECK_DISABLEDINT() CHECK_CALLEVEL() x_nested_lock_os_int() x_nested_unlock_os_int() exit_no_errorhook: return exit_errorhook: x_nested_lock_os_int() call_errorhook() 0.234568(19)	

表 12: ATK2-SC3 から検出されたコーディングパターンを長さでソートした中の上位 5 つ及び goto 文を含むマクロ関数のジャンプ先

マクロ関数	ジャンプ先
CHECK_DISABLEDINT()	exit_errorhook (CFG_USE_ERRORHOOK) exit_no_errorhook (!CFG_USE_ERRORHOOK)
CHECK_CALLEVEL()	exit_errorhook (CFG_USE_ERRORHOOK) exit_no_errorhook (!CFG_USE_ERRORHOOK)
CHECK_ID()	exit_errorhook (CFG_USE_ERRORHOOK) exit_no_errorhook (!CFG_USE_ERRORHOOK)
CHECK_RIGHT()	exit_errorhook (CFG_USE_ERRORHOOK) exit_no_errorhook (!CFG_USE_ERRORHOOK)
D_CHECK_ACCESS()	d_exit_errorhook (CFG_USE_ERRORHOOK) d_exit_no_errorhook (!CFG_USE_ERRORHOOK)
コーディングパターン 1	コーディングパターン 2
CHECK_DISABLEDINT()	CHECK_CALLEVEL()
CHECK_CALLEVEL()	CHECK_ID()
CHECK_ID()	CHECK_RIGHT()
CHECK_RIGHT()	x_nested_lock_os_int()
x_nested_lock_os_int()	D_CHECK_ACCESS()
D_CHECK_ACCESS()	d_exit_no_errorhook:
d_exit_no_errorhook:	x_nested_unlock_os_int()
x_nested_unlock_os_int()	exit_no_errorhook:
exit_no_errorhook:	exit_errorhook:
return	x_nested_lock_os_int()
exit_errorhook:	d_exit_errorhook:
x_nested_lock_os_int()	call_errorhook()
d_exit_errorhook:	goto
call_errorhook()	d_exit_no_errorhook
goto	
d_exit_no_errorhook	
0.104478(14)	0.11194(15)

コーディングパターン 3	コーディングパターン 4
CHECK_DISABLEDINT() CHECK_CALLEVEL() CHECK_ID() CHECK_RIGHT() x_nested_lock_os_int() d_exit_no_errorhook: x_nested_unlock_os_int() exit_no_errorhook: return exit_errorhook: x_nested_lock_os_int() call_errorhook() goto d_exit_no_errorhook 0.11194(15)	CHECK_CALLEVEL() CHECK_ID() CHECK_RIGHT() x_nested_lock_os_int() d_exit_no_errorhook: x_nested_unlock_os_int() exit_no_errorhook: return exit_errorhook: x_nested_lock_os_int() call_errorhook() goto d_exit_no_errorhook 0.11194(15)
コーディングパターン 5	
CHECK_DISABLEDINT() CHECK_CALLEVEL() CHECK_ID() x_nested_lock_os_int() d_exit_no_errorhook: x_nested_unlock_os_int() exit_no_errorhook: return exit_errorhook: x_nested_lock_os_int() call_errorhook() goto d_exit_no_errorhook: 0.11194(15)	

表 13: ATK2-SC1-MC から検出されたコーディングパターンを長さでソートした中の上位 5 つ及び goto 文を含むマクロ関数のジャンプ先

マクロ関数	ジャンプ先
CHECK_DISABLEDINT()	exit_errorhook (CFG_USE_ERRORHOOK) exit_no_errorhook (!CFG_USE_ERRORHOOK)
CHECK_CALLEVEL()	exit_errorhook (CFG_USE_ERRORHOOK) exit_no_errorhook (!CFG_USE_ERRORHOOK)
CHECK_ID()	exit_errorhook (CFG_USE_ERRORHOOK) exit_no_errorhook (!CFG_USE_ERRORHOOK)
CHECK_CORE()	exit_errorhook (CFG_USE_ERRORHOOK) exit_no_errorhook (!CFG_USE_ERRORHOOK)
コーディングパターン 1	コーディングパターン 2
CHECK_DISABLEDINT()	CHECK_DISABLEDINT()
CHECK_CALLEVEL()	CHECK_CALLEVEL()
CHECK_ID()	CHECK_ID()
CHECK_CORE()	CHECK_CORE()
x_nested_lock_os_int()	get_p_ccb()
x_nested_unlock_os_int()	x_nested_lock_os_int()
exit_no_errorhook:	acquire_cnt_lock()
return	release_cnt_lock()
goto	x_nested_unlock_os_int()
errorhook_start	exit_no_errorhook:
exit_errorhook:	return
x_nested_lock_os_int()	exit_errorhook:
errorhook_start:	x_nested_lock_os_int()
get_my_p_ccb()	get_my_p_ccb()
call_errorhook()	call_errorhook()
x_nested_unlock_os_int()	x_nested_unlock_os_int()
goto	goto
exit_no_errorhook	exit_no_errorhook
0.083969(11)	0.076336(10)

コーディングパターン 3	コーディングパターン 4
CHECK_DISABLEDINT() CHECK_CALLEVEL() CHECK_ID() CHECK_CORE() x_nested_lock_os_int() x_nested_unlock_os_int() exit_no_errorhook: return exit_errorhook: x_nested_lock_os_int() get_my_p_ccb() get_my_p_ccb() call_errorhook() x_nested_unlock_os_int() goto exit_no_errorhook 0.076336(10)	CHECK_DISABLEDINT() CHECK_CALLEVEL() CHECK_ID() CHECK_CORE() x_nested_lock_os_int() x_nested_unlock_os_int() exit_no_errorhook: return exit_errorhook: x_nested_lock_os_int() get_my_p_ccb() call_errorhook() x_nested_unlock_os_int() goto exit_no_errorhook 0.099237(13)
コーディングパターン 5	
CHECK_CALLEVEL() CHECK_ID() CHECK_CORE() get_p_ccb() x_nested_lock_os_int() x_nested_unlock_os_int() exit_no_errorhook: return exit_errorhook: x_nested_lock_os_int() get_my_p_ccb() call_errorhook() x_nested_unlock_os_int() goto exit_no_errorhook 0.083969(11)	

表 14: ATK2-SC3-MC から検出されたコーディングパターンを長さでソートした中の上位 5 つ及び goto 文を含むマクロ関数のジャンプ先

マクロ関数	ジャンプ先
CHECK_DISABLEDINT()	exit_errorhook (CFG_USE_ERRORHOOK) exit_no_errorhook (!CFG_USE_ERRORHOOK)
CHECK_CALLEVEL()	exit_errorhook (CFG_USE_ERRORHOOK) exit_no_errorhook (!CFG_USE_ERRORHOOK)
CHECK_ID()	exit_errorhook (CFG_USE_ERRORHOOK) exit_no_errorhook (!CFG_USE_ERRORHOOK)
CHECK_CORE()	exit_errorhook (CFG_USE_ERRORHOOK) exit_no_errorhook (!CFG_USE_ERRORHOOK)
CHECK_RIGHT()	exit_errorhook (CFG_USE_ERRORHOOK) exit_no_errorhook (!CFG_USE_ERRORHOOK)
コーディングパターン 1	コーディングパターン 2
CHECK_DISABLEDINT()	CHECK_DISABLEDINT()
CHECK_CALLEVEL()	CHECK_CALLEVEL()
CHECK_ID()	CHECK_ID()
CHECK_CORE()	CHECK_CORE()
CHECK_RIGHT()	CHECK_RIGHT()
get_p_ccb()	x_nested_lock_os_int()
x_nested_lock_os_int()	x_nested_unlock_os_int()
acquire_cnt_lock()	exit_no_errorhook:
release_cnt_lock()	return
x_nested_unlock_os_int()	goto
exit_no_errorhook:	errorhook_start
return	exit_errorhook:
release_cnt_lock()	x_nested_lock_os_int()
goto	errorhook_start:
errorhook_start	get_my_p_ccb()
exit_errorhook:	get_my_p_ccb()
x_nested_lock_os_int()	call_errorhook()
errorhook_start:	x_nested_unlock_os_int()
get_my_p_ccb()	goto
call_errorhook()	exit_no_errorhook
x_nested_unlock_os_int()	
goto	
exit_no_errorhook	
0.063953(11)	0.063953(11)

コーディングパターン 3	コーディングパターン 4
CHECK_DISABLEDINT() CHECK_CALLEVEL() CHECK_ID() CHECK_CORE() CHECK_RIGHT() x_nested_lock_os_int() x_nested_unlock_os_int() exit_no_errorhook: return goto errorhook_start exit_errorhook: x_nested_lock_os_int() errorhook_start: get_my_p_ccb() call_errorhook() x_nested_unlock_os_int() goto exit_no_errorhook 0.081395(14)	CHECK_CALLEVEL() CHECK_ID() CHECK_CORE() CHECK_RIGHT() x_nested_lock_os_int() x_nested_unlock_os_int() exit_no_errorhook: return goto errorhook_start exit_errorhook: x_nested_lock_os_int() errorhook_start: get_my_p_ccb() get_my_p_ccb() call_errorhook() x_nested_unlock_os_int() goto exit_no_errorhook 0.069767(12)
コーディングパターン 5	
CHECK_CALLEVEL() CHECK_ID() CHECK_CORE() CHECK_RIGHT() x_nested_lock_os_int() x_nested_unlock_os_int() exit_no_errorhook: return goto errorhook_start exit_errorhook: x_nested_lock_os_int() errorhook_start: get_my_p_ccb() call_errorhook() x_nested_unlock_os_int() goto exit_no_errorhook 0.087209(15)	

4.4 コーディングパターンの評価

本研究では，各機能セットから得られた計 40 個のコーディングパターンに対して評価を行った．具体的には，外部仕様書や開発文書に含まれている情報がコーディングパターンに反映されているかを確認しその数を算出した．

開発文書には以下の記述がある [16]．

```
-------------------------------------------------
6.3.3 OS 割込み禁止状態の管理

ターゲット依存部は，OS 割込み禁止状態へ遷移する機能，OS 割込み禁止解除状
態へ遷移する機能を提供する．

(6-3-3-1) void x_nested_lock_os_int(void)
…
OS 割込み禁止解除状態から，OS 割込み禁止状態へ遷移させる関数．
…
(6-3-3-3) void x_nested_unlock_os_int(void)
…
OS 割込み禁止状態から，OS 割込み禁止解除状態へ遷移させる関数．これらの関
数は，OS 割込み禁止解除状態で呼び出されることはなく，呼び出された場合の
動作は保証する必要がない．
-------------------------------------------------
```

この記述から，禁止状態へ遷移したあとは必ず禁止状態を解除して他からの割込みが行われるようにする必要があると読み取ることができる．したがって x_nested_lock_os_int() と x_nested_unlock_os_int() の関数は対応関係にあると分かる．この仕様書の情報を反映しているコーディングパターンは例えば表 4.3 のコーディングパターン 1 などが該当する．逆に，表 10 のコーディングパターン 3 などはどちらか片方の関数しか含まれていないため仕様書の情報を反映しているとはいえない．さらに，同じく表 10 のコーディングパターンのようにどちらの関数も含んでいない場合がある．しかし，これらのコーディングパターンは仕様書の情報を反映した，意味のあるコーディングパターンとはいい難いと判断した．

また，外部仕様書にはソースファイル中の各関数の機能仕様が記述されている [16]．この機能仕様から，各関数がどのようなエラーコードを返すかが読み取れる．そしてこのエラーコードはコーディングパターン中の CHECK で始まるエラーチェック関数群の返り値である

ため，機能仕様のエラーコード一覧を見るとその関数が実装しているエラーチェック関数が分かる．したがって，各コーディングパターンが属する関数を調べてその関数が実装しているエラーチェック関数がコーディングパターンに反映されているか確認することができる．

その結果，コーディングパターン中のエラーチェック関数群は 1，2 個以内の欠落に収まっていることが分かった．このことから各コーディングパターン中のエラーチェック関数群は仕様書の情報を反映していると判断した．同時に，エラーチェック関数の中に含まれる goto 文に対応するラベルを含む処理も仕様書を反映していると判断した．

以上を踏まえると全 40 個のコーディングパターンの内，25 個のコーディングパターンが仕様書の情報を反映した意味のあるコーディングパターンであると確認された．このことから，今後これらのコーディングパターンを再利用やバグ検出に利用できると判断した．

4.5 考察

実行速度の改善

実行時間に関して最も支配的なステップは R によるパターンマイニングを行うステップであり，SC3-MC への適用で 11 時間掛かる．そのほかのステップに掛かる時間はステップ 2 と比較すると非常に少ないため，全体の実行時間を改善する場合，パターンマイニングに掛かる時間を短縮する必要がある．考えられる方法としてはパターンマイニングを並列化することである．パターンマイニングでは入力の系列データベースに基づいて各系列の組み合わせを見ていくため処理に時間が掛かる．この部分を並列化することで実行時間減少が期待できる．ただし，この手法が 1 つのプロダクトに対して適用されるのは一度のみであることを踏まえると実行速度の改善問題の優先度は低い．

既存研究の手法と比較したときの絞り込みの有効性

ステップ 3 の絞り込みに関して，本研究の手法では関数呼び出しの数，制御構造の対応関係，ラベルの対応関係で不要と思われるパターンを除外した．さらに，極大化を行い部分パターンを一つにまとめた．その結果，約 99%のコーディングパターンの除外に成功しており，表 6 の通りラベルの対応関係による絞り込みでコーディングパターンの数は激減している．一方既存研究の手法では，制御構造要素が 70%を占める場合や制御構造の対応が取れていないパターンを除外した上でパターンのグループ化を行っている．既存手法をそのまま用いていた場合，ラベルの対応関係を見ないことになるため本研究の結果よりも絞り込みの割合は減少すると思われる．したがって，絞り込みによるコーディングパターンの削減という観点から，既存手法よりも優れた絞り込みを行えたと考察できる．

絞り込み手法の改善

絞り込みに関して改善するべき点がある．本手法ではコーディングパターンの支持度及び長さの上位 5 つを選択し結果としている．しかし，結果をみると類似していると思われるコーディングパターンがいくつも表れている．例えば表 7 におけるコーディングパターン 2 と 3 は return があるかないかの違いしかない．また，表 4.3 におけるコーディングパターンの結果を見ても，そのほとんどが CHECK で始まるマクロ関数と lock，unlock を行う関数列そしてマクロ関数に含まれる goto 文に対応するラベルで構成されている．さらに，各コーディングパターンが属する関数を調べるとそのほとんどが一致する．このことから，本研究の手法で選択したコーディングパターンは完全に独立したコーディングパターンではないと考えられる．したがって絞り込み手法の改善点としてコーディングパターンの極大化を行った後に，類似したコーディングパターンを 1 つにすることが挙げられる．Xin らは類似した頻出パターンは冗長性を持つとし，頻出パターンの集合の中から冗長性の低い，重要な頻出パターンを見つける手法を提案している [17]．頻出パターンとシーケンシャルパターンという違いはあるが，本研究の絞り込み手法の改善に役立つと思われる．

絞り込みの順番

本研究の手法では絞り込みの順番を関数呼び出しの数，各対応関係，極大化としている．コーディングパターンの極大化ではあるコーディングパターンがすべての極大化済みのコーディングパターンに含まれているかどうかを調べる．関数呼び出しの数や対応関係を調べる時間と比較すると極大化にはより多くの時間が掛かることが予想される．したがってもし極大化を最初に行っていた場合，千万規模のコーディングパターンに対して処理を行うため表 5 で示した実行速度より遅くなると考えられる．

評価に対する考察

評価において 40 個中 25 個のコーディングパターンを，仕様書の情報を反映した意味のあるコーディングパターンであることを確認した．残りの 15 個は関数の対応関係が取れていないことが理由で除外された．これらは類似したコーディングパターンであるため，4 章で述べた類似パターンをまとめる手法を適用することでこれらは 1 つにまとめられると思われる．その結果，除外されるコーディングパターンの数の減少が期待できる．

また，25 個のコーディングパターンはエラー処理を行うためのラベルを含んでいる場合が多い．このことから，既存手法をそのまま適用していた場合，これらのコーディングパターンは検出されなかったと思われる．

5　まとめと今後の課題

本研究では，再利用やバグ検出を目的として C 言語で記述された実時間プログラムからコーディングパターンの検出を行った．既存手法のままでは C 言語で記述された実時間プログラムではジャンプ命令やプリプロセッサ命令が多用されることに対応できないため，本研究の手法では関数呼び出し，制御構造に加えて goto 文や return 文，ラベルを要素として抽出した．また，検出されたコーディングパターンの絞り込みを行い開発者が手作業でコーディングパターンを確認するコストを削減できた．そして評価では選択した 40 個中 25 個のコーディングパターンが仕様書の情報を反映した意味のあるコーディングパターンであることを確認した．その結果，コーディングパターンを再利用やバグ検出に用いることができると判断した．

今後の課題として，まず絞り込みの手法改善を行う予定である．類似したコーディングパターンを 1 つにまとめる必要があり，そのために Xin らの冗長性が低い頻出パターンを見つける手法を適用する予定である [17]．また，実際にコーディングパターンをどう再利用するか，コーディングパターンを使ってどのようにバグ検出を行うかという課題についても検討していきたい．

謝辞

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授には常に適切な御指導及び御助言を賜りました．井上教授の御指導のおかげで本論文を完成させることができました．井上 教授に深く感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下誠 准教授には研究に関する適切な御指導及び御助言を賜りました．また，普段の生活においても御助言を賜りました．有意義な研究生活を送ることができたのは松下 准教授のおかげであると，心より深く感謝いたします．

本研究で紹介した既存研究は 大阪大学大学院情報科学研究科コンピュータサイエンス専攻 石尾隆 助教がなされた研究であり，既存研究をもとに行った本研究に関する適切な御指導及び御助言を賜りました．石尾 助教に深く感謝いたします．

名古屋大学大学院情報科学研究科附属組込みシステム研究センター / 情報システム学専攻 吉田則裕 准教授には本研究において直接御指導賜りました．本研究分野において不慣れな私でしたが，終始多くの御指導を頂いたおかげで本論文を執筆することができました．吉田 准教授に心より深く感謝いたします．

大阪大学大学院情報科学研究科 春名修介 特任教授には適切な御指導及び御助言を賜りました．本研究に関する問題点の指摘など，多くの御助言を頂いた春名 特任教授に深く感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 崔恩瀞 氏，佐野 真夢 氏には，研究の補助及び本論文の推敲など多くの御協力をいただきました．また，学生生活のおいても私を御気に掛けて下さりました．私が本論文を完成させることができたこと及び充実した生活を送ることができたのは両氏のおかげであると，心より深く感謝いたします．

最後に，御指導，御助言を頂き私を支えてくださった，大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様に深く感謝いたします．

参考文献

[1] 石尾隆, 伊達浩典, 三宅達也, 井上克郎. シーケンシャルパターンマイニングを用いたコーディングパターン抽出. 情報処理学会論文誌, Vol. 50, No. 2, pp. 860–871, 2009.

[2] Stankovic and J.A. Misconceptions about real-time computing. *IEEE Computer*, Vol. 21, No. 10, pp. 10–19, 1988.

[3] P. Mohagheghi, R. Conradi, OleM. Kili, H. Schwarz. An empirical study of software reuse vs. defect-density and stability. *Proceedings of the 26th International Conference on Software Engineering(ICSE'04)*, pp. 282–291, 2004.

[4] VDC Research. What languages do you use to develop software? `http://blog.vdcresearch.com/embedded_sw/2010/09/what-languages-do-you-use-to-develop-software.html`.

[5] 山本哲男, 吉田則裕, 肥後芳樹. ソースコードコーパスを利用したシームレスなソースコード再利用法. 情報処理学会論文誌, Vol. 53, No. 2, pp. 644–652, 2012.

[6] M. Fowler. Refactoring : improving the design of existing code. Addison Wesley, 1999.

[7] 後藤祥, 吉田則裕, 井岡正和, 井上克郎. 差分を含む類似メソッドの集約支援ツール. 情報処理学会論文誌, Vol. 54, No. 2, pp. 922–932, 2013.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingitier, J. Irwin. Aspect-oriented programming. *Proceedings of 11th European Conference on Object-Oriented Programming(ECOOP'97)*, pp. 220–242, 1997.

[9] 金明哲. R によるデータサイエンス. 森北出版株式会社, 2007.

[10] J. Srivastava, R. Cooley, M. Deshpande, P.Tan. Web usage mining: discovery and applications of usage patterns from web data. *ACM SIGKDD Explorations Newsletter*, Vol. 1, No. 2, pp. 12–23, 2000.

[11] 早川潤一. 頻出系列パターンマイニング手法を用いた web 利用パターン発見に関する研究. 名古屋工業大学大学院情報工学専攻修士論文, 2006.

[12] H. Kagdi, M. L.Collard, J. I.Maletic. Comparing approaches to mining source code for call-Usage patterns. *Proceedings of 4th International Workshop on Mining Software Repositories(MSR'07)*, pp. 123–130, 2007.

[13] Mohammed J. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, Vol. 42, pp. 31–60, 2001.

[14] 高田広章. 組込みシステム開発技術の現状と展望. 情報処理学会論文誌, Vol. 42, No. 4, pp. 930–938, 2001.

[15] The R Project for Statistical Computing. `http://www.r-project.org/`.

[16] TOPPERS プロジェクト/ATK2. `https://www.toppers.jp/atk2.html`.

[17] D. Xin, H. Cheng, X. Yan, J. Han. Extracting redundancy-aware top-k patterns. *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining(KDD'06)*, pp. 444–453, 2006.