Vol.2016-SE-191 No.6
2016/3/14

前処理命令の制御構造とその構造内のコード改変に関する調査

今西　洋二[1,a)]　渥美 紀寿[1,b)]　森崎 修司[1]　山本 修一郎[1]　阿草 清滋[2]

概要：前処理による分岐命令の制御構造に関してどのような構造が保守を困難にしているか明らかになっていない．本研究ではこれらを明らかにするため，前処理による分岐命令の制御構造とコードの改変にどのような関係があるのかについて調査した．オープンソースソフトウェアを対象に調査したところ，改変に関連する前処理条件の全体構造において，改変頻度が高い構造は，改変頻度が低い構造より分岐命令を複数含んでいることがわかった．また，改変に関連する前処理条件の集合のうち，複数リビジョンに渡って頻出する組は，複数の全体構造に含まれていることがわかった．

キーワード：前処理命令，制御構造，保守支援

1. はじめに

ソフトウェアの実行環境や利用する外部ライブラリ，利用者が要求する機能が多様化しているため，1 つのソフトウェアを構成するモジュール群が非常に多くなっている．実行環境には CPU アーキテクチャの違いや OS の違いがある．利用する外部ライブラリには，同じ機能を提供する複数のライブラリが開発されており，どのライブラリを利用するかは利用者に依存する．たとえば SSL では OpenSSL，NSS，GnuTLS などが開発されている．また，多くのソフトウェアでは基本機能に追加可能なモジュールが提供されており，それらを組み合わせることによって，利用者の要求に応じたソフトウェアを構築する．

C 言語によるマルチバリアントソフトウェアの開発では，実行環境や利用モジュールの違いを前処理命令によってソースコードを切り分けることによって実現されている．一般にこのようなソフトウェアでは，Configure スクリプト等によって利用者が指定したオプションに基づいて適切なマクロ定義が生成され，プリプロセッサは生成されたマクロ定義を利用して，選択された実行環境および利用モジュールによって実現されるバリアントのソースコードが生成される．前処理命令を用いたソースコードの切り分けは，実行環境や利用モジュールの違いを吸収するためだけでなく，デバッグやテスト，ログ生成などを目的として利用される．このことは前処理命令による生成されるコードの組み合わせを多くする原因の一つである．

Linux カーネルのバージョン 4.2 では, 約 15,200 のフィーチャで構成されている [6]．実行環境や利用するモジュールの多様化は今後も進むと考えられ，それらの組み合わせは複雑になり，ソースコードを切り分ける前処理命令の構造も複雑になる．このことによってマルチバリアントソフトウェアの開発が困難になっている．これを解決するために，ソースコードからフィーチャの依存関係を抽出する研究 [9] や，フィーチャの組み合わせにおける制約を抽出する研究 [4]，前処理命令に起因する不具合に関する研究 [5], [8] などが行われている．

本研究では，保守を困難とする前処理による分岐命令による制御構造を明らかにすることを目的とする．前処理命令によるソースコードの切り分けでは，複数の方法で同じ切り分けが可能である．例えば，ソースコード 1 とソースコード 2 はいずれも，`FLOAT` が定義されている場合には `float x;` が生成され，`FLOAT` と `DOUBLE` が定義されている場合には `double x;` が生成され，それ以外の場合には `int x;` が生成される．`#ifdef` の構造は異なっているが，すべてのバリアントにおいて同じコードが生成されるソースコードである．我々はこれらの `#ifdef` の構造の違いが保守性に影響を与えていると考えている．

そこで本研究では，前処理による分岐命令による制御構

[1] 名古屋大学
Nagoya University, Furo-cho, Chikusa-ku, Nagoya 464-8601, Japan
[2] 南山大学
Nanzan University, Yamazato-cho, Showa-ku, Nagoya 466-8673, Japan
[a)] imanishi.youji@d.mbox.nagoya-u.ac.jp
[b)] atsumi@nagoya-u.jp

ソースコード 1　#ifdef の例

```
1 #ifdef FLOAT
2     #ifdef DOUBLE
3     double x;
4     #else
5     float x;
6     #endif
7 #else
8     int x;
9 #endif
```

ソースコード 2　#ifdef の例

```
1 #ifndef FLOAT
2     int x;
3 #elif defined(DOUBLE)
4     double x;
5 #else
6     float x;
7 #endif
```

造とコードの改変にどのような関係があるのかについて調査した．頻繁に改変されるコードが依存する前処理による分岐命令に着目し，改変が多く行われている理由を明らかにする．また，前処理による分岐命令の制御構造の違いによって保守性に影響を与えるかどうかについて考察する．

以降，2 章では，関連研究を紹介し，本研究の位置付けを行う．3 章では，本研究で調査する対象となる前処理による分岐命令の制御構造について述べる．4 章では，調査内容について述べ，5 章でその結果について述べる．6 章では，本稿のまとめについて述べ，今後の課題を議論する．

2. 関連研究

本章では前処理命令を用いたマルチバリアントソフトウェアを効率良く開発するための手法と，前処理命令によって発生する不具合に関する既存研究について紹介する．

2.1 マルチバリアントソフトウェアの開発

マルチバリアントソフトウェアを効率良く開発するための開発環境として，C-CLR [7] や CIDE [1] などがある．

C-CLR [7] では，ソースコードを解析し，前処理命令による条件から定義するかしないかを選択可能なフィーチャを提示する．開発者は提示されたフィーチャから，着目しているフィーチャの組み合わせを指定し，その組み合わせによるコード断片を黒で表示し，指定されていないフィーチャに関連するコード断片をグレーで表示する．これによって，着目しているフィーチャの組み合わせに関連するコードを強調して表現する．

CIDE [1] では，開発者が着目しているフィーチャの組み合わせを指定することによって，それによって構成されるコードを前処理命令による条件を解釈して展開したコードを表示する．また，それぞれのフィーチャに関連するコードをフィーチャごとに色付けすることによって，個々のコード断片がどのフィーチャに関連しているかを示する．複数のフィーチャに関連しているコードは色を重ねて表現する．`#if` や `#ifdef` などの前処理命令による条件命令を削除し，コードを合成しているため，ソースコードの可読性が高くなっている．

これらのツールにより，特定のバリアントに対して効率良く開発することが可能となるが，複数のバリアントに跨ったコードの修正には向いていない．また，前処理条件の追加等により

2.2 前処理命令に依存する具合

前処理に依存する不具合には，与えられたフィーチャの組み合わせに対し，構文エラーや未定義の変数や関数の参照などのコンパイル時に検出される不具合の他，実行時に発生する不具合などがある．

文献 [2] では，リリースされたソフトウェアに前処理命令による構文エラーがあるか，コミットされたソースコードに前処理命令による構文エラーがあるか，どのようにして構文エラーとなるコードが埋め込まれるか，どれくらいの期間構文エラーとなるコードが存在しているかなどを調査している．その結果，リリース後のソフトウェアにも構文エラーとなるコードが存在していること，既存コードの修正によって構文エラーとなるコードが埋め込まれることなどが知見として得られている．

文献 [3] では，前処理命令によって未定義の変数や関数を参照する不具合について調査した結果を報告している．この調査により，これらの不具合は数年間修正されていないか，未修正のまま残っていることが明らかとなった．また，1 つか 2 つのフィーチャを選択するかしないかを切り替えるだけで，これらの不具合の 87%を検出することが可能だと報告している．

文献 [4], [5] は，Kconfig などの Configuration スクリプトで定義されるフィーチャの制約と，ソースコード中に前処理条件を用いて表現されたフィーチャ間の関係における不整合に関する研究である．この不整合の原因は Kconfig の設定不足の場合と，前処理条件の誤りの場合のいずれもあり得るが，いずれの場合においても，ソフトウェアの不具合につながるため，修正が必要である．

これらは前処理命令が引き起こす不具合に関する研究であるが，本研究では，前処理命令の構造が保守性に与える影響について調査する．

3. 前処理による分岐命令の制御構造

特定の箇所が依存する前処理による分岐命令によって表される条件文を前処理条件と呼ぶ．特定の箇所が依存する直前の前処理命令が`#if`，`#ifdef`，`#ifndef` のいずれかのとき，前処理条件は単純に，`#if`，`#ifdef`，`#ifndef` とその条件式による条件文となる．例えばソースコード 3 で，2 行目が依存する前処理命令を考えると，その前処理条件は`#if defined(__GNUC__)` となる．次に，特定の箇所が依存する直前の前処理命令が`#elif`，`#else` のときについて考える．4 行目が依存する前処理による分岐命令を考えると，直前の前処理による分岐命令は 3 行目の`#elif defined(_MSC_VER)` であるが，1 行目の `#if defined(__GNUC__)` にも依存するため，その前処理条件は`#if defined(__GNUC__)` - `#elif defined(_MSC_VER)` となる．また, 6 行目が依存する前処理による分岐命令を考えると，直前の前処理による分岐命令は 5 行目の`#else` であるが，1 行目の`#if defined(__GNUC__)`，3 行目の`#elif defined(_MSC_VER)` にも依存するため，その前処理条件は`#if defined(__GNUC__)` - `#elif defined(_MSC_VER)` - `#else` となる．

前処理による分岐命令が入れ子になっている箇所が依存する前処理条件には，上位の前処理条件も関連してくるため，これをコードに関連する前処理条件とする．例えばソースコード 4 で, 6 行目の`#include <unixlib.h>`が定義されている箇所の前処理条件は，`#if defined(__DECC)` - `#else` であり，上位の前処理条件である`#ifdef SYS_VMS`, `#ifndef W_OK` がコードに関連する前処理条件となる．

特定箇所が依存する前処理条件及びコードに関連する前処理条件のセットを存在条件と呼ぶ．例えばソースコード 4 で，10 行目の`#include <sys/file.h>`が定義されている箇所の前処理条件は，`#ifdef SYS_VMS` - `#elif !defined(SYS_VXWORKS)&&!defined(SYS_WINDOWS)` と，`#ifndef W_OK` の 2 条件である．そのため，これらを繋いだ`#ifndef W_OK` - `#ifdef SYS_VMS` - `#elif !defined(SYS_VXWORKS)&&!defined(SYS_WINDOWS)` が 10 行目の存在条件となる．

4. 制御構造とコードの改変に関する調査

本研究では，以下の 5 つの調査を行った．

(1) どのような前処理条件が改変に関与するのか
(2) 頻繁に改変箇所が依存する前処理条件を含む全体構造の内，どのような全体構造で頻繁に改変に関与しているのか
(3) 改変に依存する前処理条件の内，頻繁に出現する前処理条件の組はどのようなものか
(4) 頻繁に改変に関連する前処理条件の組を含む存在条件はどのような構成なのか
(5) 頻繁に改変に関連する前処理条件の組を含む全体構造の改変はどのようになっているのか

調査対象について 4.1 節で述べ，各調査について 4.2 節から 4.6 節にかけて，調査内容について述べる．

ソースコード 3 前処理による分岐命令の制御構造

```
1  #if defined(__GNUC__)
2  #define ALIGN32 __attribute((aligned(32)))
3  #elif defined(_MSC_VER)
4  #define ALIGN32 __declspec(align(32))
5  #else
6  #define ALIGN32
7  #endif
```

ソースコード 4 前処理による分岐命令の制御構造

```
1  #ifndef W_OK
2      #ifdef SYS_VMS
3          #if defined(__DECC)
4          #include <unistd.h>
5          #else
6          #include <unixlib.h>
7          #endif
8      #elif !defined(SYS_VXWORKS) \
9              && !defined(SYS_WINDOWS)
10     #include <sys/file.h>
11     #endif
12 #endif
```

4.1 調査対象

オープンソースソフトウェアでは，configure スクリプトを用いて，利用者が指定したオプションに基づき，プリプロセッサによってバリアントのソースコードが生成される．それに伴い生成された Makefile を用いてコンパイルを行う．そのため，オープンソースソフトウェアに着目し，その中で Git[*1] によって版管理がなされているものを対象とし，その公開されているデータから，リポジトリ内の全てのリビジョンに対して調査を行った．

また，調査するファイルは，各リビジョンで，改変された C ファイルのみに着目する．調査対象は，OpenLDAP[*2]，OpenSSL[*3]，nginx[*4]，apache[*5]，Dovecot[*6] の 5 つであり，これらは C 言語によって実装がなされている．

[*1] プログラムのソースコードなどの変更履歴を，記録や追跡するための分散型バージョン管理システム．
[*2] http://www.openldap.org/software/repo/openldap.git
[*3] git://git.openssl.org/openssl.git
[*4] https://github.com/nginx/nginx.git
[*5] git://git.apache.org/httpd.git
[*6] git://git.infradead.org/Dovecot/Dovecot-2.0

調査対象のリポジトリでのリビジョン数，最新リビジョンでの C ファイル数，調査したリビジョンの中で最も新しいリビジョンのハッシュ値 7 桁をそれぞれ表 1 に示す．

表 1 各 OSS のリビジョン数及び C ファイル数

	OpenLDAP	OpenSSL	nginx	apache	Dovecot
リビジョン数	21,832	14,762	5,522	28,051	12,629
C ファイル数	537	840	205	319	703
ハッシュ値	b6974cc	ca0004e	aa8fa81	cec27a8	c68fb0d

表 1 にて示した各オープンソースソフトウェアのリビジョンから以下の手順に従って，改変箇所の存在条件を取得した．

(1) リポジトリからコミットログを取得
(2) 最古のログから順に，それぞれのリビジョンで改変された C ファイルを取得
(3) それぞれの改変された C ファイルから改変行を取得
(4) 前処理による分岐命令の制御構造内に改変行が含まれる場合，改変行の存在条件を取得

手順（4）では，存在条件の他に，存在条件を構成する各前処理条件，その存在条件が取得されたリビジョン，その存在条件が取得されたファイル，その存在条件が含まれる全体構造，を付属する情報として取得した．また，存在条件は取得されたファイル，含まれる全体構造で区別し，各リビジョンにおいて単一となるように取得する．

4.2 調査 1：頻繁に改変箇所が依存する前処理条件

どのような前処理条件が改変に関連するのかについて調査するために，4.1 節で取得したデータセットから，前処理条件と，それが改変に関連したリビジョンを取得し，各前処理条件とリビジョン間の改変関係をプロットする．前処理条件は，ファイル及び出現行で区別をせず，全てのリビジョンを通して単一に取得する．

また，頻繁に改変箇所が依存する前処理条件がどのようなものであるかを取得するために，各前処理条件で最初の改変から最後の改変までのリビジョンでの改変頻度を算出し，複数リビジョンに渡って改変箇所が依存する前処理条件を抽出する．ただし，改変回数が少ない前処理条件に関しては考慮にいれない．

4.3 調査 2：全体構造の構成と改変間の関係

どのような全体構造に頻繁に改変箇所が依存しているかを調査するために，4.2 節で取得した頻繁に改変箇所が依存する前処理条件を含むデータセット内の全体構造に着目する．全体構造は，ファイル及び出現行で区別をせず，各リビジョンで全体構造を単一とする，また，各全体構造において，1 リビジョンで 1 回として改変回数を算出する．それぞれの全体構造で，最初の改変から最後の改変までのリビジョンで改変頻度を算出し，全体構造を抽出する．ただし，改変回数の少ない全体構造に関しては考慮に入れない．

抽出した全体構造の内，改変頻度が高い全体構造，改変頻度が低い全体構造には，構造内に分岐を複数含むような全体構造はそれぞれどれくらいの割合で含まれているのか調査する．また，改変頻度が高い全体構造と改変頻度が低い全体構造でそれぞれ見られる，分岐を複数含むような全体構造を各構造の分岐数で比較し，構成を分析する．

4.4 調査 3：関連して改変に関わる前処理条件

4.2 節で取得した，頻繁に改変箇所が依存する前処理条件が，他にどのような前処理条件と頻繁に改変に関与しているのかについて調査するために，頻繁に改変箇所が依存する前処理条件が改変に関与するリビジョンから，改変箇所が依存する他の前処理条件を抽出する．改変に依存する前処理条件の集合の内，複数リビジョンに渡って，頻繁に出現する前処理条件の組を求める．また，その出現回数を集計する．

4.5 調査 4：組の前処理条件を含む存在条件

4.4 節で取得した，改変に関連する前処理条件の組が同一の存在条件に含まれているかを調査するために，前処理条件の組の内，出現回数の多い前処理条件の存在条件を全て取得する．データセット内の存在条件はファイル及び出現行で区別をせず単一とし，4.4 節で取得した，改変に関連する前処理条件の組が，同じ存在条件に含まれているのかを確認する．

4.6 調査 5：組の前処理条件を含む全体構造

4.4 節で取得した，改変に関連する前処理条件の組の関係を調査するために，4.4 節で取得した，改変に関連する前処理条件の組を含む全体構造の改変回数を算出する．また，算出した組を含む全体構造の改変回数を，4.4 節で取得した，改変に関連する前処理条件の組における改変回数と比較をし，分析を行う．

5. 調査結果

5.1 調査 1 の結果

5 つの調査対象の内，一例として，OpenLDAP における前処理条件とリビジョン間の改変関係をプロットした図を図 1 に示す．

図 1 は，縦軸がリビジョン，横軸が前処理条件となっている．リビジョンは値が大きくなるに連れ最新のリビジョンとなる．前処理条件は最古のリビジョンから順に解析していった際の出現順に ID 付けを行っている．

図 1 を見ると，特定の前処理条件の改変プロットが複数リビジョンに連なって縦線のように見えている．このような傾向は，調査対象である他のソフトウェアの前処理条件とリビジョン間の改変関係図でも見られた．

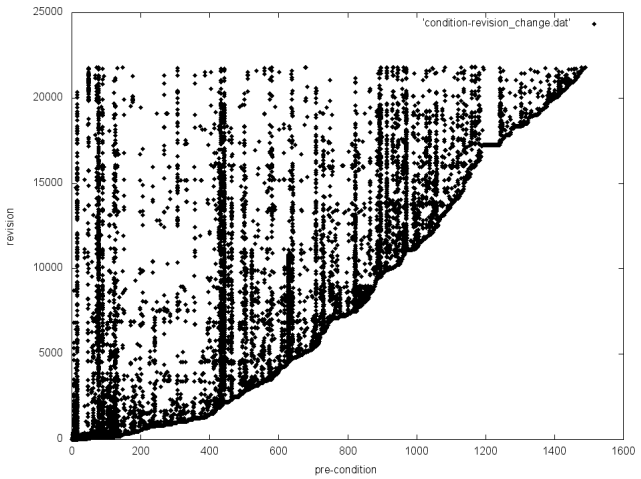

図 1 OpenLDAP より前処理条件-リビジョン間の改変関係

調査対象の各ソフトウェアで，最初に改変に関与したリビジョンから最後に改変に関与したリビジョンまでの改変頻度を算出し，複数リビジョンに渡って改変箇所が依存する前処理条件を抽出した．ただし，改変回数の低い前処理条件は除外した．閾値と抽出された頻繁に改変箇所が依存する前処理条件の数を表 2 に示す．

調査対象である全てのソフトウェアで，頻繁に改変箇所が依存する前処理条件は演算子を含むような条件では無く，単一のマクロによるものが大半を占めていることが分かった．一例として，OpenLDAP における頻繁に改変箇所が依存する前処理条件を表 3 に示す．

表 2 各 OSS の頻繁に改変箇所が依存する前処理条件数

	OpenLDAP	OpenSSL	nginx	apache	Dovecot
前処理条件数	21	16	13	9	10
閾値	100	100	50	50	50

5.2 調査 2 の結果

各ソフトウェアにおける，抽出した全体構造の数を表 4 に示す．

抽出した全体構造を改変頻度でソートし，上位 10 の全体構造と下位 10 の全体構造をそれぞれ改変頻度が高い全体構造，改変頻度が低い全体構造とし，全体構造の構成を分析した．`#if`，`#ifdef`，`#ifndef` を 1 つしか含まない，`#ifdef A - #else - #endif` 等のような構成をした全体構造を simple な構造とし，simple でない全体構造を complex な構造とする．改変頻度の高い全体構造，改変頻度の低い全体構造それぞれの中に simple な構造，complex な構造はどれくらい含まれるのか調査した結果を表 5 に示す．

表 5 では，high，low が改変頻度の高い，低いをそれぞれ意味しており，simple，complex がそれぞれ simple な構造，complex な構造を指している．

表 3 OpenLDAP より頻繁に改変箇所が依存する前処理条件

前処理条件	改変回数	改変頻度
NEW_LOGGING	680	0.1028
!(NEW_LOGGING)	636	0.0962
LDAP_SLAPI	455	0.0333
SLAPD_OVER_SYNCPROV	353	0.0330
HAVE_CYRUS_SASL	536	0.0275
HAVE_TLS	464	0.0235
0	466	0.0215
SLAPD_OVER_PROXYCACHE	228	0.0189
SLAPD_SQL	126	0.0156
LDAP_DEBUG	279	0.0137
SLAPD_OVER_RWM	165	0.0135
SLAPD_OVER_ACCESSLOG	117	0.0130
SLAPD_OVER_PPOLICY	139	0.0117
!(BDB_HIER)	152	0.0107
SASL_VERSION_MAJOR>=2	117	0.0104
SLAPD_ACI_ENABLED	133	0.0086
ENABLE_REWRITE	118	0.0075
LDAP_CONNECTIONLESS	159	0.0074
LDAP_PF_LOCAL	114	0.0066
!(HAVE_WINSOCK)	105	0.0050
!(_WIN32)	104	0.0048

表 4 各 OSS で取得した全体構造数

OpenLDAP	OpenSSL	nginx	apache	Dovecot
316	134	40	48	34

表 5 全体構造の simple な構造,complex な構造の数

		OpenLDAP	OpenSSL	nginx	apache	Dovecot
high	simple	0	1	5	2	5
	complex	10	9	5	8	5
low	simple	4	1	6	7	8
	complex	6	9	4	3	2

改変頻度が高い全体構造では，全てのオープンソースソフトウェアで complex な構造が多く含まれていた．改変頻度が低い全体構造では，ソフトウェアによっては complex な構造を多く含むものもあるが，それぞれのソフトウェアで改変頻度が高い全体構造と比較すると，改変頻度が高い全体構造より complex な構造は含まれる数が少ないことがわかった．

改変頻度の高い全体構造で見られる complex な構造と，低い全体構造で見られる complex な構造間に違いがあるのかを調査するために，complex な構造の構成を，分岐（`#if`，`#ifdef`，`#ifndef`）の数で比較した．OpenSSL，Dovecot の結果を表 6 に示す．

表 6（a）は OpenSSL で，（b）は Dovecot である．また，high，low がそれぞれ改変頻度が高い complex な構造と改変頻度が低い complex な構造を示す．ID は各全体構造に単一に付けられた識別番号であり，それぞれの全体構造での分岐数を示している．

表 6 前処理条件-リビジョン間の改変関係
（a）OpenSSL　（b）Dovecot

high		low	
ID	分岐数	ID	分岐数
312	15	242	2
538	57	301	2
457	6	102	2
528	63	244	3
532	55	162	2
512	12	201	2
470	4	107	2
94	5	47	2
636	38	18	3

high		low	
ID	分岐数	ID	分岐数
85	2	25	13
91	7	51	2
93	6		
52	7		
27	8		

OpenLDAP，OpenSSL，nginx では，改変頻度が高い complex な構造の方が，改変頻度が低い complex な構造よりも分岐数は大きい傾向にあった．Dovecot では，そのような傾向は見られ無かったが，リビジョン数が少なく，十分なデータが取れなかったことが原因に考えられる．apache では，上記のような傾向は少し見られるものの，追調査が必要であると考える．

追調査が必要であると考える．

改変頻度の高い全体構造の方が改変頻度の低い全体構造よりも，分岐を複数含む構造が多く存在する傾向が見られたが，分岐を複数含む構造は，ネストが深く複雑に分岐が含まれているのでは無く，同階層での分岐が多く含まれ，縦に広い全体構造の構成であった．そのため，前処理による分岐命令の入れ子や前処理条件に論理演算子が含まれていることによって，改変頻度が高いわけではなく，単純に全体構造が囲う行数が多いため，改変頻度が高くなっていることが考えられる．

5.3 調査 3 の結果

表 7 にて，改変に依存する前処理条件の組における出現回数を示す．表 7 は OpenLDAP であり，出現回数でソートしたときの上位 7 を抜粋して示す．

改変に依存する前処理条件の集合の内，頻繁に出現する前処理条件の組を見つけることが出来た．頻繁に改変に関連する前処理条件の組における出現回数が多い組の前処理条件同士は改変に関係があると考えられるが，単一のマクロによって構成される前処理条件同士の組が多い．そのため，同一リビジョンで改変される前処理条件間の関係に関して追調査を行った．その内容について，4.5，4.6 節で述べる．また，このような傾向は調査対象である他のソフトウェアでも見ることが出来た．

5.4 調査 4 の結果

表 8 では，頻繁に改変箇所が依存する前処理条件，それと改変に関連する前処理条件の組が同一存在条件に含まれている数，そして，同一リビジョンで改変に関与する前処理条件の数をそれぞれ行ごとに示す．また，表 8 に，OpenLDAP から取得した前処理条件に関して示す．

表 8 OpenLDAP より特定の前処理条件を含む存在条件

前処理条件	同一存在条件に含まれていた数	同一リビジョンで改変に関与する前処理場件数
NEW_LOGGING	47	54
!(NEW_LOGGING)	47	54
LDAP_SLAPI	12	40
SLAPD_OVER_SYNCPROV	3	17
HAVE_CYRUS_SASL	24	59
HAVE_TLS	24	61
0	39	63
SLAPD_OVER_PROXYCACHE	7	26
SLAPD_SQL	12	15
LDAP_DEBUG	26	72
SLAPD_OVER_RWM	5	17
SLAPD_OVER_ACCESSLOG	0	12
SLAPD_OVER_PPOLICY	2	16
!(BDB_HIER)	5	8
SASL_VERSION_MAJOR>=2	6	16
SLAPD_ACI_ENABLED	8	16
ENABLE_REWRITE	6	10
LDAP_CONNECTIONLESS	12	40
LDAP_PF_LOCAL	17	38
!(HAVE_WINSOCK)	9	35
!(_WIN32)	4	5

存在条件`#ifdef HAVE_CYRUS_SASL - #ifdef SASL_VERSION_MAJOR>=2` では，抽象度の高い HAVE_CYRUS_SASL と，それの具体的なバージョンを示す SASL_VERSION_MAJOR>=2 のような，同存在条件に含まれている前処理条件間の機能的な繋がりが見える．これは同存在条件に含まれている組の一部で確認することができた．確認できなかった組でも，マクロ名からは判断出来ないが，同存在条件に含まれているため，機能的な繋がりがあるものと考えられる．

このような傾向は調査対象の他のソフトウェアでも見ることができ，存在条件の構成を見ることで，機能的な繋がりがある前処理条件間の関係を取得出来る事が分かった．また，図 2 の 4 行目`#ifdef B` と，8 行目`#ifdef C` のような，全体構造の同階層にある入れ子の位置の前処理条件等に関しては，同じ存在条件内に含めることは出来ず，前処理条件間の関係を判断することが出来ないことが分かった．

5.5 調査 5 の結果

改変に関連する前処理条件の組を含む全体構造の改変回数と，改変に関連する前処理条件の組が同一のリビジョンで改変される回数の比較結果を表 9 に示す．

表 9 は, データセットは OpenLDAP であり, SLAPD_SQL を含む組の結果である．表 9 には，SLAPD_SQL と同一リビジョンで改変に関与する前処理条件があり, SLAPD_SQL と改変に関連する前処理条件の組を含む全体構造の改変回

表 7 OpenLDAP より改変に依存する前処理条件の組における出現回数 (上位 7)

頻繁に改変に関わる前処理条件	関連して改変に関わる前処理条件	組での出現回数
NEW_LOGGING	!(NEW_LOGGING)	611
HAVE_CYRUS_SASL	SASL_VERSION_MAJOR>=2	116
HAVE_CYRUS_SASL	HAVE_TLS	84
HAVE_CYRUS_SASL	NEW_LOGGING	73
0	NEW_LOGGING	67
LDAP_SLAPI	NEW_LOGGING	56
HAVE_TLS	NEW_LOGGING	52

```
1  ...
2  #ifdef A
3  /* A 向けコード */
4  # ifdef B
5  /* A 且つ B 向けコード */
6  # endif
7  /* A 向けコード */
8  # ifdef C
9  /* A かつ C 向けコード */
10 # endif
11 /* A 向けコード */
12 #endif
13 ...
```

図 2 同階層に前処理による分岐命令を複数含むソースコード

数と，SLAPD_SQL と改変に関連する前処理条件の組が同一リビジョンで改変される回数を示す．

表 9 SLAPD_SQL を含む組における各改変回数

前処理条件	全体構造の改変回数	組における改変回数
HAVE_TLS	0	5
HAVE_CYRUS_SASL	0	7
SLAPD_SQL_DYNAMIC	5	5
LDAP_DEBUG	5	10
0	17	18
NEW_LOGGING	4	17
!(NEW_LOGGING)	4	16
!(BACKSQL_REALLOC_STMT)	7	7
BACKSQL_REALLOC_STMT	17	17
BACKSQL_TRACE	11	11
LDAP_SLAPI	0	10
SLAPD_SQL==SLAPD_MOD_DYNAMIC	5	5
BACKSQL_ARBITRARY_KEY	11	11
!(BACKSQL_ARBITRARY_KEY)	11	11

改変に関連する前処理条件の組の一部では，その組を含む全体構造の改変回数が，組が同一リビジョンで改変に関与する回数より少ない，といった傾向が見られた．このような傾向は，5 つのオープンソースソフトウェアのそれぞれで取得した，出現頻度の高い前処理条件の組で多数見られた．表 10 は，各ソフトウェアで取得された，頻繁に改変箇所が依存する前処理条件の数と，それを含む組の一部で上記のような傾向があった数を示す．

頻繁に改変に関連する前処理条件の組を含む全体構造の改変回数が，改変に関連する前処理条件の組が同一リビジョンで改変に関与する回数より少ないということは，組の前処理条件が異なる全体構造にそれぞれ含まれており，その異なる全体構造が同リビジョンで改変されていることを指す．異なる全体構造間に含まれているが，同一リビジョンで改変される理由として，次の 2 点が考えられる．

- 一度のコミットに複数の目的が存在
- 前処理条件間に何らかの繋がりが存在

本調査では，Git により版管理しているオープンソースソフトウェアを対象にしている．Git は，一度の改変で一つの目的を達成しやすいような版管理システムになっている．そのため，前者のような一度のコミット時に複数の目的で改変することは起こり辛い．

異なる全体構造中に含まれる前処理条件間になんらかの繋がりがあるならば，一方の前処理条件が関与する箇所を改変する際に，もう一方の全体構造中にある前処理条件が関与する箇所も改変しなければならない可能性が生じる．これは，改変のし忘れが起こりやすいと考えられ，一つの全体構造内で前処理条件間の繋がりが完結している場合より，保守が困難な構造であると考えられる．しかし，まだ異なる全体構造間の繋がりに関しては明確にできておらず，追調査する必要がある．

表 10 各 OSS の前処理条件数

	OpenLDAP	OpenSSL	nginx	apache	Dovecot
傾向がある数	20	16	13	9	10
前処理条件数	21	16	13	9	10

6. おわりに

6.1 まとめ

保守を困難とする前処理による分岐命令の制御構造を明らかにすることを目的とし，それを実現するために，前処理による分岐命令の制御構造とコードの改変にどのような関係があるのか調査を行った．

調査対象である 5 つのオープンソースソフトウェアそれ

ぞれで，データセットを取得し，どのような前処理条件が改変に関与するのか調査を行った結果，複数リビジョンに渡って改変箇所が依存する前処理条件を見つけ，抽出した．

抽出した前処理条件を含む全体構造を分析した結果，調査対象全てのソフトウェアで，分岐を複数含む全体構造は改変頻度が高い構造に多く含まれる傾向があった．分岐を複数含む全体構造の内, 改変頻度の高い全体構造と低い全体構造間を分岐数で比較した結果，OpenLDAP，OpenSSL，nginx では，改変頻度が高い全体構造の方が，改変頻度が低い全体構造よりも分岐数が多い傾向にあることが見られた．Dovecot では，上記のような傾向は見られなかったが，リビジョン数が少なく，十分なデータが取れなかったことが原因に考えられる．apache では，少し傾向は見られるものの，追調査が必要であると考える．

頻繁に改変箇所が依存する前処理条件と，それと同一のリビジョンで改変に関与する前処理条件の組を含む存在条件の構成を見ることで，前処理条件間の機能的な繋がりを確認することが出来た．

改変に依存する前処理条件の組の内．出現頻度の高い組での出現回数が，その組を含む全体構造の改変回数より多い傾向が見られることが分かった．2 つの改変回数の差分は，異なる全体構造に含まれている前処理条件が同一リビジョンで改変に関与する回数を示しており，異なる全体構造間に何らかの繋がりが存在することがわかった．異なる全体構造に含まれる前処理条件同士に何らかの繋がりがあるならば，1 つの全体構造内で前処理条件間の繋がりが完結しているより，保守が困難な構造であると言えるが，異なる全体構造間の繋がりに関しては追調査が必要である．

6.2 今後の課題

今回調査した，ソフトウェアの一部では，分岐を複数含む改変頻度の高い全体構造と改変頻度の低い全体構造間で，分岐数で差は出たが，一方で，分岐数で差の無い構造間の関係も存在した．そのため，囲う行数等，比較できる情報を拡張し，分岐を複数含む改変頻度の高い全体構造と改変頻度の低い全体構造間の関係を明確にすることで，改変頻度が高くなる全体構造の特徴を見つけることを目指す．

異なる全体構造間に何らかの繋がりがあり，これが保守を困難としている要因の一部であることが考えられたが，異なる全体構造間の何らかの繋がりに関しては明確にできていない．一方の全体構造内で宣言されたマクロ（`#define`，`#undef`）が，もう一方の全体構造の条件式で用いられるなどの繋がりが考えられ，宣言されたマクロの使用の仕方等，評価できる情報を拡張し，異なる全体構造間の関係を明らかにすることで，どのような構造間の繋がりが保守を困難としているのかを見つけることを目指す．

謝辞 本研究は JSPS 科研費 24300006，15K15973，15H02683 の助成を受けた．

参考文献

[1] Kästner, C., Apel, S. and Kuhlemann, M.: Granularity in Software Product Lines, *Proceedings of the 30th International Conference on Software Engineering*, pp. 311–320 (2008).
[2] Medeiros, F., Ribeiro, M. and Gheyi, R.: Investigating Preprocessor-based Syntax Errors, *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, pp. 75–84 (2013).
[3] Medeiros, F., Rodrigues, I., Ribeiro, M., Teixeira, L. and Gheyi, R.: An Empirical Study on Configuration-related Issues: Investigating Undeclared and Unused Identifiers, *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pp. 35–44 (2015).
[4] Nadi, S., Berger, T., Kästner, C. and Czarnecki, K.: Mining Configuration Constraints: Static Analyses and Empirical Results, *Proceedings of the 36th International Conference on Software Engineering*, ACM, pp. 140–151 (2014).
[5] Nadi, S., Dietrich, C., Tartler, R., Holt, R. C. and Lohmann, D.: Linux Variability Anomalies: What Causes Them and How Do They Get Fixed?, *Proceedings of the 10th Working Conference on Mining Software Repositories*, pp. 111–120 (2013).
[6] Rothberg, V., Dintzner, N., Ziegler, A. and Lohmann, D.: Feature Models in Linux: From Symbols to Semantics, *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, pp. 65–72 (2016).
[7] Singh, N., Gibbs, C. and Coady, Y.: C-CLR: A Tool for Navigating Highly Configurable System Software, *Proceedings of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software* (2007).
[8] Tartler, R., Lohmann, D., Sincero, J. and Schröder-Preikschat, W.: Feature Consistency in Compile-time-configurable System Software: Facing the Linux 10,000 Feature Problem, *Proceedings of the Sixth Conference on Computer Systems*, pp. 47–60 (2011).
[9] 渥美紀寿，小林隆志，阿草清滋：プリプロセス命令の制御構造を利用したフィーチャ間の依存性解析，Vol. 112, No. 373, pp. 67–72 (2013).