空間統計モデルを用いたコード修正ホットスポットの検出に関する考察

佐野 直紀[1]　岡村 寛之[1]　土肥 正[1]

概要：既に存在するシステムをもとにして新規のシステム開発を行う派生開発では，もととなるシステムの設計やふるまいを正しく理解する必要がある．一方，派生開発では整理された設計書などが存在しないこともあり，ソースコードそのものからシステムを理解する場面が少なからず生じる．ソースコードからシステムの理解を行うためには，現在のソースコードだけではなく，それらが修正されてきた履歴を確認することも重要である．しかしながら，リポジトリに格納された膨大な修正履歴を詳細に調査することは現実的に不可能であり，修正履歴に関する何らかの要約が必要となる．
本論文では，ソースコードの修正履歴から修正が集中している部分（ホットスポット）を統計的に検出にする手法の検討を行う．ホットスポットは，修正履歴に関するある種の要約であり，ホットスポットを特定することで，その期間で行われた主要な修正とその経緯を容易に読み取ることができる．特に，本論文ではモジュール間の依存関係を空間（地図）と見立てた空間統計モデルを利用することで，修正履歴頻度だけによる単純な検出よりも，意味のある単位でホットスポットを検出することを可能とする．

A Note on Identifying Hot Spots of Program Modifications with Statistical Spatial Models

Abstract: In the derivational development, it is important to recognize the design and behavior of system which is used as original program. On the other hand, it is not always that there exist documents of system design. In such situation, we have to recognize the design and behavior of system only from existing source codes. When we read source codes, it is also significant to check the history of source codes. However, it is not practical to check all the history of source codes, and it requires a certain summary on the history of source codes.
In this paper, we propose a method to find hot spots on program modification using spatial statistics. In our approach, the identified hot spots are recognized as a summary of program modifications. The hot spots enables us to recognize a major modification and its process for a given duration. In particular, we discuss a statistical spatial model where the spatial information is given by to the module dependencies, and the model gives us a useful information compared to the simple hot spot by change frequency.

1. はじめに

近年のソフトウェア開発では新規でソフトウェアを開発することは稀であり，開発のほとんどは派生開発と呼ばれる形態をとることが多い．派生開発では，既存のソフトウェアをもとにして新しい機能付け加えたり，別のソフトウェアへの流用を行う．これは，従来からあるプログラムの部品化と似た概念であるが，部品目的として作成されたプログラムのように整備された API（Application Programming Interface）を備えていないため，派生開発では一般的にプログラム品質が低下すると言われている．

しかし，新規開発のコストと比較すると低コストでソフトウェアが構築できるため，現実的に数多くのソフトウェアが派生開発で作成されている．派生開発において品質を向上するためには既存プログラムの理解が重要である．特にソフトウェアの設計指針や修正履歴・理由が記録された設計文書は既存プログラムを理解するために必要不可欠であるが，実際の派生開発ではそのような設計文書が紛失あるいは最初から作成されていないことが多い．一方，現在のソフトウェア開発ではリポジトリと呼ばれるデータベー

[1] 広島大学大学院工学研究科
東広島鏡山 1-4-1, 739-8527

スでソフトウェアの修正履歴などを管理している．そのため，設計文書がないソフトウェアについてはソースコードやリポジトリに残されている情報からプログラムの理解を行うことが必要となってくる．しかしながら，詳細な情報から巨視的なプログラムの理解を行うことは困難である．特に，プロジェクトが大きくなると修正が多岐にわたり，プログラム修正履歴が雑多に混ざっていることも多いことから開発者が修正履歴だけから修正理由を推測することは不可能と言って良い．

そこで，本研究では空間統計量を利用することで，プログラムの詳細な修正履歴からプログラム修正ホットスポットを検出する手法の提案を行う．この技術を用いることにより，ある時点で修正が集中しているソフトウェアモジュールが容易に検出できる．これは，既存プログラムの理解において，プログラム全体ではなく修正が集中したモジュールだけに着目することで理解するための工数を低減することができると同時に，プログラムの理解を促進させることで派生開発などのソフトウェア品質の向上が見込まれる．

本論文の構成は以下の通りである．まず第 2 節において，一般的な空間においてホットスポットを検出するための空間スキャン統計の紹介を行う．特に，空間スキャン統計の統計量ならびに領域選択に関する説明を行う．次に，3 節では 2 節で述べた空間スキャン統計をプログラム修正ホットスポットに適用するための議論を行う．4 節では，二つの GitHub リポジトリから履歴にあるバージョン間のプログラム修正ホットスポットの検出を行う．5 節では，本論文のまとめと今後の課題について述べる．

2. 空間スキャン統計

空間統計学 [1] とは空間データ（位置データと属性値）に対する統計手法の総称であり，疫学などの地理上における疾病の発生や流行状況の分析，あるいは画像データの復元などに応用されている．ここでは，Kulldorff [1], [2] による空間上のホットスポットを検出するための空間スキャン検定について紹介する．

2.1 検定統計量

空間スキャン検定とは与えられた空間データからあるイベントが集中している場所（ホットスポット）をさがす手法である．いま空間データとして「領域」の集合を考える．各領域にはそれぞれ位置データと属性があり，位置データにより各領域が隣接しているかどうかの判断が可能である．また，一般的に属性はその領域で何らかのイベントが発生した回数と領域の大きさ（面積やイベント発生元となるイベント回数など）であることが多い．

空間データの全体集合を G として，G 内で隣接した要素からなる領域群（ひとかたまりの領域）を $Z \subseteq G$ とする．空間スキャン統計量は「領域群 Z がホットスポットではない」という帰無仮説と，「領域群 Z がホットスポットである」という対立仮説における尤度比（の対数）で定義される．具体的に，領域群 Z^c を Z の補集合（$G = Z \cup Z^c$, $Z \cap Z^c = \phi$）とするとき，帰無仮説 H_0 と対立仮説 H_1 は以下のように与えられる．

$$H_0 : P_Z = P_{Z^c} \tag{1}$$

$$H_1 : P_Z > P_{Z^c} \tag{2}$$

ここで，P_Z と P_{Z^c} はそれぞれ Z と Z^c でイベントが発生する確率を表す．

イベントの発生回数に対する分布としては，二項分布ならびにポアソン分布が利用されることが多い．いまイベントの発生回数が二項分布に従うものと仮定すると，空間スキャン統計量は H_1 の条件下での最大尤度を H_0 の条件下での最大尤度で割った値の対数（尤度比の対数）で定義される．具体的に，ある領域 $i \in G$ の属性値としてイベント発生回数 c_i と大きさ s_i を考える．このとき，対立仮説 H_1 ($P_Z \neq P_{Z^c}$) での尤度 L_1 は次のようになる．

$$L_1(P_Z, P_{Z^c}) = L(Z)L(Z^c). \tag{3}$$

ここで $L(Z)$, $L(Z^c)$ は領域群 Z, Z^c に対する尤度であり，それぞれ以下のようになる．

$$L(Z) = \prod_{i \in Z} \binom{s_i}{c_i} {P_Z}^{c_i} (1 - P_Z)^{s_i - c_i} \tag{4}$$

$$L(Z^c) = \prod_{i \in Z^c} \binom{s_i}{c_i} {P_{Z^c}}^{c_i} (1 - P_{Z^c})^{s_i - c_i}. \tag{5}$$

また，帰無仮説 H_0（$P_Z = P_{Z^c} = P$）のもとでの尤度 L_0 は以下のようになる．

$$L_0(P) = \prod_{i \in G} \binom{s_i}{c_i} P^{c_i} (1 - P)^{s_i - c_i}. \tag{6}$$

任意の領域群 $Z \subseteq G$ においてイベントの回数の操作ならびに大きさの総和を次のように定義する．

$$C(Z) = \sum_{i \in Z} c_i, \quad S(Z) = \sum_{i \in Z} s_i. \tag{7}$$

このとき，対立仮説 H_1 のもとでの尤度を最大にする P_Z ならびに P_{Z^c} は次のように与えられる．

$$\hat{P}_Z = \frac{C(Z)}{S(Z)}, \quad \hat{P}_{Z^c} = \frac{C(Z^c)}{S(Z^c)}. \tag{8}$$

さらに，帰無仮説 H_0 のもとで尤度を最大にする P は

$$\hat{P} = \frac{C(G)}{S(G)} = \frac{C(Z) + C(Z^c)}{S(Z) + S(Z^c)} \tag{9}$$

となる．領域 Z に対する検定統計量 (空間スキャン統計量) は H_1 と H_0 の尤度比の対数（LLR: log-likelihood ratio）で与えられる．

$$
\begin{aligned}
LLR(Z) =& \log \frac{L_1(\hat{P}_Z, \hat{P}_{Z^c})}{L_0(\hat{P})} \\
=& C(Z) \log \left(\frac{C(Z)}{S(Z)} \right) + C(Z^c) \log \left(\frac{C(Z^c)}{S(Z^c)} \right) \\
&+ (S(Z) - C(Z)) \log \left(1 - \frac{C(Z)}{S(Z)} \right) \\
&+ (S(Z^c) - C(Z^c)) \log \left(1 - \frac{C(Z^c)}{S(Z^c)} \right) \\
&- C(G) \log \left(\frac{C(G)}{S(G)} \right) \\
&- (S(G) - C(G)) \log \left(1 - \frac{C(G)}{S(G)} \right) \quad (10)
\end{aligned}
$$

ただし，H_1 のもとでは $P_Z > P_{Z^c}$ を仮定しているため $\hat{P}_Z > \hat{P}_{Z^c}$ とならない場合は $LLR(Z) = 1$ とする．空間スキャン検定では，検定統計量 $LLR(Z)$ を最も大きくする領域群 Z をホットスポットの候補として検定を行う．$LLR(Z)$ と用いて検定を行う場合，この統計量が従う分布を解析的に導出することが難しいため，通常はモンテカルロ検定を用いて p 値の計算を行う．帰無仮説 H_0 が棄却された場合，領域群 Z におけるイベント発生確率が他の領域群のイベント発生確率よりも有意に差があると判断し領域群 Z をホットスポットとする．

2.2 領域 Z の探索

領域群 Z の候補は集合 G の連続した要素で構成されるが，集合 G の要素数が増加すると領域 Z の候補が爆発的に増加する．そこで，実際の空間スキャン統計量を用いた検定では領域 Z を探索を効率化するための手法が用いられる．ここでは Circular Scan（CS）[1] と Echelon Scan（ES） [3] と呼ばれる手法の紹介を行う．

2.2.1 Circular Scan

Circular Scan（CS）は，空間 G 内の一つの領域 Z_0 を選び，その領域を中心とした同心円によって選択される領域を候補 Z として，ホットスポットの検出を行う手法である．具体的には以下のような手順に従って行われる．

(1) 与えられた空間データのすべての領域 for all $i \in G$ について以下 (2) から (5) の操作を行う．
(2) 領域 $Z = \{i\}$ とする．
(3) 終了条件が満たされるまで以下の (4), (5) の操作を繰り返す．
(4) $LLR(Z)$ を計算する．
(5) 領域 i と位置が最も近い領域 j を領域群 Z の要素に加える．$Z = Z \cup \{j\}$.
(6) $LLR(Z)$ が最大となる Z をホットスポットの候補とし，検定を行う．

2.2.2 Echelon Scan

Echelon Scan (ES) では Echelon 解析と呼ばれる空間データの位相的な構造を階層構造（Echelon デンドログラム）によって表す手法によって，空間データの構造を見つけ，その構造にもとづいてホットスポットの候補 Z を決定する．

Echelon 解析では, 空間データの階層構造を見つけるために空間データ上で連結している周辺領域の値よりも高い値からなる領域の集合であるピークとピークを形成する集団には属さず, 2 つ以上の階層集団の根を連結するための土台となる下位の階層集団であるファウンデーションと呼ばれる集合を見つける．ピークとファウンデーションの検出については, 以下のような手順で行う．

ピークの検出手順

(1) 全領域の内で, 最も値（c_i/s_i）が高い領域 (i) を第 1 ピーク（P_1）の要素とする．
(2) P_1 に隣接する領域の内で最も高い値を持つ領域 j を第 1 ピークの要素の候補とする．
(3) 領域群 $\{i, j\}$ に隣接する領域に j よりも大きい値を持つものがなければ, j を P_1 に加え（2）に戻る．そうでなければ j を加えず P_1 を決定する．
(4) 全領域の内で, すでにピークの要素となっている領域を除いて最も値が高い領域 k がその周辺の領域よりも値が高ければ第 2 ピーク (P_2) に加える．そうでなければ, ピークの検出を終了する．
(5) P_2 に隣接する領域の内で最も高い値を持つ領域 l を第 2 ピークの要素の候補とする．
(6) 領域群 $\{k, l\}$ に隣接する領域に l よりも大きい値を持つものがなければ, l を P_2 に加え 4 に戻る．そうでなければ l を加えず P_2 を決定し, 以降のピークを第 2 ピークの検出と同様な手順で検出する．

ファウンデーションの検出手順

(1) すでに見つかったピーク, ファウンデーションに含まれない領域の内, 値が最大となる領域 n を見つける．
(2) n と隣接するピークまたはファウンデーションを合わせた集合がファウンデーションと定義される．(n が P_1, P_2 と隣接していた場合, 第 1 のファウンデーション F_1 は $\{n, P_1, P_2\}$ と定義される．)
(3) F_1 に隣接する領域の内, 最大の値を持つ領域 m を F_1 の要素の候補とする．
(4) $\{F_1, m\}$ に隣接する領域の内, m よりも値が高くなるものがなければ, m を F_1 に加える．そうでなければ, F_1 を決定し, F_1 と同様の手順で, 別のファウンデーションを見つける．
(5) 全ての領域がファウンデーションの要素となった時点でファウンデーションの検出を終了する．

上記のような手順で Echelon 解析を行った結果, 明らかになったピークとファウンデーションによって構成された階層構造を利用し, 以下のようにホットスポットの候補 Z を決定する.

ホットスポットの候補 Z の決定

(1) P_1 の要素の内で最大値をとる領域 i を $Z = \{i\}$ とし, LLR を計算する.
(2) P_1 の要素の内で i の次に大きい値をとる領域 j を $Z = \{i, j\}$ とし,$LLR(Z)$ を計算する. 以降, 同様にして値が高い順に領域を Z に加え, Z が更新されるたびに $LLR(Z)$ を計算する.
(3) P_1 と同様にして別のピークに対しても Z を構成し, $LLR(Z)$ を計算する.
(4) ファウンデーションについても上記の P_1 のように Z を構成し, $LLR(Z)$ を計算する.
(5) LLR を計算した領域群の内で LLR が最大となるものをホットスポットの候補 Z として決定する.

3. プログラム修正履歴に対する空間スキャン統計量

ここでは, 空間スキャン統計量を用いたプログラム修正ホットスポットの検出を考える. 空間スキャン検定をプログラムの修正履歴に適用させるためには, プログラムの修正履歴のデータを空間データとしてとらえることが必要になる. つまり, 空間データの各領域の単位, 属性値, 位置データを定義する必要がある.

本論文では, 一つの領域を一つのファイルとして, 位置データについては, ファイルに記述されているプログラムの間の呼び出し関係で定義する. つまり, あるファイルから別のファイルの関数（メソッド）などを呼び出している場合,「これらのファイルが隣接してる」とし, 任意のファイル間の距離は呼び出しに関する最小ホップ数で定義する.

また, 属性値については, イベント発生回数としてファイル上のプログラム修正行数, 大きさとしてファイル上のプログラム行数として定義する. ただし複数のコミットをまとめた際の修正行数ならびにプログラム行数については, 以下の二つのパターンについて考察する.

(i) 各コミットの総コード行数の合計値を利用ある n 回のコミットによってプログラムが変更された場合, ファイル i に対する修正行数 c_i ならびに大きさ s_i を次のように定義する.

$$c_i = \sum_{l=1}^{n}(a_{i,l} + d_{i,l}), \quad s_i = \sum_{l=1}^{n}(s^-_{i,l} + s^+_{i,l}). \qquad (11)$$

ここで $a_{i,l}$ と $d_{i,l}$ はそれぞれ l 番目のコミットで追加および削除された行数, $s^-_{i,l}$ と $s^+_{i,l}$ は l 番目のコミット直前および直後の総コード行数を表す.

(ii) 各コミットの総コード行数の最大値を利用ある n 回のコミットによってプログラムが変更された場合, ファイル i に対する修正行数 c_i ならびに大きさ s_i を次のように定義する.

$$c_i = \sum_{l=1}^{n}(a_{i,l} + d_{i,l}), \quad s_i = n \times \max_{l=0,\ldots,n}\{s^-_{i,l}, s^+_{i,l}\}. \qquad (12)$$

ここで $a_{i,l}$ と $d_{i,l}$ はそれぞれ l 番目のコミットで追加および削除された行数, $s^-_{i,l}$ と $s^+_{i,l}$ は l 番目のコミット直前および直後の総コード行数を表す.

上述の (i) の定義は, n 回のコミット中に新規でファイルが追加された場合, それ以前の修正行数ならびに総コード行数が 0 となる. そのため, 最後のコミットで新規追加されたファイルの修正割合 c_i/s_i が高くなる傾向になる. (ii) の定義では, 各コミットでの総コード行数の和を用いる代わりに, 総コード行数の最大値を用いている.

4. 実験

4.1 実験 1

リポジトリのプログラム修正履歴に対し, 空間スキャン統計量によるプログラム修正ホットスポットの検出を行い, 検出されたホットスポットについて考察する. 実験では Java のプログラムを対象とし, GitHub 上の二つのリポジトリからある特定のソフトウェアバージョン間における修正履歴を取得し, ここで提案する空間スキャン検定によるホットスポット（修正が集中するプログラム）についての検出をおこなった. 実験 1 では比較的小規模な Java プロジェクトについて特定のバージョン間におけるホットスポット検出を行う. 対象とするリポジトリは `https://github.com/okamumu/JSPetriNet.git` であり, Java によるペトリネット解析ツールであり, 著者らのグループにより開発されたものである. 対象リポジトリについてバージョン 0.9.10 とバージョン 0.10.0 間のコミットデータを収集した. 収集した修正履歴等の情報を表 7 に示す.

収集したデータに対して, 空間スキャン検定によるホットスポットの検出を行う. 領域 Z の特定には CS ならびに ES を適用した. CS の領域群拡大の終了条件には「領域群の合計のコード行数が全体の 50 %になるまで」という条件を用いた. さらに, 属性値の定義として前述の (i) および (ii) の両方を適用し比較を行う. 表 2 は CS ならびに ES および二通りの属性値定義 (i), (ii) を適用した時, LLR が最も高くなった領域におけるファイル数, LLR ならび領域の属性値 $C(Z)/S(Z)$ を示している. なお, これらの領域はすべて, サンプル数は 10000 個のモンテカルロ検定を通じて, 有意水準 1% で棄却（「ホットスポットである」と検出）されている. また, 表 3 から表 6 はそれぞれの手法ならびに属性値の定義で検出されたホットスポッ

表 1 JSPetriNet リポジトリの情報（バージョン 0.9.10〜バージョン 0.10.0）

情報	数値
総ファイル数	94
コミット数	3
変更があったファイルの数	13

表 2 検出されたホットスポットの LLR および属性値 (JSPetriNet).

手法	属性値の定義	$LLR(Z)$	$C(Z)/S(Z)$	ファイル数
CS	(i)	4138.1	0.075	3
	(ii)	1991.3	0.024	4
ES	(i)	4138.1	0.075	3
	(ii)	2559.1	0.024	5

トのファイル群を示している．各表では，ファイル毎の属性値 c_i/s_i と全体のファイル中における属性値の順位を表している．

表 3 JSPetriNet のホットスポット（CS, 属性 (i)）.

ファイル名	c_i/s_i	順位
jspetrinet/parser/JSPNLBaseListener.java	0.035	5
jspetrinet/parser/JSPNLListener.java	0.040	2
jspetrinet/parser/JSPNLParser.java	0.044	1

表 4 JSPetriNet のホットスポット（CS, 属性 (ii)）.

ファイル名	c_i/s_i	順位
jspetrinet/parser/JSPetriNetParser.java	0.032	1
jspetrinet/parser/JSPNLBaseListener.java	0.024	3
jspetrinet/parser/JSPNLListener.java	0.024	2
jspetrinet/parser/JSPNLParser.java	0.024	5

表 5 JSPetriNet のホットスポット（ES, 属性 (i)）.

ファイル名	c_i/s_i	順位
jspetrinet/parser/JSPNLBaseListener.java	0.035	5
jspetrinet/parser/JSPNLListener.java	0.040	2
jspetrinet/parser/JSPNLParser.java	0.044	1

表 6 JSPetriNet のホットスポット（ES, 属性 (ii)）.

ファイル名	c_i/s_i	順位
jspetrinet/parser/JSPetriNetParser.java	0.032	1
jspetrinet/parser/JSPNLBaseListener.java	0.024	3
jspetrinet/parser/JSPNLLexer.java	0.024	4
jspetrinet/parser/JSPNLListener.java	0.024	2
jspetrinet/parser/JSPNLParser.java	0.024	5

小規模なリポジトリに対する実験結果から，属性値の定義 (i) を用いた CS と ES が全く同じファイル群をホットスポットとして検出していることがわかる．これに対して，属性値の定義 (ii) を用いた場合，ES では JSPNLLexer.java がホットスポットの要素であったが，CS ではこのファイルがホットスポットとして検出されていない．これは，属性値の定義 (i) と (ii) の違いによって，各ファイルの c_i/s_i の値が変化したことで，エシェロン解析で得られるエシェロンデンドログラムの構造が変化したことと，c_i, s_i の値の変化によって，最終的に求められる LLR の値が変化したことに原因があると考えられる．また，ホットスポットとして検出された集合について見ていくと，CS，ES どちらの手法でも c_i/s_i の値が大きいファイルがホットスポットの要素となっていることがわかる．さらに，具体的な変更内容を見ると，JSPetriNet のバージョン 0.9.10 からバージョン 0.10.0 ではパーサのプログラムを JavaCC から ANTLR4 へ移行しており，その修正に関するファイル（JSPNLPaser や JSPNLLexer）がいずれの方法でも適切に検出されている．

4.2 実験 2

実験 2 では大規模な Java プロジェクトについて特定のバージョン間におけるホットスポット検出を行う．対象とするリポジトリは `https://github.com/apache/tomcat.git` であり，Java によるサーブレットならびに JSP を実現するためのアプリケーションフレームワークである．対象のリポジトリについてバージョン 8.5.42 からバージョン 8.5.43 間のコミットデータを収集し，収集した修正履歴等の情報からホットスポットの検出を行う．バージョン 8.5.42 からバージョン 8.5.43 間の情報を表 7 に示す．

実験 1 と同様に領域 Z の特定には CS ならびに ES を適用した．CS の領域群拡大の終了条件には「領域群の合計のコード行数が全体の 50 %になるまで」という条件を用いた．さらに，属性値の定義として前述の (i) および (ii) の両方を適用した．表 8 は CS ならびに ES および二通りの属性値定義 (i), (ii) を適用した時，LLR が最も高くなった領域におけるファイル数，LLR ならび領域の属性値 $C(Z)/S(Z)$ を示している．なお，実験 1 と同じくこれらの領域はすべて，サンプル数は 10000 個のモンテカルロ検定を通じて，有意水準 1% で棄却された．さらに，表 **??** から表 **??** はそれぞれの手法でホットスポットとして検出されたファイル群を示している．各表では，ファイル毎の属性値 c_i/s_i と全体のファイル中における属性値の順位を表している．特に，ES では 112 のファイル群が検知されてたため，属性の定義 (i) と (ii) で違いがあったファイルのみを示しており，他の 111 個のファイルは定義 (i) と (ii) で共通している．詳細な結果を `https://git.io/fjH5l` に掲載している．

Tomcat の実験結果では，JSPetriNet と比べて，CS と ES によるホットスポットに大きな違いが生じた．CS での検出は変更の割合（属性値）が高かった `Ranges.java` や `ContentRange.java` を中心として，そこから距離 1 のファ

表 7 Tomcat リポジトリの情報（バージョン 8.5.42〜バージョン 8.5.43）

情報	数値
総ファイル数	2350
コミット数	91
変更があったファイルの数	187

表 8 検出されたホットスポットの LLR と属性値 (Tomcat).

手法	属性値の定義	$LLR(Z)$	$C(Z)/S(Z)$	ファイル数
CS	(i)	1445.2	0.00031	4
	(ii)	1991.3	0.02385	4
ES	(i)	4319.0	0.00009	112
	(ii)	2559.1	0.02386	112

表 9 Tomcat のホットスポット（CS, 属性 (i)）.

ファイル名	c_i/s_i	順位
servlets/DefaultServlet.java	3.22E-4	7
util/http/parser/ContentRange.java	1.64E-2	1
util/http/parser/HttpParser.java	7.42E-5	65
util/http/parser/SkipResult.java	0.00E+0	2560

表 10 Tomcat のホットスポット（CS, 属性 (ii)）.

ファイル名	c_i/s_i	順位
servlets/DefaultServlet.java	3.19E-4	7
util/http/parser/HttpParser.java	7.17E-5	65
util/http/parser/Ranges.java	1.92E-3	1
util/http/parser/SkipResult.java	0.00E+0	2560

表 11 Tomcat のホットスポット（ES, 属性 (i)）.

ファイル名	c_i/s_i	順位
util/http/parser/Ranges.java	1.92E-3	1

表 12 Tomcat のホットスポット（ES, 属性 (ii)）.

ファイル名	c_i/s_i	順位
util/http/parser/ContentRange.java	1.64E-2	1

イルを含むように領域が設定されている．そのため，修正がなかった `SkipResult.java` を含んでいる．これに対して，ES で検出されたホットスポットは属性値が高いファイルが集まっており，修正がないファイルは含まれていなかった．しなしながら，ファイル数が 112 であり，バージョン間で修正があったファイル数 187 の約 60%を占めている．つまり，機能的にまとまったホットスポットをうまく検出できておらず，別々の機能が 1 つの集合となっている．

CS と ES の結果を比べると，CS ではファイル間の呼び出し関係が近いことが担保されるため，CS が出力する結果は機能的にまとまっている可能性が高い．一方で，ES は CS よりも修正が集まっている集合を見つけ出す能力は高いことが，機能的なまとまりを見るには集合が大きくなりすぎる傾向がある．そのため，検出されたファイル集合がどのような意味を持つのかが判断しにくい．ES の高いホットスポット検出性能を有効に利用するために，ES に工夫（集合の属性値の合計に制限を設けるなど）を加えて検出されるホットスポットに何らかの意味付けを行っていく必要があると思われる．

5. まとめ

本論文では空間スキャン検定を用いたプログラム修正ホットスポット検出を議論した．具体的に，リポジトリの修正履歴に提要するための手法について述べ，実際に二つの GitHub リポジトリから得られたプログラム修正履歴を用いてプログラム修正ホットスポットの検出を行った．少ない修正履歴に対しては，ホットスポットの検出によってバージョン間で変更された機能のうちの 1 つに関連するファイルの集合を検出することができた．一方で，コミット数多い大規模なリポジトリでは，CS と ES のホットスポット検出能力の違いが顕著に現れ，CS ではホットスポットの検出能力の低さ，ES では検出されたホットスポットの意味付けという課題が浮き彫りとなった．

今後の課題として，CS のホットスポット検出能力の改善，ES で出力されるホットスポットの意味付けや利用方法，複数の機能に変更があった場合，どのようにして変更に関係があるファイルを検出できるかという問題を考える．また，今回の実験はリポジトリ内の Java ファイルに限定してプログラム修正ホットスポットの検出を行ったため，Java ファイル以外のファイルについてどのようにバージョン間での修正ホットスポットを検出していくかという問題を取り扱う予定である．

参考文献

[1] M. Kulldorff, “A spatial scan statistic,” *Communications in Statistics - Theory and Methods*, 26(6), pp.1481–1496, 1997.

[2] I. Jung, M. Kulldorff, O.J. Richard, “A spatial scan statistic for multinomial data” *Statistics in Medicine*, 29(18) pp. 1861–1961, 2010.

[3] 石岡文生, 栗原考次, “エシェロン解析に基づくスキャン法によるホットスポットの検出について,” **統計数理**, 60(1), pp. 93–108, 2012.