前処理命令解析と関数呼出し解析に基づく機能スイッチ特定

相澤 遥也[1,a)]　森園 宏紀[1,†1]　小林 隆志[1]

概要：ソースコード中の可変点，特にユーザがオプションで切り替える個所を特定することは，開発者のプログラム理解に役立つ．本論文では C/C++の静的な可変点を構成するマクロのうち，ユーザが構成ごとに切り替えるマクロ (機能スイッチ) を抽出する方法を提案する．提案手法では，マクロ変数と関数の関係グラフを構築し，グラフ上の構造的特徴から機能スイッチを推定する．提案手法を実装し，Linux を含む 4 つの OSS に対して評価を行い，構造的特徴を利用する方法が有効であることを確認した．

1. 序論

ソフトウェアの中には，機能の切り替えや環境の差異を吸収するために，構成を設定可能な形で開発，配布されるものがある．このようなソフトウェアは Configurable Software(CS) と呼ばれ，OS，サービス指向アーキテクチャ，データベース，組み込みなど，多くの分野で利用されている [1,2]．また，オープンソースソフトウェアのプロジェクトにも，コンフィギュラブルなものは多数存在する．

CS は構成を静的に変更する方法と，動的に変更する方法の 2 つに分類される [3]．動的な CS は実行時引数や設定ファイル，レジストリファイルなどの外部リソースから読み込まれたパラメータによって実行を分岐する方法が一般的であり，実装は共通である．これに対して静的な CS は，ビルド時に構成を指定するなどして，コンパイル時にソースコードを変更し，プログラムの実行前に振る舞いを決定する．静的な CS は動的なものに比べて実行ファイルが小さくなるほか，実行時のオーバーヘッドがなくなるという利点がある．

本研究では，静的な方法である C/C++の前処理命令を対象とする．前処理命令で利用される変数をマクロ変数と呼ぶ．前処理命令には指定した文字列と置換するマクロ定義，マクロ変数や定数式を判定してソースコードの一部を組込み，または抑止して切り替えるものなどがある．C/C++の前処理ではこれらを組合わせて，コードの一部をマクロ変数で置換する，または，一部を有効化もしくは無効化することで，静的に柔軟な構成変更を実現する．前処理命令は複雑な依存関係や条件に依存するコードの切り替えが可能である反面，1 つの機能の切り替えが多数のコードの切り替えを伴うため，開発者はその追跡に多大な時間を要する [4]．

本研究では，ソースコードやマクロ定義の変化を指定するマクロ変数を**前処理スイッチマクロ**と呼ぶ．また，その中で特に振る舞いの変化を指定するものを**機能スイッチマクロ**と呼ぶ．機能スイッチマクロは 2 種類に分類できる．一つは開発者の要件を満たすためのもので，ソフトウェアの移植性のために用いられる．もう一つはエンドユーザーの要件を満たすためのもので，使用者の望む振る舞いを実現するためにビルドの際に設定を行うことが可能なものである．

機能スイッチマクロによって引き起こされる振舞いの変化を把握することは，その機能を理解することの助けとなる．例えば，プログラムの可変な機能に関心をもってコードを読んでいるとき，機能スイッチマクロによる分岐は注目すべき箇所である．また，機能スイッチマクロを見つけることはコードの理解だけでなく，Kbuild のようなコンパイル時マクロ変数の自動設定ツールをプロジェクトに導入する場合にも必要となる．

しかし，マクロ変数には機能スイッチマクロ以外にも多くの役割がある．例えば，コードの切り替えを伴わない関数や定数として利用されるもの，コードを切り替える前処理スイッチマクロであっても，OS や CPU アーキテクチャ，参照できるライブラリの有無などの環境の差異を解決するような，機能スイッチマクロではないものも存在する．他にも，ヘッダファイルを include する際に定義が重

[1] 東京工業大学 情報理工学院
School of Computing, Tokyo Institute of Technology
[†1] 現在，株式会社 Donuts
Presently with Donuts Co. Ltd.
[a)] yaizawa@sa.cs.titech.ac.jp

© 2019 Information Processing Society of Japan

複されることを防ぐ include guard の用途で使用される場合もあり，単にソースコードの変化に関わることをもって機能スイッチマクロとそれ以外を区別することはできない．また，機能スイッチマクロは最終的にはプログラムの振る舞いに関わるコードを変化させるものであるが，直接切り替えるもの以外にも，マクロ定義を変化させることで間接的にコードを変化させるものや，関数呼出しを介して依存するものもある．このため，識別には多数のファイル内に横断的に存在する依存関係を解析する必要があり，単一のファイルを見ただけでは容易に識別することができない．マクロ変数の命名規則からその役割が分からない場合，機能スイッチマクロを特定するには多くの労力が必要である．

この問題に対し我々は，プログラムの振る舞いの変化が関数の定義や呼出しの切り替えを通して行われることに着目する．マクロ変数の依存関係と関数呼出し関係をグラフ構造としてとらえ，構造上の特徴から機能スイッチマクロを特定する手法を提案する．加えて，特定精度向上のためのヒューリスティクスも提案する．

本研究では，提案手法を実装し，評価実験によってその有効性を明らかにする．具体的には以下の研究課題に回答する．

RQ1 マクロ変数の依存関係と関数呼出し関係の構造上の特徴で機能スイッチマクロを特定するのに有効な関係は何か

RQ2 構造的特徴を利用すると機能スイッチマクロはどの程度発見できるか

RQ3 構造的特徴による判別にヒューリスティクスを加えて精度は向上するか

評価の結果，機能スイッチマクロの特定には，関数に関係するマクロ変数を単純に抽出するもの，それらの関数の間に呼び出し関係を要求するものがそれぞれ Precision，Recall の面で優れ，これらの中間的な方法を使ったものが F-mesure の面で優れることが分かった．このときの F-measure は 0.17 から 0.63 であった．また，これらの構造的特徴にヒューリスティクスを組み合せるとより効果的であった．

2. 関連研究

ソースコード中における前処理命令の出現について分析した研究は多数存在する．Sutton らは静的解析ツール srcML を用いてライブラリを解析することで，前処理命令内で利用できるポータビリティに関するパターンを発見した [5]．

Liebig らは前処理命令を解析するツール cppstats[*1] を作成し，40 の OSS を対象に複数のマクロ変数による依存関係の発生や前処理命令による変更の大きさという観点で

*1 http://fosd.net/cppstats

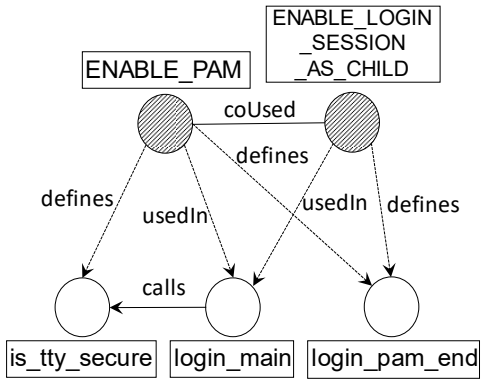

図 1 BusyBox における前処理命令と関数呼出しの依存関係グラフの一部

#ifdef 命令の使われ方を測定した [2]．

Kästner らは#if 命令に着目し，機能スイッチマクロの実行時の振る舞いを明らかにする TypeChef を提案した [6,7]．Tartler らはソースコード中の前処理命令の依存関係と比較する Undertaker を開発した [8]．Nadi らは Undertaker を Makefile にも応用し，Linux カーネルにおける依存関係の矛盾を特定した [9]．Tibor らはソースコード上でマクロ変数が呼ばれる様子を可視化する Visual Studio のアドインを提案した [10]．また，Nadi らはソースコードをマイニングすることにより前処理スイッチマクロの依存関係を抽出し，前処理スイッチマクロの制約モデルを自動生成する手法を提案した [11]．

これらの研究は前処理命令の出現や依存関係，制約の発見に主眼を置いており，プログラムの機能の切り替えを実現する機能スイッチマクロを特定する試みは我々が初めてである．

3. 提案手法

本手法は前処理命令解析と関数コール解析を行い，マクロ変数，関数間の関係をグラフとして表現し，グラフ上の特徴から機能スイッチマクロの特定を行うための 3 つのルールを提案する．また，これらのルールと併用する 2 つのヒューリスティクスを提案する．

3.1 グラフモデル

本手法ではソースコード中のマクロ変数及び関数をノード，それらの依存関係や呼出し関係をエッジとする，図 1 のような有向グラフとして扱う．網掛けのノードはマクロ変数，白抜きのノードは関数を表す．以下にグラフのエッジとして表現する関係について定義する．

defines: 前処理スイッチマクロ内に出現する関数と条件となるマクロ変数間の，マクロ変数から関数への有向関係を defines と定義する．例えば，リスト 1 のコードの場合，マクロ変数 A が定義されているとき関数 func1 が定義される．機能スイッチマクロであるマクロ変数は，機能とな

© 2019 Information Processing Society of Japan

る関数の定義を切り替えている可能性があるため，関数と defines 関係にあるマクロ変数に注目する．

```
1 #if A
2 void func1(){
3   printf("A is ON");
4 }
5 #endif
```

リスト 1　defines 関係が成り立つ例

usedIn:　関数とその中に出現する前処理スイッチマクロの条件となるマクロ変数間の，マクロ変数から関数への有向関係を usedIn と定義する．リスト 2 のコードの場合，マクロ変数 B の有無によって関数 func2 の一部が変更される．関数の一部を変更するような働きをするマクロ変数は機能スイッチマクロである可能性がある．

```
1 void func2(){
2 #if B
3   printf("B is ON");
4 #endif
5 }
```

リスト 2　usedIn 関係が成り立つ例

calls:　関数内で関数呼出しを行う関数同士の有向関係を calls と定義する．この関係は機能スイッチマクロを直接示さないが，機能の一部が関数化されている可能性を考慮するために利用する．

coUsed:　#if 命令の条件式として同時に出現するマクロ変数同士の関係を coUsed と呼ぶ．この関係は向きを持たない．機能スイッチマクロに関係のあるマクロ変数を探索するために利用する．

これ以外の関数とマクロ変数の関係，例えば#ifdef が関数を部分的に含んでいるような場合は無視する．

3.2　構造的検出手法

前節で定義したグラフ構造を利用して，機能スイッチマクロを特定するための 3 つのルールを提案する．

3 つのルールは最も緩い制約の簡易ルールと，厳しい制約の厳格ルール，中間の拡張ルールからなる．

3.2.1　簡易ルール

簡易ルールは，defines，usedIn 関係を持つマクロ変数を機能スイッチマクロと判別する．すなわち，関数に変更を加えるようなマクロ変数は全て機能スイッチマクロであるとみなす．

3.2.2　厳格ルール

単純な defines，usedIn 関係より高精度な機能スイッチマクロの候補として，これらにかかわる関数が calls 関係にあるようなグラフ上の 4 種類の三角関係に着目する．同じマクロ変数によって変更され，かつそれらの間の関数呼出しを条件とすることで，機能を実装する複数の関数に変更を加えるマクロ変数を機能スイッチマクロであるとみなす．以下の関係に出現するマクロ変数が厳格ルールが判別する候補である．

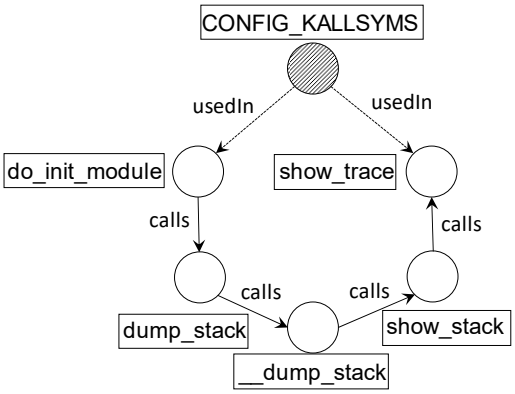

図 2　Linux における間接的な呼出しを含んだ UU 関係

UD 関係: マクロ変数が caller 関数で usedIn され，マクロ変数によって callee 関数が defines される関係を UD 関係と定義する．この関係は機能スイッチの切り替えによって，callee 関数が定義され，caller 関数から callee 関数を呼ぶ場合に出現する．BusyBox のプロジェクトでは図 1 に示すように，マクロ変数 ENABLE_PAM，caller 関数 login_main，callee 関数 is_tty_secure がこの関係にある．

UU 関係: マクロ変数が caller 関数，callee 関数の両方で usedIn される関係を UU 関係と定義する．この関係は呼出し関係にある 2 つの関数両方を 1 つの機能スイッチが部分的に変更する場合に出現する．

DD 関係: マクロ変数によって caller 関数，callee 関数の両方が定義される (defines) 関係を DD 関係と定義する．この関係は呼出し関係にある 2 つの関数両方が，同じマクロ変数によって定義される場合に出現する．

DU 関係: マクロ変数によって caller 関数が defines され，マクロ変数が callee 関数で usedIn される関係を DU 関係と定義する．callee がユーティリティ関数のように多くの関数から呼出されるときこのような関係が出現する．

関数呼出し延長: UD，UU，DD，DU 関係が関数の呼出し関係を利用して機能スイッチマクロを推定する一方で，複数の関数呼出しを経た先に同じマクロ変数による可変部分が存在する可能性もある．この場合も同様にそのマクロ変数は機能スイッチマクロである可能性がある．実際に，Linux Kernel のデバッグ用の機能スイッチマクロである CONFIG_KALLSYSMS は，図 2 で示すように do_init_module 関数と show_trace 関数の両方と usedIn 関係にあり，これらの間には 4 回の関数呼出しが存在する．

3.2.3　拡張ルール

厳格ルールによる推定が 2 つの関数との関係を要求する一方で，マクロ変数の多くが 1 つの関数に対してのみ振る舞いの変化を生じさせているということが既存の研究から明らかになっている [12,13]．本論文では，これら多くのマクロ変数から機能スイッチマクロを判別するために，機能

スイッチマクロと推定したマクロ変数との関係に着目する．

ある機能にかかわる関数のみと関係するようなマクロ変数は，その機能と依存関係を持つか関連する機能の機能スイッチマクロである可能性がある．このような機能スイッチマクロを抽出するために，厳格ルールの caller と callee のいずれかと呼出し関係にある関数に対して，usedIn 及び defines 関係にあるマクロ変数を収集する．この中で，これらの関数のみと関係をもつようなマクロ変数を機能スイッチマクロと特定する．

3.3 ヒューリスティクス

3.3.1 条件式に同時出現するマクロ変数

複数の機能スイッチマクロがひとつの機能切り替えの条件となっている場合がある．ヒューリスティクスとして，条件付きコンパイル命令の条件式に複数のマクロ変数があり，片方が機能スイッチマクロであるとわかっているとき，もう片方のマクロ変数も機能スイッチマクロであるとみなす．すなわち，機能スイッチマクロと coUsed 関係にあるマクロ変数を再帰的に機能スイッチマクロと判定する．例えば図 1 においてマクロ変数 ENABLE_PAM が機能スイッチマクロとされたとき，これと coUsed な関係にあるマクロ変数 ENABLE_LOGIN_SESSION_AS_CHILD は，このヒューリスティクスの下で機能スイッチマクロとみなされる．

3.3.2 include guard 除去

C 言語のヘッダファイル内で定義された関数と include guard で利用されるマクロ変数は形式上 defines の関係となるため，このような関数間に呼出し関係があれば DD 関係の厳格ルールが成立してしまう．これを防ぐために，本論文では次の条件を持つマクロ変数を include guard とみなして，あらかじめ除去する．

(1) ソースコードのどの前処理命令よりも先に#ifndef によって分岐が行われ，どの前処理命令よりも後ろで #endif によって閉じられる
(2) (1) の#ifndef...#endif の外に (コメントを含む) 空白以外が存在しない
(3) (1) の#ifndef...#endif の内部にそのマクロ変数による #define が存在する

4. 実装

提案手法に基づきツールを実装した．本ツールの動作は (1) ソースコードの解析，(2) グラフの構築，(3) 機能スイッチの抽出の 3 フェーズからなる．

4.1 ソースコードの解析

srcML [14] を用いてソースコード中から前処理命令及び関数定義，関数呼出しの情報を抽出した．

判定対象のマクロ変数

提案手法では定義されたが一度も利用されないマクロ変数は利用しない．すなわち#if cond または#define body の cond, body 内に出現するマクロ変数のみを取得する．

同名の関数の扱い

関数は定義されるファイル名を含めることでそれぞれ区別する．前処理スイッチマクロによってコンパイル時に片方を利用する場合，同一ファイルに同名の関数が複数存在する場合がある．この場合は両者を同一の関数とみなす．

関数の呼出し先

関数の呼出しはその名前により判別を行う．呼び出される関数と同名の関数が複数ある場合は次のように呼出し先を解釈する．

(1) 呼出し元と同じファイルに同名の関数がある場合，その関数のみと calls 関係があるとする
(2) 呼出し元と同じファイルに同名の関数がない場合，同名の関数すべてと calls 関係があるとする

呼出し先の関数が解析した名前の中にない場合 (外部ライブラリなど)，呼出し先不明とし，MissingFunction と calls 関係を作成する．

4.2 グラフの構築

抽出した情報をグラフとしてデータベースに格納する．データベースには Neo4J を利用した．関数に関する情報は Function ノード，マクロ変数に関する情報は Macro ノードと別種のノードに格納した．

3.1 節で述べた関係は同種の関係が同一の始点，終点に複数存在する場合がある (例えば関数 A のなかに関数 B の呼出しが複数回ある)．この場合これらの関係は 1 つの関係として格納し，探索のコスト削減を図った．

4.3 機能スイッチの特定

構築したグラフをもとに機能スイッチマクロを特定する．Neo4j はグラフ上のノード，エッジが特定の関係にあるような部分グラフを検索することができる．このため，簡易ルールである，関数とマクロ変数が defines 関係にあるようなものはもちろん，厳格ルールの 1 つのマクロ変数と 2 つの関数のそれぞれが特定の関係で結ばれているようなものまで，クエリ 1 つで問い合わせることができた．

5. 実験準備

評価実験に先立ち，対象とした OSS について説明し，それぞれについて機能スイッチマクロの正解セットを作成する．また，3.3.2 節で述べた，ヒューリスティクスである include guard 除去の有効性を確認する．評価実験は include guard に利用されるマクロ変数を除去した状態で行う．

© 2019 Information Processing Society of Japan

表 1　対象プロジェクト

プロジェクト	バージョン	C/H ファイル数	総行数 (KLOC)
BusyBox	1_27_0	719	256
uClibC	0.9.33	3,950	436
CoreBoot	4.6	11,274	2,842
Linux	v4.15-rc1	46,322	22,898

表 2　正解セット

プロジェクト	Kconfig オプション数	機能スイッチマクロ数
Busybox	989	528
uClibC	287	143
CoreBoot	2,338	430
Linux	17,136	5,712

5.1　対象

実験対象として，Kconfig によって振る舞いを設定することのできる 4 つの OSS，BusyBox，uClibC，CoreBoot，Linux Kernel を用いる．それぞれの OSS の規模を表 1 に示す．これらのプロジェクトはいずれもソースコードが 20 万行を超える中規模以上の CS であり，特に Linux は 2000 万行を超える大規模な CS である．Kconfig とは，Linux カーネル用に開発された，ビルドを行う際にユーザーが切り替えを行うことのできる設定を定義するモデル言語である．

5.2　正解セット

正解となる機能スイッチマクロの集合は Kconfig を利用して作成する．本実験の対象とする機能スイッチマクロは，ソフトウェアの利用者あるいは開発者が自由に機能の切り替えを行うものである．Kconfig で定義されるオプションは，その設定の間に依存関係があるものの，原則的にユーザーがオンまたはオフを自由に決めることができる機能スイッチマクロである．Kconfig には開発者がソフトウェアの移植の際に使う機能スイッチマクロは含まれておらず，また，プロジェクトによっては環境の差異を吸収するためのオプションが含まれることもある．しかし，機能スイッチマクロを機械的に作成することが可能であり，かつ対象としたプロジェクトにおいては十分な数を抽出できるため，正解セットは Kconfig に含まれるもののみを対象とした．

Kconfig によって定義されるオプションは，そのままソースコード中のマクロ変数名とはならず，名前に変化が起こる．このため，正解セット作成の際にはこの変化後の名前を用いる．

例えば BusyBox では，オプション名に ENABLE_という接頭辞をつけた名前のマクロ変数が autoconf.h に定義される．実際に設定を行い，このファイルから抽出した．また，ENABLE_という接頭辞が付いているにも関わらず，Kconfig に含まれないマクロ変数が 5 つ存在したが，コメントなどから判断し，正解セットに含めた．

Kconfig 中で定義される全てのオプションがソースコード中で用いられているわけではない [15]．Kconfig のオプションの多くはビルドを行う際にコンパイルをする対象となるファイルを制御するためにも使用され，ファイル単位で可変性を実現させる．この場合は，Makefile にのみその記述が利用される．本研究ではソースコード中において前処理命令を用いて可変性を実現したもののみを対象とし，Makefile 中のものについては対象としない．各プロジェクトにおける正解セットの大きさを表 2 に示す．

5.3　グラフの規模

ルールを適用するために，各 OSS から 3.1 節で提案したグラフを構築した．各プロジェクトのグラフの規模を結果を表 3 に示す．ノード，エッジともにプロジェクトの規模が大きくなるほど増加した．グラフの構築に要した時間もこれに比例し，一番規模の小さい BusyBox が 126 秒，uClibC は 12 分，CoreBoot は 1 時間 12 分，一番規模の大きい Linux で 3 時間 56 分であった．

いずれのプロジェクトにおいても defines，usedIn の関係が存在し，グラフにおいてもマクロ変数による変更が表現されていることが確認できた．

検出されたマクロ変数のうち正解の割合は，BusyBox で 8.7%，Linux で 0.37%であった．

5.4　include guard で利用されるマクロ変数の特定

各プロジェクトについて，3.3.2 節で述べた include guard の条件を満たすマクロ変数を抽出し正解セットを比較し，この手法が誤って機能スイッチマクロを除外しないことを確かめた．

本手法で include guard と判定したマクロ変数はそれぞれ，BusyBox で 40，uClibC で 359，CoreBoot で 1,997，Linux で 17,383 であった．このうち，正解セットに含まれたものは一切存在しなかった．したがって，以降の実験ではこれらのマクロ変数は機能スイッチマクロでないと推定する．

6.　評価実験

まず，RQ1 と RQ2 に回答するために以下の実験を行う．

(1) 簡易ルールと厳格ルール，前処理スイッチマクロの比較
(2) 厳格ルールにおける関数呼出し延長の効果を検証
(3) 拡張ルールの効果を検証

これらを総合して構造的検出手法の効果をまとめる．

次に，RQ3 に回答するために，条件式中の同時出現マクロ変数探索による効果を検証し，5.4 節の結果と合わせ，ヒューリスティクスの効果を確認する．

6.1　簡易ルールと厳格ルールの比較

各対象プロジェクトから簡易ルールと厳格ルール，比較

© 2019 Information Processing Society of Japan

表 3　構築されたグラフの規模

プロジェクト	Function	Macro	MissingFunction	usedIn	defines	calls	coUsed
BusyBox	4,527	6,083	881	1,347	1,369	20,332	856
uClibC	5,031	14,970	737	1,606	2,032	20,258	1,337
CoreBoot	37,296	105,663	1,129	2,215	3,327	380,602	1,239
Linux	508,037	1,545,795	13,563	20,034	98,351	3,886,147	4,225

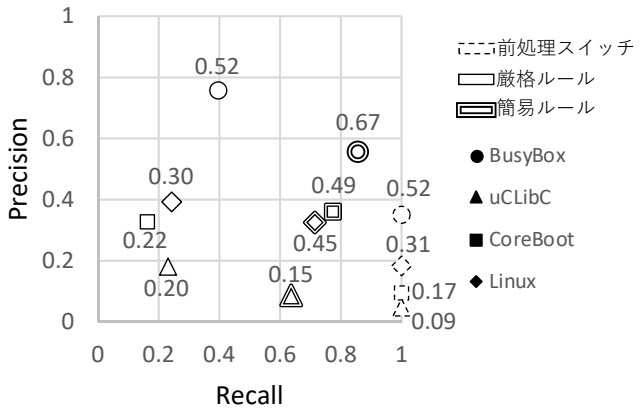

図 3　簡易ルールと厳格ルールのプロジェクト間の比較

対象として，前処理スイッチマクロを全て機能スイッチマクロであると判定するものをそれぞれ利用して機能スイッチマクロを抽出する．正解セットと比較して Precision-Recall を比較する．簡易ルールは defines，usedIn のいずれかの関係にあるもの，厳格ルールは UD，DU，DD，UU のいずれかの関係にあるものを対象とする．

結果

Precision-Recall を各プロジェクト間で比較すると，図 3 のようになる．マーカー付近の数字は F-measure の数値である．提案した 2 つのルールは Linux を除く全てのプロジェクトで，前処理スイッチマクロを全て機能スイッチマクロとみなすものより F-measure が高かった．プロジェクトによって精度に差があるものの，厳格ルールは Precision が高く，簡易ルールはそれと比較して Recall が高いという傾向にあった．F-measure は多くのプロジェクトで簡易ルールが最も優れていた．

6.2　関数呼出し延長の効果

前節の実験で，関数呼出しが機能スイッチマクロ特定の精度向上に効果があることがわかった．本節では，間接的な呼出しでも効果があるかを明らかにする．

実験設定

各プロジェクトに，厳格ルールのそれぞれについて間に他の関数呼出しが入ることを許容する回数を 1〜4 まで変化させて評価指標を計測する．

結果

どのプロジェクトのどの関係においても，計算量は関数呼出しの回数に対して指数的に増加したのに対し，評価指標に与える影響はわずかであった．例えば BusyBox で関数呼出しを 4 回まで許すと，1 回のときと比べて F-measure の上昇は，UD 関係で 0.003，DU 関係で 0.03，UU 関係で 0.05，DD 関係で 0.003 のみであった．上昇の要因は Recall によるものであった．したがって，関数呼出しが機能スイッチマクロ特定の精度向上に寄与するのは，直接の呼出しに限ると結論付けられる．

6.3　拡張ルールの効果

実験設定

厳格ルールの対象となる三角関係の頂点となる関数を，caller，callee，またはその両方に関数呼出しを辿り，それらの関数のみと usedIn もしくは defines を持つマクロ変数を機能スイッチマクロと判別し，評価指標を計測する．プロジェクトごとに，関数を呼出しを辿る回数を 0〜8 まで変化させ，これによる変化を確認する．厳格ルールは 4 つの関係のうち，いずれかが成り立つのことを条件とする．

結果

各プロジェクト，各呼出し方向の関数呼出し 4 回までの評価指標を表 4 に示す．下線付きの数字はプロジェクトと指標に対して最も大きい数値である．全体として，辿る関数呼出しの回数を増やすと Recall が向上するが，Precision の低下は小さかった．このため F-measure は向上した．呼出し回数を 4 回より増やしても F-measure はほとんど変化しなかった．関数呼出しの辿る方向による差異としては，特に caller 方向に辿るものは Precision の低下がわずかであるか，プロジェクトによっては大きく上昇するものもあった (CoreBoot)．Callee 方向に辿るものはこれに比べると Precision は劣化するものの，両方向での Recall が伸びたことから，異なる機能スイッチマクロを検出できたことがわかる．厳格ルールのみのものと比較すると，Recall は大きく改善したが，Precision は低下することもあった．最も精度が良かったのは関数を両方向に 4 回辿るものであった．

3 つルールを比較すると，簡易ルールは Recall が，厳格ルールは Precision がそれぞれ高く，拡張ルールは中間的な性能だが F-measure は最も高かった．

RQ1 に対する回答 : 機能を構成する関数を推定し，それらの関数のみと関係を持つマクロ変数を特定する，拡張ルールが有効であった．

© 2019 Information Processing Society of Japan

表 4　拡張ルールの効果

		F-measure												
			caller				callee				both			
		0	1	2	3	4	1	2	3	4	1	2	3	4
BusyBox	Precision	**0.68**	0.66	0.67	0.65	0.65	0.62	0.59	0.58	0.58	0.62	0.59	0.57	0.58
	Recall	0.53	0.58	0.60	0.61	0.62	0.62	0.64	0.65	0.65	0.65	0.68	0.68	**0.70**
	F-measure	0.60	0.62	0.63	0.63	**0.64**	0.62	0.61	0.61	0.61	0.63	0.63	0.62	0.63
uClibC	Precision	**0.11**	**0.11**	0.10	0.10	0.10	**0.11**	**0.11**	**0.11**	**0.11**	**0.11**	**0.11**	0.10	0.10
	Recall	0.29	0.34	0.34	0.36	0.38	0.41	0.45	0.45	0.45	0.47	0.50	0.50	**0.51**
	F-measure	0.16	0.17	0.15	0.16	0.16	**0.18**	**0.18**	**0.18**	0.17	**0.18**	0.17	0.17	0.17
CoreBoot	Precision	0.37	0.37	0.37	0.45	**0.46**	0.36	0.40	0.39	0.38	0.36	0.38	0.42	0.41
	Recall	0.25	0.30	0.34	0.54	0.58	0.34	0.46	0.50	0.51	0.37	0.51	0.67	**0.70**
	F-measure	0.29	0.33	0.35	0.49	0.51	0.35	0.43	0.44	0.44	0.37	0.44	0.51	**0.52**
Linux	Precision	**0.37**	0.36	0.36	**0.37**	**0.37**	0.35	0.34	0.34	0.33	0.35	0.34	0.35	0.35
	Recall	0.26	0.35	0.40	0.43	0.44	0.32	0.35	0.36	0.36	0.40	0.46	0.50	**0.52**
	F-measure	0.31	0.36	0.38	0.40	0.40	0.33	0.34	0.35	0.35	0.37	0.39	**0.41**	**0.41**

RQ2 に対する回答：拡張ルールを用いた時の機能スイッチマクロ特定の精度は，F-measure で 0.17〜0.63 であった．

表 5　条件式中の同時出現マクロ変数探索による効果

ルール	coUsed	F-measure			
		BusyBox	uClibC	CoreBoot	Linux
簡易ルール	なし	0.67	0.15	0.49	0.45
	あり	0.67	**0.16**	0.49	0.45
拡張ルール	なし	0.63	0.17	0.52	0.41
	あり	**0.64**	**0.18**	**0.53**	**0.42**
厳格ルール	なし	0.52	0.20	0.22	0.30
	あり	**0.55**	**0.24**	**0.27**	**0.32**

6.4　条件式中の同時出現マクロ変数探索による効果

機能スイッチマクロとともに条件式で利用されているマクロ変数を機能スイッチマクロと推定することの効果を確認する．複数のプラットフォームに対する差分の吸収に用いられるマクロ変数が AND(&&) によって使われる場合が多く見られたため，条件が OR(||) のものに限定する．また，条件が AND のものは#ifdef 命令がネストしたものと同様に働くため，AND と OR は区別すべきである．

実験設定

厳格ルール，拡張ルール，簡易ルールのそれぞれで抽出されたマクロ変数を起点とし，coUsed 関係でかつ OR で結ばれているものを辿り，到達できるマクロ変数全てを機能スイッチマクロとし，F-measure の変化を比較する．

結果

各プロジェクト，各基本ルールについて F-measure を測定した結果を表 5 に示す．多くのプロジェクトで F-measure が上昇し，悪影響を与えたものはなかった．上昇の原因を調査したところ，Recall が上昇していた．組合わせた基本ルール毎に見ると，もともとの Recall が低いルール程 F-measure は大きく上昇していた．条件式中で同時出現するマクロ変数を探索するというヒューリスティクスは有効であったと言える．

RQ3 に対する回答：全ての構造特徴による判別に対して，2 つのヒューリスティクスは悪影響なく精度を向上させた．

7.　議論

実験結果を考察し，機能スイッチマクロに対するさらなる理解と，今後の手法改善につなげるための議論を行う．誤判定したマクロ変数をソースコード上で調査し，誤って機能スイッチマクロと判定されてしまった偽陽性マクロ変数と，機能スイッチマクロであるのに判定されなかった偽陰性マクロ変数の 2 種類に分けて議論する．

7.1　偽陽性マクロ変数

デバッグ用マクロスイッチ

DEBUG や TEST などのマクロ変数が，簡易ルール及び厳格ルールによって機能スイッチマクロと判定された．これらのマクロ変数は多くのマクロ変数に影響を与え，関数やその一部を切り替える．これらは利用者が Kconfig で自由に設定するものではなく，開発者は必要に応じて切り替えるものである．本研究における機能スイッチマクロは，利用者または開発者が切り替えできるものであるため，これらが特定されるのは妥当である．

開発者が固定したマクロスイッチ

デバッグ用マクロと同様に，開発者側から見れば機能スイッチマクロであったものは他にもあった．特に uClibC でこの影響は大きく，libm ディレクトリの下には多くのファイルで，_IEEE_LIBM というマクロ変数が利用され，それぞれの関数に可変性を与える機能スイッチマクロと

しての役割を担っており，実際に検出された．しかし，このマクロ変数は Kconfig から設定できず，偽陽性であった．一方，冒頭のコメントには Copyright (C) 1993 by Sun Microsystems という記述があった．uClibC の開発開始は 1999 年からなので，これらのファイルは別プロジェクトから移植されたものである．よってこのマクロ変数は開発者が移植の際にオフにすると決めた機能スイッチマクロと捉えることができる．

ソースコード中で#define 命令によって定義されているマクロ変数をオン状態で固定された機能スイッチマクロとみなし，判定から取り除いたところ，uClibC における厳格ルールの pricision は 0.18 から 0.47 に大幅上昇した (recall は 0.23 から 0.22 へ微減)．

以上のように実験において偽陽性であったマクロ変数には開発者にとっての機能スイッチマクロであるものが多数含まれていたため，本手法の実際の精度はより高いものであると考えられる．

プラットフォームの差異を吸収する前処理スイッチマクロ

一方で，偽陽性であったマクロ変数の中には複数のプラットフォームの差異を吸収するために利用されるものが存在する．例えば，__GNU__というマクロ変数は，実験対象とした 4 つのプロジェクト全てで偽陽性であった．このマクロ変数は gcc に依存したマクロであり，コンパイラバージョンの差分吸収の役割がある．このようなプラットフォームの差分吸収のために，専用の関数とそれを切り替えるためにマクロ変数を利用することは一般に行われるため，本手法の精度を低下させる要因となる．

しかし，プラットフォームに依存するマクロ変数は，複数のプロジェクトで同様に利用されていることが期待できるため，多くのプロジェクトで利用されるマクロ変数を収集できれば，プラットフォーム依存のマクロ変数を判定できる可能性がある．

7.2 偽陰性変数

関数マクロに変更を与えるマクロスイッチ

機能スイッチマクロの中には直接関数に影響を与えず，関数マクロの一部分にのみ影響を与えるものが存在する．例えば，BusyBox の機能スイッチマクロである FEATURE_VERBOSE_USAGE は関数マクロ MAKE_USAGE を利用して振る舞いを変更するため，本手法では特定できなかった．

機能切り替え以外の Kconfig オプション

Linux の Kconfig は本来機能スイッチマクロではないオプションを含んでいる．例えば arch/mips の CPU_HAS_SMARTMIPS や CPU_HAS_MSA などである．これらのオプションはビルドの際にユーザーがオンまたはオフに切り替えることができるが，実際はターゲットのプラットフォームに依存する切り替えをユーザーにやらせているものである．また，CoreBoot においてもそのような Kconfig オプションの存在がドキュメント中に説明されている．

8. 妥当性に対する脅威

ツールの実装において内的妥当性が存在する．srcML によるソースコードの解析は時間効率を優先するために厳密でなく，解析に誤りがある場合が確認されている．大規模なシステムの全ソースコードを対象に実験を行ったため，確認できていない誤りが存在する可能性がある．しかし，解析はファイル単位で完結しているため，実験結果の妥当性には影響を与えないと考える．

実験結果の一般性に関して外的妥当性が存在する．データベースの構築に関しては 2000 万行を超える大規模な対象に適用できたため，一般の CS に対しても適用可能である．実験の対象を Kconfig を利用するプロジェクトに限定したため，そうでないプロジェクトでは傾向が異なる可能性がある．しかし，本手法はマクロ変数と関数の構造的特徴を利用するものであるため，Kconfig がなくとも有効であると考える．

9. 結論

本論文では，機能スイッチマクロを特定するために，関数の呼出し関係と前処理命令中に出現するマクロ変数との依存関係をグラフとして扱い，グラフ上の特徴から特定を行う方法を提案した．本手法はグラフの構造的特徴から機能スイッチマクロを抽出する簡易ルール，厳格ルール，拡張ルールと，これらと組み合わせる 2 つのヒューリスティクスから構成される．

グラフの構築および提案手法をツールで実装し，4 つの CS を対象に評価実験を行った．評価実験の結果，簡易ルールと厳格ルールは，それぞれ Recall，Precision の面で優れ，F-measure が最も高いのは拡張ルールであることが分かった．また，ヒューリスティクスを基本ルールに組み合わせることで性能を改善することが明らかになった．グラフ上の特徴で機能スイッチマクロを判別するアプローチは有効であることがわかった．

今後の課題

今後の課題としては以下が挙げられる．

機能スイッチと，機能スイッチの特徴を持つプラットフォーム間の差異の吸収に使われるマクロ変数の区別．プラットフォーム間の差異の吸収に利用されるマクロ変数の中には，機能スイッチマクロと同様に振る舞いの変化を伴うものが存在する．本手法ではこれらを区別することはできない．しかし，プラットフォーム固有のものであるなら，異なるプロジェクトにおいても同じ名前で使われることが考えられるため，複数のプロジェクトを解析した結果を統

© 2019 Information Processing Society of Japan

合することでこれらを区別できる可能性がある．実際に，4 つのプロジェクト全てで誤って機能スイッチマクロと判定されたマクロ変数も存在した．

謝辞　本研究の一部は科研費 (#15H02683，#18H03221) の助成を受けた．C/C++前処理命令の仕様や利用法に関する助言を頂いた東京工業大学修士 2 年の夏目雅槻氏に感謝いたします．

参考文献

[1] D. Jin, X. Qu, M. B. Cohen and B. Robinson: Configurations everywhere: implications for testing and debugging in practice, *Proc. ICSE Companion*, pp. 215–224 (2014).

[2] J. Liebig, S. Apel, C. Lengauer, C. Kästner and M. Schulze: An Analysis of the Variability in Forty Preprocessor-based Software Product Lines, *Proc. ICSE*, pp. 105–114 (2010).

[3] S. Apel, D. Batory, C. Kstner and G. Saake: *Feature-Oriented Software Product Lines: Concepts and Implementation*, Springer (2013).

[4] A. J. Ko, B. A. Myers, M. J. Coblenz and H. H. Aung: An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks, *IEEE TSE*, Vol. 32, No. 12, pp. 971–987 (2006).

[5] A. Sutton and J. I. Maletic: How We Manage Portability and Configuration with the C Preprocessor, *Proc. ICSM*, pp. 275–284 (2007).

[6] A. Kenner, C. Kästner, S. Haase and T. Leich: TypeChef: Toward Type Checking #ifdef Variability in C, *Proc. FOSD*, pp. 25–32 (2010).

[7] C. Kästner, K. Ostermann and S. Erdweg: A Variability-Aware Module System, *Proc. OOPSLA*, pp. 773–792 (2012).

[8] R. Tartler, D. Lohmann, J. Sincero and W. Schröder-Preikschat: Feature Consistency in Compile-time-configurable System Software: Facing the Linux 10,000 Feature Problem, *Proc. EuroSys*, pp. 47–60 (2011).

[9] S. Nadi and R. C. Holt: Mining Kbuild to Detect Variability Anomalies in Linux, *Proc. CSMR*, pp. 107–116 (2012).

[10] L. Vidács, R. Dévai, R. Ferenc and T. Gyimóthy: Developer Support for Understanding Preprocessor Macro Expansions, *Proc. ASEA/DRBC 2012*, pp. 121–130 (2012).

[11] S. Nadi, T. Berger, C. Kästner and K. Czarnecki: Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study, *IEEE TSE*, Vol. 41, No. 8, pp. 820–841 (2015).

[12] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Leßenich, M. Becker and S. Apel: Preprocessor-based variability in open-source and industrial software systems: An empirical study, *Empirical Software Engineering*, Vol. 21, No. 2, pp. 449–482 (2016).

[13] R. Queiroz, L. Passos, M. T. Valente, C. Hunsen, S. Apel and K. Czarnecki: The shape of feature code: an analysis of twenty C-preprocessor-based systems, *Software and Systems Modeling*, Vol. 16, No. 1, pp. 77–96 (2017).

[14] M. L. Collard, M. J. Decker and J. I. Maletic: SrcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration, *Proc. ICSM*, pp. 516–519 (2013).

[15] C. Dietrich, R. Tartler, W. Schröder-Preikshat and D. Lohmann: Understanding Linux Feature Distribution, *Proc. MISS*, pp. 15–20 (2012).

© 2019 Information Processing Society of Japan