社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

パターンマイニング技術を用いたC言語プログラムからのコーディングパターン抽出

中村　勇太† 　崔　恩瀞†† 　吉田　則裕††† 　春名　修介† 　井上　克郎†

† 大阪大学 大学院情報科学研究科
〒 565-0871 大阪府吹田市山田丘 1 番 5 号
†† 大阪大学 大学院国際公共政策研究科
〒 560-0043 大阪府豊中市待兼山町 1 番 31 号
††† 名古屋大学 大学院情報科学研究科
〒 464-8601 名古屋市千種区不老町
E-mail: †{n-yuuta,haruna,inoue}@ist.osaka-u.ac.jp, ††ejchoi@osipp.osaka-u.ac.jp, †††yoshida@ertl.jp

あらまし　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．コーディングパターンは守らなければいけないルールを表しているため，そのルール通りにプログラムが記述されているか検査することでバグの検出ができる．本研究では，組込みシステム開発など高い信頼性が要求されるドメインで頻繁に用いられるC言語を対象として，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を抽出する手法を提案する．適用実験では，提案手法を用いて抽出したコーディングパターンが対象ソフトウェアの仕様の一部を表していることを確認した．

キーワード　コーディングパターン，パターンマイニング技術，信頼性，組込みプログラム

Coding Pattern Detection for C Programs Using Pattern Mining Technique

Yuta NAKAMURA†, Eunjong CHOI††, Norihiro YOSHIDA†††, Syusuke HARUNA†, and Katsuro INOUE†

† Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871 Japan
†† Osaka School of International Public Policy, Osaka University
1-31 Machikaneyama, Toyonaka Osaka, 560-0043, Japan
††† Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, Nagoya, 464-8601, Japan
E-mail: †{n-yuuta,haruna,inoue}@ist.osaka-u.ac.jp, ††ejchoi@osipp.osaka-u.ac.jp, †††yoshida@ertl.jp

Abstract　Coding pattern is an idiomatic code fragment that distributes in modules. It can be used for finding bugs because it implies rules that programmers should follow. In this paper, we propose a sequential pattern mining approach to detect coding pattern from programs written in C language that is commonly used in reliable software engineering. In the experiment, we confirmed that the detected coding patterns corresponded to a part of the specification of each target software system.

Key words　Coding pattern, Pattern mining approach, Reliability, Embedded program

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言語プログラムにおいて，ファイルのオープン `fopen()` を行った後にファイルをクローズ `fclose()` するコードが複数箇所に記述されている場合，この

`fopen()` と `fclose()` の関数呼び出しの対応はコーディングパターンといえる．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である．

コーディングパターンは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．fopen() と fclose() のコーディングパターンを例に挙げると，このコーディングパターンは fopen() を呼び出した後は必ず fclose() を呼び出す必要があるというルールを表している．そしてプログラム中に fclose() が呼び出されていない箇所がある場合は，ルールを無視したバグを含んだプログラムとなる．ゆえに，プログラムから抽出したコーディングパターンに基づいて，ルールに則った記述がされているか調べることでプログラム中のバグを検出することが可能である．

これまでにコーディングパターンの抽出やそれを使ったバグの検出に関する研究が行われてきた．石尾らは Java プログラムに対してコーディングパターン抽出を行った [1]．彼らは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，その要素列を基に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抽出を行った．シーケンシャルパターンマイニングとは順番性を持つパターンを見つけるための技術である [2]．また，Li らはコピーアンドペーストによって生成されるコーディングパターンの抽出を行う手法を提案した [3]．彼らの手法では，関数呼び出しや変数の参照からなる要素列を基に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抽出を行っている．さらに，手法を C 言語で記述された Linux カーネルソースプログラムに適用し，実際にバグを検出している．

本研究では，組込みシステム開発など高い信頼性が要求されるドメインで頻繁に用いられる C 言語を対象として，コーディングパターンの抽出を行う．例えば，組込みプログラムでは結果の正確さだけでなく決められた時間内に計算を完了させることが求められる実時間性 [4] を持っている．自動車のエンジン制御部や航空制御システムに代表される実時間性を持つ組込みプログラムは不具合の影響が大きいため，信頼性を高く保つ必要がある．したがって，コーディングパターンに基づくバグ検出が有効だと思われる．一方，組込みプログラムではコストダウンの需要によりリソースに厳しい制約が掛かる（低速度のプロセッサ，少ないメモリ) [5]．この問題に対処するために，プリプロセッサ命令を用いて実行コードを静的に切り替え，メモリ使用量を削減するという工夫が行われる．また，厳しい時間的制約が掛かることも多いためジャンプ命令が多用される．しかし，石尾らや Li らの研究ではプリプロセッサ命令やジャンプ命令を考慮していない．したがって彼らの手法を用いて組込みプログラムから有効なコーディングパターンを抽出し，バグ検出のために利用していくことは困難である．

そこで本研究では，ジャンプ命令やプリプロセッサ命令が使われるという組込みプログラムの特性を踏まえてコーディングパターンの抽出を行う手法を提案する．対象言語は高信頼ソフトウェア開発で頻繁に用いられる C 言語とした [6]．本手法ではまず，プログラムから関数呼び出し，制御構造，goto 文，ラベル文そして return 文からなる要素列を抽出する．そして石尾らの研究と同じくシーケンシャルパターンマイニングを適用してコーディングパターンの抽出を行う．最後に，膨大な量のコーディングパターンを絞り込む．また，この手法を組込みプログラムに適用し，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 章ではコーディングパターンの抽出に用いるパターンマイニング技術について説明し，3 章では本研究が提案するコーディングパターン抽出の手法を述べる．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．

表 1　シーケンシャルパターンマイニングの入力となる系列データベース

系列番号 (SID)	系列
1	A B C D
2	E A B F
3	A B F C
4	D G H

2. パターンマイニング技術

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 [7]，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の 1 つとしてシーケンシャルパターンマイニングがある．これは検出されるパターンの順番性を考慮したマイニング技術で，例えば 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 [8], [9]．

シーケンシャルパターンマイニングの他にも，順番性を考慮しないマイニング技術として頻出パターンマイニングがある．しかし Kagdi らは，順番性を持たないパターンよりも順番性を考慮したパターンのほうが欠陥検出率は優れていると述べている [10]．よってここでは，シーケンシャルパターンマイニングに焦点を当てて説明する．

2.1 シーケンシャルパターンマイニング

シーケンシャルパターンマイニングは入力として表 1 に示されるような系列データベースと支持度を取る．系列データベースは複数のアイテムから構成される系列及び系列番号を持つ．また，支持度とは抽出される各系列パターンが系列データベース中に出現する頻度を表す値である．例えば表 1 では系列パターン `A B` は系列番号 1，2，3 の計 3 つの系列に含まれるためその支持度は 3 となる．シーケンシャルパターンマイニングとは与えられた系列データベースに対して，指定した支持度以上で出現する頻出系列パターンを見つける技術である．

2.2 SPADE アルゴリズム

この節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) アルゴリズムについて説明する [2]．SPADE は入力として表 1 のデータベースに各アイ

表 2　SPADE アルゴリズムの入力となる系列データベース

SID(系列番号)	EID(出現順序番号)	アイテム
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

表 3　表 2 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 4　表 3 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

テムの出現順序番号 (EID) を加えた，表 2 に示される系列データベースと支持度を受け取る．SPADE は系列データベースから表 3 のように各アイテムごとに系列番号 (SID) と出現順序番号 (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．結合は同じ SID を持ち，かつ EID が昇順になるよう行われる．また，結合した際に支持度を計算し，入力された支持度以下であればそれ以上の結合は打ち切る．これを繰り返して全頻出系列パターンを抽出していく．例として表 4 に，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示した．

3. コーディングパターン抽出手法

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

手法はコーディングパターン構成要素の抽出 (ステップ 1)，コーディングパターン抽出 (ステップ 2)，マイニング結果の絞り込み (ステップ 3) の 3 つのステップから成る．以降，各ステップについて説明していく．

3.1　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，プログラムから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する．

- 関数呼び出し
- 制御構造
- goto 文
- ラベル文
- return 文

関数呼び出し．関数呼び出しについてはその関数名のみを要素とする．引数については，関数名の列だけでプログラムのルールを表すことができると判断し考慮していない．また，C 言語プログラムではマクロ関数呼び出しが存在する．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ため，展開後のコードではなくマクロ関数名をそのまま抽出する方が処理内容を把握しやすい．そこで本手法ではマクロ関数呼び出しについても同様にその名前のみを要素として抽出している．

制御構造．本手法では制御構造として if 文，while 文，for 文を考えている．これらの制御構造からどのように要素が抽出されるかを図 1 に示した．

if 文は IF，ELSE，END-IF を基本構造としており，条件式中で関数呼び出しが用いられていればその関数名を IF の直前の要素として抽出する．さらに，if 文中に現れる他の要素はすべて IF と END-IF 内に挟まれる形で抽出される (図 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，変更式中の関数名は適切な位置の要素として抽出される（図 1(b)(c)）．これらの位置は各命令が実行される順番に依存している．また，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．

ジャンプ命令．また，組込みプログラムでジャンプ命令が多用されることを考え本手法では goto 文を要素として抽出する．同様に，プログラム中に存在するラベル文も要素とする．さらに，ジャンプ命令によってプログラム上を行き来することになるため，処理の流れを明確にする必要があると考え return 文も要素として抽出している．

3.2　コーディングパターン抽出と絞り込み

ステップ 1 で得られた要素列から表 2 のデータベースを作成し，SPADE アルゴリズムを適用してコーディングパターンの抽出を行う．ここでは最低 10 個の関数に出現するコーディングパターンのみを出力するように支持度を指定している．

また，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．そのままでは開発者が手作業ですべてを確認することは困難なため，以下の 3 つの条件で絞り込みを行った．

- 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

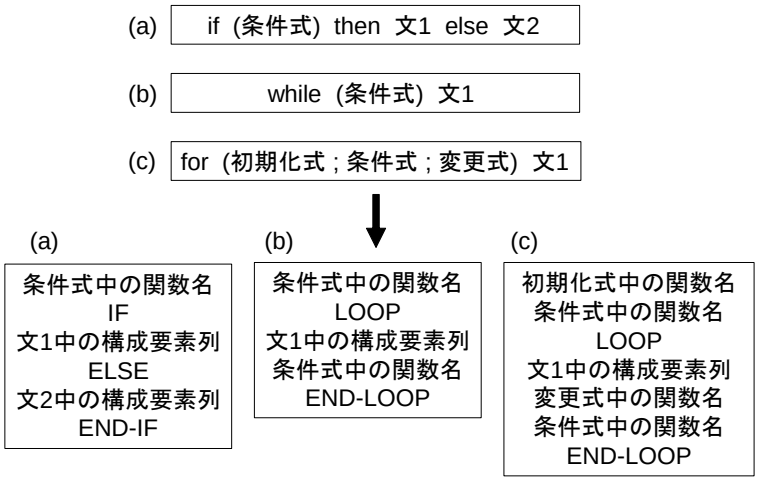

図 1　制御構造から抽出される要素

表 6　適用実験で用いた実験対象の規模

対象名	C ファイルの数	関数の数	LOC
ATK2/SC1	12	81	4,620
ATK2/SC1-MC	17	131	7,726
ATK2/SC3	16	135	7,698
ATK2/SC3-MC	19	172	10,244
Linux/pci	30	991	24,441
Linux/arm	61	738	18,672

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．

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逆に，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，抽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．それがマクロ定義された関数呼び出しの内部で goto 文を実行している場合である．本手法ではマクロ定義された関数呼び出しがコーディングパターンの要素として含まれる場合，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 文が含まれているかを調べている．また，ジャンプ先のラベルが#ifdef 命令によって切り替えられている場合についても考慮して各コーディングパターンのラベルの対応関係が取れているかの判断を行う．

また，SPADE アルゴリズムによるパターンマイニングでは頻出系列パターンとその部分頻出系列パターンがすべて出力される．そこで上の 3 つの条件による絞り込みを行った後，包含関係にあり，かつ同じ支持度のコーディングパターンを 1 つにまとめる操作を行う (コーディングパターンの極大化)．

4. 適 用 実 験

本研究では，ジャンプ命令とプリプロセッサ命令を考慮する本手法の有効性を確認するために 6 つの組込みプログラムに対して適用実験を行った．さらに，抽出されたコーディングパターンの有用性についても評価を行った．

4.1 適 用 対 象

対象には自動車制御用実時間 OS(オペレーティングシステム) である TOPPERS/ATK2 を用いた [11]．ATK2 にはいくつかの機能セットがあり，今回はその中から SC1，SC3，SC1-MC，SC3-MC を選択した．SC1 は基本機能セットであり，SC3 等は SC1 の機能拡張版である．また，Linux カーネルソースからも 2 つのプログラムを対象とした．1 つはアーキテクチャ固有のカーネルコードを含んだ arch から arm ディレクトリを選択した．もう 1 つはデバイスドライバをまとめた drivers から，PCI 仮想ドライバのためのソースコードをまとめた pci ディレクトリを選択した．表 6 にそれぞれの対象プログラムの規模を示した．ATK2 の各機能セット及び arm はサブディレクトリに存在する kernel ディレクトリ内の C ファイルを対象にした．そして pci はディレクトリ直下の C ファイルのみを対象に手法を適用した．

4.2 抽出されたコーディングパターン

手法の適用に際して，ステップ 3 によってコーディングパターンがどれほど絞り込まれたかを表 5 に示した．左から順番に，各絞り込みを行った後のコーディングパターンの数が記されている．また，極大化実施後のコーディングパターンの数は本手法による最終的な抽出数と同義である．絞り込みによって約 99%の不要と判断されたコーディングパターンが除外されており，特に ATK2/SC1 や SC3 では開発者の手作業による確認が可能な個数となっている．

そして，各実験対象から抽出されたコーディングパターンを表 7 に示す．ただし論文のスペースの都合上，一部のみを掲載している．表には対象名，抽出されたコーディングパターン，支持度，そしてコーディングパターンの長さを記述した．

4.3 コーディングパターンの評価

適用対象の内，TOPPERS/ATK2 に関しては外部仕様書や開発文書が公開されている [11]．そこで本研究では，ATK2 の 4 つの機能セットから得られたコーディングパターンが意味の

表 5　各絞り込み及び極大化を適用した後のコーディングパターンの数

対象名	抽出された コーディングパターン	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	極大化実施後	削減割合 (%)
ATK2/SC1	408,613	392,648	258,441	9,026	29	99.993
ATK2/SC3	157,148	150,259	97,909	3,639	64	99.959
ATK2/SC1-MC	17,561,490	17,499,108	17,204,504	154,253	240	99.999
ATK2/SC3-MC	34,279,185	34,246,413	34,138,960	274,025	511	99.999
Linux/pci	492,995	991	296	296	197	99.960
Linux/arm	31,959	17	14	14	7	99.978

表 7　抽出されたコーディングパターンの一例

対象名	コーディングパターン	支持度	長さ
(1)ATK2/SC1	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_error_hook	14	13
(2)ATK2/SC1	x_nested_lock_os_int() / x_nested_unlock_os_int()	34	2
(3)ATK2/SC3	CHECK_DISABLEDINT() / CHECK_CALLEVEL() / CHECK_ID() / CHECK_RIGHT() / x_nested_lock_os_int() / D_CHECK_ACCESS() / d_exit_no_errorhook: / x_nested_unlock_os_int() / exit_no_errorhook: / return / exit_errorhook: / x_nested_lock_os_int() / d_exit_errorhook: / call_errorhook() / goto / d_exit_error_hook	14	16
(4)ATK2/SC1-MC	CHECK_DISABLEDINT() / CHECK_CALLEVEL() / CHECK_ID() / CHECK_CORE() / 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_error_hook	11	18
(5)ATK2/SC3-MC	CHECK_DISABLEDINT() / CHECK_CALLEVEL() / CHECK_ID() / CHECK_CORE() / CHECK_RIGHT() / get_p_ccb() / x_nested_lock_os_int() / acquire_cnt_lock() / release_cnt_lock() / x_nested_unlock_os_int() / exit_no_errorhook: / return / release_cnt_lock() 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_error_hook	11	23
(6)Linux/pci	pci_read_config_word() / pci_write_config_word()	47	2
(7)Linux/arm	raw_spin_lock_irqsave() / raw_spin_lock_irqstore()	25	2

あるコーディングパターンかどうかを評価した．意味のあるコーディングパターンとは，バグ検出のために利用できる可能性を持つ機能的なコーディングパターンを指す．評価はまず，各機能セットから得られたコーディングパターンを支持度と長さでソートし，それぞれの上位 5 つ計 40 個のコーディングパターンを選択する．そしてそれらのコーディングパターンが外部仕様書や開発文書に含まれている情報を反映した機能的なコーディングパターンであるかを確認する．1 つの機能を表していると判断したコーディングパターンの数を算出することで今回の手法及び抽出されたコーディングパターンの評価とした．

その結果，40 個中 25 個のコーディングパターンが 1 つの機能を表したコーディングパターンであることを確認した．その中には例えば，表 7(1) のような割込み処理に伴うエラーチェックと対応するエラー処理という機能を含んだコーディングパターンがあった．また，(2) のように他プロセスからの割込みの禁止状態を管理するコーディングパターンも検出されている．

残りの 15 個は例えば，(2) に現れる 2 つの関数呼び出しが対になっていなかったり，断片的な要素しか含まれていないためにどのような機能を表しているか把握できなかったコーディングパターンが含まれている．

評価の結果，本研究では意味のあるコーディングパターンを抽出できたとし，今後バグ検出のために利用していくことができると判断した．

4.4 考　　察

ジャンプ命令，プリプロセッサ命令を考慮することの有効性． 表 7(1) のコーディングパターンは割込み処理に伴う例外処理を表している．評価の項でも述べたように，これは仕様書の情報を反映した意味のある機能的なコーディングパターンである．そしてこの中には goto 文及びラベル文が含まれている．これらの要素を考慮しなかった場合，この例外処理という機能を表すコーディングパターンは抽出されない．よって，ジャンプ命令を考慮することは有効であったと考えられる．また，各 CHECK 関数はマクロ関数でありその内部で goto 文を呼び出している．この goto 文のジャンプ先のラベルは#ifdef 命令によって条件分岐している．手法ステップ 3 ではラベル文が#ifdef 命令に影響を受けている場合を考慮してラベルの対応関係による絞り込みを行っているため，プリプロセッサ命令を無視することはできないと考えられる．

一方，表 7(6)，(7) のように Linux カーネルソースプログラムから抽出されたコーディングパターンはジャンプ命令に関する要素を含んでいない．しかし実際のソースコードを見てみると goto 文やラベル文は存在するため，この結果だけでは Linux カーネルソースプログラムに対しても手法が有効だったかは分からない．有効性を確かめるための方法として，コーディング

パターン抽出における支持度の指定値を下げての実験や対象を変えての実験が考えられる．実験を通して，TOPPERS/ATK2に対してのみ有効だったのかあるいは別の組込みプログラムに対してもジャンプ命令，プリプロセッサ命令を考慮する本手法が有効だったのかが明らかになると期待される．

計算速度． 手法の適用実験における計算時間は対象によって大きく変わる．CPU：Intel(R) Xeon(R) CPU E5-2690 @ 2.90GHz 2.90GHz，RAM：256GB の環境で ATK2/SC1 や SC3，Linux/pci，arm を対象とした場合，数秒から数十秒で全ステップが完了した．その一方，SC1-MC では 4 時間，SC3-MC に関しては計算完了までに 12 時間を要した．計算時間の大部分はステップ 2 のシーケンシャルパターンマイニングによるコーディングパターン抽出が占めており，表 5 と合わせてみると計算時間は抽出されるコーディングパターンの数に大きく依存していると考えられる．パターンマイニング部分を並列処理化することでこのようなコーディングパターンを多く含むプログラムへ適用した際の計算時間の短縮が見込まれる．

類似したコーディングパターン． また，今回の実験では類似したコーディングパターンが多く抽出された．例えば表 7(1) と類似したコーディングパターンとして，CHECK_ID() 関数が存在しないケースが確認された．これは ID のチェックを行う必要がない関数がいくつか存在するために起きた結果であることが仕様書から分かった．このような同じ処理を表す類似したコーディングパターンは 1 つのグループとして考えるべきであり，そのための方法として Xin らが提案する冗長性の低いパターン集合を見つける手法がある [12]．これは頻出パターンマイニングに対する手法ではあるが，この考え方をシーケンシャルパターンマイニングで抽出されたコーディングパターンに対しても適用してコーディングパターンのグループ化を行うことを検討している．

5. まとめと今後の課題

本研究ではバグ検出による信頼性向上を目的として，C 言語で記述されたプログラムからコーディングパターンを抽出する手法を提案した．本手法によるコーディングパターン抽出は既存手法と異なり，ジャンプ命令やプリプロセッサ命令を考慮している．そして，手法を TOPPERS/ATK2 及び Linux カーネルソースに対して適用し，抽出されたコーディングパターンの有用性について評価した．その結果，選択した 40 個中 25 個のコーディングパターンが意味のある機能的なコーディングパターンであることを確認した．

今後の課題としてはまず，本手法を様々な対象に適用する予定である．今回は自動車制御用実時間 OS と Linux カーネルソースを用いたが，その他の組込みプログラムに対しても適用実験を行い手法の有効性について検証する必要がある．また，今回コーディングパターンの抽出に使用したマイニングアルゴリズム以外のアルゴリズムを用いた場合や別のアプローチを用いた場合との比較を行って本手法の長所，短所について調査を行いたい．特に，プログラム依存グラフに基づいた抽出アプローチとの比較に興味がある [13]．

謝辞 本研究を実施するにあたり，貴重なご意見を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 ソフトウェア工学講座および名古屋大学 大学院情報科学研究科 情報システム学専攻 組込みリアルタイム研究室の皆様に感謝いたします．本研究は JSPS 科研費 25220003，26730036 の助成を受けたものです．

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] M.J. Zaki, “SPADE: An efficient algorithm for mining frequent sequences,” Machine Learning, vol.42, pp.31–60, 2001.
[3] Z. Li, S. Lu, S. Myagmar, Y. Zhou, “CP-Miner: Finding copy-paste and related bugs in large-scale software code,” IEEE Transactions on Software Engineering, vol.32, no.3, pp.176–192, 2006.
[4] Stankovic and John A., “Misconceptions about real-time computing,” IEEE Computer, vol.21, no.10, pp.10–19, 1988.
[5] 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.
[6] “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.
[7] 金明哲，R によるデータサイエンス，森北出版株式会社，2007.
[8] 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.
[9] 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，” 2006.
[10] 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.
[11] “TOPPERS プロジェクト/ATK2”. https://www.toppers.jp/atk2.html.
[12] 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.
[13] 山田吾郎，吉田則裕，井上克郎，“プログラム依存グラフの一貫性検査に基づく欠陥候補の検出手法，” 電子情報通信学会技術研究報告，vol.110，no.169，pp.23–28，2010.