Vol.2013-SE-181 No.2
2013/7/17

OSS 開発におけるタスク割当ての最適化に関する考察

柏 祐太郎[1,a]　大平 雅雄[1,b]

概要：大規模オープンソースソフトウェア (OSS) 開発プロジェクトには，大量の不具合が日常的に報告されている．プロジェクト管理者は，個々の不具合報告を精査し，共同開発者らと議論しながら不具合の原因を特定する役割を担っている．さらに，不具合を修正するのに最も適した開発者を選別し，不具合の修正作業 (不具合修正タスク) を割当てる必要がある．しかしながら，不具合が大量に報告される状況下では，個々の不具合それぞれに対して適任の開発者を選ぶ事は容易ではない．多くの研究では最適な開発者の推薦方法が提案されてきた．しかしながら，多くの既存手法は開発者が修正できるタスク量の上限を考慮していないため，タスクが特定の開発者に集中する傾向にある．そのため本研究では，タスク割当の最適化を目的とする．提案手法の評価を目的とする実験を行なった結果，提案手法は，(1) 実際の割当て結果より修正時間の短縮が可能なこと，(2) タスクの分散が可能であることを確認した．

1. はじめに

近年，大規模オープンソースソフトウェアには多くの不具合が報告されている [1]．報告された不具合はプロジェクトの管理者によって不具合の割当てが行なわれている．プロジェクト管理者は，個々の不具合報告を精査し，共同開発者らと議論しながら不具合の原因を特定する役割を担っている．さらに，不具合を修正するのに最も適した開発者を選別し，不具合の修正作業 (不具合修正タスク) を割当てる必要がある．

しかしながら，不具合が大量に報告される状況下では，個々の不具合それぞれに対して適任の開発者を選ぶ事は容易ではない．その理由として，修正すべき不具合にはセキュリティの脆弱性に関するような緊急性の高いものから，プロダクトのユーザビリティの改善に関するような緊急性の低いものまで様々であるためである．また，修正作業にあたって高度な技術的スキルを必要とするものや，当該不具合に関する専門知識を必要とするものも含まれる．したがって，現状の大規模 OSS プロジェクトでは，必ずしも適任ではない開発者にも不具合が割当てられる事が頻繁に生じており，その結果として，プロジェクト全体として不具合修正時間が長期化している [2]．

これまで，不具合修正タスクの割当を支援するために，過去に同じ不具合が報告されていないかを発見する方法 [3][4] や管理者が割当に必要とするの情報の研究 [5][6][7] といった間接的に割当てを支援する方法から，不具合報告の内容をテキストマイニングし適任の開発者を推薦する方法 [8] などの直接的に割当てを支援する方法が提案されてきた．近年では後者が割当支援の主流であるが，その支援手法のほとんどは，各開発者が一定期間内に修正可能な不具合数の上限を考慮していないため，プロジェクト内の極めて有能な一部の開発者に不具合修正タスクが集中して割当たるという傾向があり，プロジェクト全体としては依然として不具合修正が効率的には行なえないのが現状である．

ここで本研究の目的を以下に示す．
- タスク集中の回避手法の構築
- プロジェクトにおける修正時間の長期化の緩和

前者には開発者に割当てるタスク量に上限を設定する事により，タスク集中の回避を試みる．後者には不具合がどの開発者に直してもらうべきかをプロジェクト全体で最適な組合せを求めることで，プロジェクトの修正活動の効率化を目指す．本研究では不具合の割当問題を開発者と不具合の組合せ問題と考え，これにタスク量の制約条件を課した上で解を求める事が出来る整数計画法を用いる．

2. 関連研究

近年では複数の観点に着目したタスクの割当てに関する研究が盛んに行なわれている．

2.1 再割当が発生する傾向を考慮した割当て

近年の不具合の長期化の主な原因の 1 つとして，不具合の担当者が何度も変更されること（再割当）の頻発があ

1 和歌山大学
Wakayama University
a) s141015@sys.wakayama-u.ac.jp
b) masao@sys.wakayama-u.ac.jp

© 2013 Information Processing Society of Japan

る [2]．Eclipse プロジェクトにおいて再割当は 37-44%の不具合で発生し，再割当が起こるたびに修正が大きく遅れると報告されている [9].

Jeong らは，再割当が発生する傾向が開発者間の関係に左右されることに着目し，各開発者に割当てられた不具合が誰によって修正されているかの確率をマルコフ連鎖に基づいてグラフ化した．このグラフを不具合の割当てに応用することで，再割当の回数が少なくなるような不具合の割当を行なう．

2.2 開発者が修正する不具合の傾向を考慮した割当て

近年の開発者推薦方法の主流として機械学習を用いて適任の開発者を推薦する方法が多く提案されている [8][10][11]. これらの方法では，不具合票のタイトルと概要に出現する単語を応用する．開発者が修正した不具合に含まれる単語の出現頻度に対して機械学習のアルゴリズム (Naive-Bayes[12], SVM[13], C4.5[14] など) を適用することで不具合推薦のモデルを得ることができる．不具合が報告された際には，このモデルに従うことで適任の開発者の推薦を行なう．

2.3 開発者のタスク量を考慮した割当て

近年では，開発者のタスク量に着目した研究が盛んに行なわれている．Mockus らは OSS プロジェクトの修正活動を分析し，数百人を超える開発者がプロジェクトに存在しても，一部の開発者たちにより不具合修正が行なわれていると報告し [15]，Guo らは，長期化の原因である再割当は開発者のタスク量が多くなることが主な原因の 1 つあると報告した [16].

開発者のタスク量に着目した研究が盛んに行なわれる一方，タスク量に着目した不具合の割当手法に関する研究はほとんど存在しない．タスク量を考慮しない割当て手法では，特定の開発者にタスクが集中してしまうことが考えられ，再割当ての原因となり得る．そこで本研究では，タスク集中の回避を目的とした割当手法の構築を提案する．

3. 提案手法

3.1 整数計画法

整数計画法は与えられた条件の下で目的を達成するためにより良い解を求める方法である．整数計画法に代表される数理計画法は，近年の計算機の発達により再注目されている最適化手法である．生産問題やスケジューリング問題といった数学の分野をはじめとして，ソフトウェア工学の分野でも応用され始めている [17]．整数計画法は以下のように定義され [18],

$$Max: \quad c^{\mathsf{T}}x \tag{1}$$

$$Subject: \quad Ax \leq b \tag{2}$$

$$x \geq 0 \tag{3}$$

$$x \text{ は整数} \tag{4}$$

x は n 次元整数ベクトル，c は n 次元ベクトル，b は m 次元ベクトル，A は m 行 n 列の行列である．x が問題の解で目的変数と呼ばれる．式 (1) は目的関数と呼ばれ，式 (2), 式 (3)，式 (4) の制約の下，目的関数を最大とする目的変数 x の組合せを求める．

3.2 整数計画法に基づくタスク割当て

不具合修正タスクの割当問題とは，どの開発者がどの不具合をいくつ担当すればプロジェクトで最も効率的に不具合修正が行われるかを求める問題である．

本研究では不具合修正タスクの割当問題を開発者とタスクの組合せ問題と捉え，組合せ問題において最適解を得ることができる整数計画法を用いる．

3.2.1 定義

ここでは以下の議論を円滑に行なうため，次の 3 つについてあらかじめ定義する．

A タスクの分類

本研究では，開発者（$D_i, i = 1, 2, ..., m$）とカテゴリー j（$j = 1, 2, ..., n$）で分類された不具合票が存在すると想定する．不具合が修正される時間はコンポーネントと優先度で違う事から [15]，本研究では不具合票をコンポーネントと優先度で分類した．本研究で用いる Eclipse の不具合票では優先度は “P1”, “P2”, “P3”, “P4”, “P5” の 5 段階で示される．しかし，ほとんどの不具合が優先度がデフォルトの “P3” であるため，本研究では，優先度が “P1” と “P2” を優先度 “高”，“P3” を優先度 “普通”, “P4” と “P5” を優先度 “低” とし，3 段階で分けた．また 1 件の不具合票を 1 つの不具合修正タスク（以下，タスクと呼ぶ）と定義し，複数の開発者で分割して修正されないものとする．

B プリファレンス

整数計画法の目的関数では，目的変数に係数が必要である．提案手法では係数として，プリファレンス P を定義する．プリファレンスとはタスクをどの開発者に優先的に割当てるべきかを示す尺度である．本研究では過去に修正した不具合数が多い開発者ほど不具合が割当てられるべきとする．プリファレンスの求め方は分類した各カテゴリでデータセット内の全修正不具合数に対する開発者 D_i の修正不具合数の比とする．

$$P_{ij} = \frac{\text{カテゴリ j における開発者 } D_i \text{の修正数}}{\text{カテゴリ j における全開発者の修正数}} \tag{5}$$

C 修正可能なタスク量の上限

© 2013 Information Processing Society of Japan

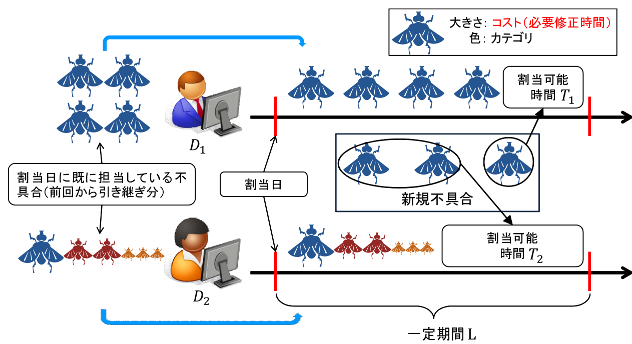

図 1 修正可能なタスク量の求め方

Fig. 1 A way to calculate the amount of tasks for each developer

開発者が一定期間内に修正できるタスク量には限りがある．本研究では開発者が修正可能なタスク量を考慮した不具合の割当てを行なう．図 1 は修正可能なタスク量を求め方を表した概略図である．

開発者 D_i の**修正可能なタスク量**は開発者 D_i がカテゴリー j の不具合を直す際にかかる**コスト** C_{ij} を基に算出する．コスト C_{ij} は開発者 D_i がカテゴリー j の不具合 B_j を修正に要した時間の中央値である．

開発者 D_i の修正可能なタスク量は割当てるタスクのコスト C_{ij} の合計が担当可能時間 T_i を超えないようなタスク量である．開発者 D_i の担当可能時間 T_i はあらかじめ設定する**一定期間** L と開発者 D_i が担当している不具合のコストから求め，次の式で表される．

$$\text{修正可能時間 } T_i = \text{一定期間 } L - \text{担当している不具合のコスト} \quad (6)$$

新規のタスク割当てを行なう際に，割当てるタスクの総コストが，担当可能時間 T を超えないように割当てることで，特定の開発者にタスクが極端に集中するのを防ぐ効果を期待できる．

3.2.2 定式化

本研究では目的変数，目的関数，制約条件を次のように定義する．

A 目的変数：x_{ij} とは開発者 D_i がカテゴリー j のタスクをいくつ担当させるかを表す．

$$x_{ij} \geq 0 \quad (7)$$

x_{ij} は整数に限る

B 目的関数：各開発者と各カテゴリのプリファレンスとその目的変数の関の総和を最大化する．つまり，個々の開発者の適性に合うタスクがプロジェクト全体として最大となるような組合せを求める．

$$Max : \sum_{i=1}^{m}\sum_{j=1}^{n} P_{ij}x_{ij} \quad (8)$$

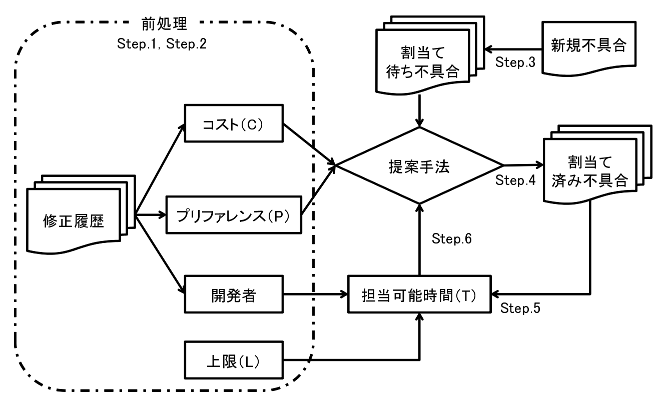

図 2 フローチャート

Fig. 2 Flowchart

C 制約条件

(a) 各開発者が一定期間内に修正可能なタスク量は有限であること．

$$\sum_{j=1}^{n} C_{ij}x_{ij} \leq T_i \ (i = 1, 2, ..., m) \quad (9)$$

(b) 割当てられる不具合数は報告された不具合数以下であること．

$$\sum_{i=1}^{m} x_{ij} \leq R_j \ (j = 1, 2, ..., n) \quad (10)$$

3.3 提案手法の適用手順

以下に提案手法の適用手順を以下に示し，図 2 にフローチャートを示す．

Step.1: **パラメータの設定**

提案手法では前もって上限 L を設定する．その L の値を各開発者の担当可能時間 T_i に代入する．

Step.2: **プリファレンスとコストの算出**

それぞれの開発者（$D_i, i = 1, 2, ..., m$）とカテゴリー（$j = 1, 2, ..., n$）のコスト C_{ij} とプリファレンス P_{ij} を算出する．

Step.3: **報告された不具合を割当ての対象に含める**

報告された不具合を割当て待ち不具合に追加する．

Step.4: **整数計画法の適用**

割当て待ち不具合を整数計画法で開発者に割当てる．割当てられた不具合は割当て待ち不具合から外す．

Step.5: **各開発者の担当可能時間** T_i **を更新**

各開発者に割当てられてた不具合のコスト分だけ各開発者の T_i を減らす．

Step.6: **次の割当てを行なう日に進む（Step.3 へ）**次の日に進み，各開発者の T_i を 1（日）増やす（Step.1 で設定した L より大きくならないとする）．

4. 実験方法

4.1 実験の概要と目的

本実験は、提案手法が開発者のタスク量を考慮し，修正

表 **1** データセット
Table 1 Dataset

データセット	期間	全解決済み不具合数（件）	対象不具合数（件）	コンポーネント数	カテゴリー数
A	2004/6/21～2008/6/24	10,994	6,032	18	32
B	2008/6/25～2012/6/26	5,472	2,109	18	41

表 **2** 本実験における計算環境
Table 2 Computional environment in our experiment

CPU	Intel Core i7 2.30 GHz
OS	CentOS 6.2
メモリ	2GB

時間が削減できることを確認する目的で行なう．提案手法の効果を有効性を確かめるために 2 つの実験を行ない, 実験 I では提案手法で実際の修正時間より削減できるかを, 実験 II では、提案手法に課した条件が有効であるかを検証する．

4.2 準備

4.2.1 データセットとデータ整形

本研究のデータセットには比較的大きく，十分成熟しているプロジェクトである Eclipse 3.x リリース時の Eclipse platform プロジェクトの解決済み不具合を用いる．Eclipse の不具合データベースから修正時間及び修正者を特定でき，1 回で修正 (FIXED) とされた不具合をデータセットとして用いる．また，Eclipse プロジェクトでは，ほとんどの不具合が少数の開発者のみが行なっている．本研究では全開発者の 115 人から修正活動を行なう見込みのある 37 人を選定し，それらの開発者以外が修正した不具合はデータセットに含めていない．

本研究では 2004 年 6 月 21 日～2008 年 6 月 24 日の間に報告された不具合（データセット A）で提案手法を構築し，2008 年 6 月 25 日～2012 年 6 月 26 日の間に報告された不具合（データセット B）を開発者に割当てた．各データセットを表 1 に示す．

4.2.2 修正時間

開発者の修正時間の算出には以下の式を用いる [11]．

$$\begin{aligned} &bug\ fix\ time\ (in\ day) = \\ &\quad fixed\ time - assigned\ time + 1\ day \end{aligned} \tag{11}$$

4.2.3 実行環境

本実験において整数計画法の実行には表 2 の環境で行なった．整数計画法の解を求めるにはオープンソースの数理計画ソフトウェアである lp_solve5.5.2.0 [*1] を用いた．

4.2.4 パラメータの設定

提案手法では上限 L を設定しなくてはいけない．本研究では各開発者が不具合修正にかかるコストの第 3 四分位値を求め，その値を切り上げた値である 13 を上限 L と設定した．

		プリファレンス	
		無	有
上限	無	プリファレンス：無 上限：無	プリファレンス：有 上限：無
	有	プリファレンス：無 上限：有	プリファレンス：有 上限：有（提案手法）

図 **3** 評価のためのモデルの分類
Fig. 3 Category of models for evalution

4.3 評価項目

4.3.1 修正時間の削減効果

提案手法の最終的な狙いは，プロジェクト全体としての不具合修正時間の短縮化である．そこで，提案手法がプロジェクト全体としての不具合修正時間の短縮に寄与するかどうかを評価する．評価項目には各不具合が割当てられてから修正されるまでにかかった時間を足し合わせた時間（総修正時間と呼ぶ）を使う．提案手法の総修正時間は不具合が割当てられてから修正に取り組まれるまでの待ち時間を考慮している（本実験では割当てられた不具合の順で修正されるものとしてシュミレーションする）．

4.3.2 課された条件のタスクの分散効果

提案手法には複数の条件が課せられている．1 つ目は，一定期間内に開発者が修正できるタスクは有限であること．2 つ目は，プリファレンスが大きい開発者に優先的に不具合を割当てていることである．これらの条件がタスクの分散に与える効果について検証するため，開発者が修正に取り組んだの平均時間，開発者が修正に取り組んだ最大時間を評価項目とする．開発者のが修正に取り組んだ平均時間と最大時間は小さいほど良いとする．

4.4 実験手順

4.4.1 修正時間の削減効果

実験 I では，提案手法を実行し，得られたタスクの割当て結果と実際にタスクが割当てられ，修正履歴に残っている修正時間を比較する．

4.4.2 課された条件のタスク分散効果

実験 II では，プリファレンスも上限も課さないモデルつまり，不具合修正が早い開発者に割当てるモデルに対し，

*1 http://lpsolve.sourceforge.net/

表 3　提案手法と実際の割当て結果との修正時間比較
Table 3　Comparison of the proposed method and actual task assignment

	実際の割当方法	提案手法
割当てた不具合（件）	2,109	2,089
割当てた開発者（人）	25	24
総修正時間（日）	56,644.1	5,967.2
平均修正時間（日）	26.9	2.9

2 つの条件を課すことでタスクがどう分散されるかを確認する．以下に各条件が有および無である場合の組合せである 4 つのモデルを示す（図 3）.

- プリファレンス：無 / 上限：無
  プリファレンスと上限を設定せず，過去の修正数が多い開発者に不具合を割当てるモデル.
- プリファレンス：有 / 上限：無
  プリファレンスのみ設定し，カテゴリーごとのプリファレンスが高い開発者にタスクを割当てるモデル
- プリファレンス：無 / 上限：有
  上限のみ設定し，過去の修正数が多い開発者中心に上限を超えないように不具合を割当てるモデル.
- プリファレンス：有 /上限：有 （提案手法）
  上限を設定した条件下で，プリファレンスの合計（プロジェクト全体）が最大となるようにタスク割当てるモデル．提案手法に相当する.

5. 実験結果

5.1 実験 I：修正時間の削減効果

実験 I の結果を表 3 に示す．平均修正時間とは不具合が割当てられて修正がされるまでの時間（他の不具合が修正されている場合の待ち時間も含む）の平均である．表 3 を見てわかるように実際の修正時間は 56,644.1 日で，提案手法を用いて得られた総修正時間は 5967.2 日となり，実際の修正時間の約 10 分の 1 となった.

5.2 実験 II：課された条件のタスクの分散効果

実験 II の結果を表 4 に示し，各モデルと比較を行なった．平均コストとは本実験で用いたコスト（平均修正時間とは異なり，待ち時間は含まれていない）の平均値である.

- { プリファレンス：無，上限：無 } と { プリファレンス：有，上限：無 }
  上限を設定しない状態でプリファレンスの有無で比較すると，プリファレンスを設定する事で割当て人数は 13 人増加し，開発者が修正に取り組んだ平均時間は約 1150 日短く，平均修正時間は 2.2 日短くなった.
- { プリファレンス：無，上限：無 } と { プリファレンス：無，上限：有 }
  プリファレンスを設定しない状態で上限の有無で比較すると，上限を設定する事で割当て人数は 8 人増加し，開発者が修正に取り組んだ平均時間は約 12000 日短く，平均修正時間は 4.2 日短くなった.

表 5　実験 I の追加実験結果
Table 5　Additional result in experiment I

	≤100	≤365	提案手法
割当てた不具合（件）	1,970	2,085	2,089
割当てた開発者（人）	25	25	24
総修正時間（日）	21,594.9	42,099.5	5,967.2
平均修正時間（日）	11.0	20.2	2.9

- { プリファレンス：有，上限：無 } と { プリファレンス：有，上限：有 }
  プリファレンスを設定している状態で上限の有無で比較すると，上限を設定する事で割当て人数は 10 人増加し，開発者が修正に取り組んだ平均時間は約 340 日，不具合の平均修正時間は 1.2 日短くなった．また，開発者が修正に取り組んだ最大時間も約 1600 日短くなった．一方，平均プリファレンスは約 12.3 小さくなった.
- { プリファレンス：無，上限：有 } と { プリファレンス：有，上限：有 }
  上限を設定している状態でプリファレンスの有無で比較すると，上限を設定する事で割当て人数は 15 人増加し，開発者のが修正に取り組んだ平均時間と開発者が修正に取り組んだ最大時間は 200 日短くなった．一方，不具合の平均修正時間は約 0.8 日長くなった.

実験 II の結果から，プリファレンスと上限は設定することで共に割り当て人数が増加した．{ プリファレンス：有，上限：無 } と { プリファレンス：有，上限：有 } の比較では上限を設定する事でプリファレンスは小さくなるものの，開発者が修正に取り組んだ平均時間と最大時間が短くなる結果となり，タスクの分散には効果的である．また，{ プリファレンス：無，上限：有 } と { プリファレンス：有，上限：有 } の比較からプリファレンスを設定するだけでは不具合の平均修正時間が短くなるとは限らない結果となったが，開発者が修正に取り組んだ平均時間と最大時間が小さくなったことから，タスクの分散には効果的である.

6. 考察

6.1 総修正時間

実験 I では実際の割当方法と比べて修正時間が大きくなった．修正時間が遅くなった理由は一部の不具合が修正時間大きいことにある．実際の修正時間には 365 日以上の要した不具合の総修正時間は全体の約 26%を占め，100 日以上の要した不具合の総修正時間は全体の約 63%を占める．特に 365 日以上要した不具合は 365 日もの間にずっと

表 4 モデルごとの実験結果
Table 4 Experimental result of each model

プリファレンス：	無	有	無	有
上限：	無	無	有	有
割当てた不具合（件）	2,109	2,097	2,109	2,089
割当てた開発者（人）	1	14	9	24
総修正時間（日）	29,977.2	10,431.6	5,281.7	5,967.2
平均修正時間（日）	14.2	5.0	2.5	2.9
平均コスト（日）	5.9	3.7	1.8	2.5
開発者が修正に取り組んだ平均時間（日）	12,511.7	557.1	419.8	219.1
開発者が修正に取り組んだ最大時間（日）	12,511.7	3,088.6	1,471.3	1,245.2
開発者が修正に取り組んだ最小時間（日）	12,511.74	8.0	12.7	8.0
開発者が修正に取り組んだ時間の分散	0	971,513.8	286,553.8	84,998.2
不具合あたりの平均プリファレンス	-	52.4	-	40.1

修正していたとは考えられない．もし，修正していたとしていれば他の不具合もすべて 365 日以上かかるってしまうことが考えられる．そのため，365 日以上要した不具合は不具合が割当てられた後，放置されていたと考えるのが妥当と思われる．そこで，本来修正されるべきである不具合だけを比較するため，365 日以上及び，100 日以上要した不具合を修正時間の計算から外し，再度，提案手法の修正時間と比較を行なった結果を表 5 に示す．修正に要した時間が 100 日以内，365 日以内の不具合を対象にした結果と提案手法の修正時間を比較すると，100 日以内のデータの総修正時間より約 73%の削減，365 日以内のデータの総修正時間より約 86%の削減となった．

6.2 プリファレンスの必要性

実験 II ではプリファレンスがない方が不具合の修正時間を削減できるという結果となった．しかし，プリファレンスを用いないという事はソフトウェア部品に対する専門性を考慮しないということになる．本節では，不具合の修正においてソフトウェア部品の専門性は実際に必要とされているか及び，修正時間が短くなった理由について考察する．

まずソフトウェア部品の専門性が必要であるかを確かめるためにデータセット A で修正数が上位 6 位である開発者のコンポーネント別による修正数を図 4 に示す．この図の通り，多くの開発者が特定のコンポーネントを優先的に直す傾向が見られた．つまり，開発者の得意なコンポーネント以外を割当てても修正を行なわれない可能性が高く，プリファレンスは必要である．

次に，プリファレンスを設定しないモデルの方が修正時間が短くなった理由について考察する．その理由は，コンポーネントで平均修正時間が違う事にある．表 6 はコンポーネントごとの平均修正時間である．この表から見てわかるように，平気修正時間が最も短いコンポーネントと長いコンポーネントを比較すると，10 倍以上も違う．平均修正時間の短いコンポーネントを普段修正している開発者に

表 6 コンポーネントごとの平均修正時間
Table 6 Mean time to fix a bug by component

コンポーネント	A	B	C	D	E	F
平均コスト（日）	10.1	8.6	7.8	6.4	6.1	5.1
コンポーネント	G	H	I	J	K	L
平均コスト（日）	4.7	4.1	3.6	1.9	1.4	1.3
コンポーネント	M	N	O	P	Q	R
平均コスト（日）	1.2	1.0	1.0	1.0	1.0	1.0

割当てられたため，プリファレンスを設定しないモデルの方が修正時間が短くなった．

6.3 提案手法の適用に向けて

本手法を用いることでタスク集中の回避が可能となった．また，本手法は自動化も可能であり，管理者の日々多くの不具合精査する負担を減らす事もできる．特に本手法は，不具合報告が多いときに負担軽減の効果を発揮すると考えている．不具合報告が増加すれば，開発者と不具合の組合せが膨大に増え, 適切な開発者への割当ての難易度も上がると考えられる．しかし，本手法は整数計画法を用いているため，最適な組合せを高速に求める事ができるため，不具合が増加した場合にも対応可能である．

一方，この手法の問題点は，割当て基準が報告時に設定する優先度やコンポーネントの値が大きく影響することである．これらの値を適切に設定されなければ，不具合が適切な開発者に割当てられない可能性がある．

6.4 研究の制約

ここでは本研究の結果を脅かす要因を内的，外的，構造的要因の 3 つの観点から考察する．

- 内的要因
  今回の実験でプリファレンスを設定すると修正時間が長くなった．同様にさらにプリファレンスをコンポーネントと優先度だけでなく，もっと細かく分類していくと，実験 I の実際の割当てと比較し必ずしも修正時

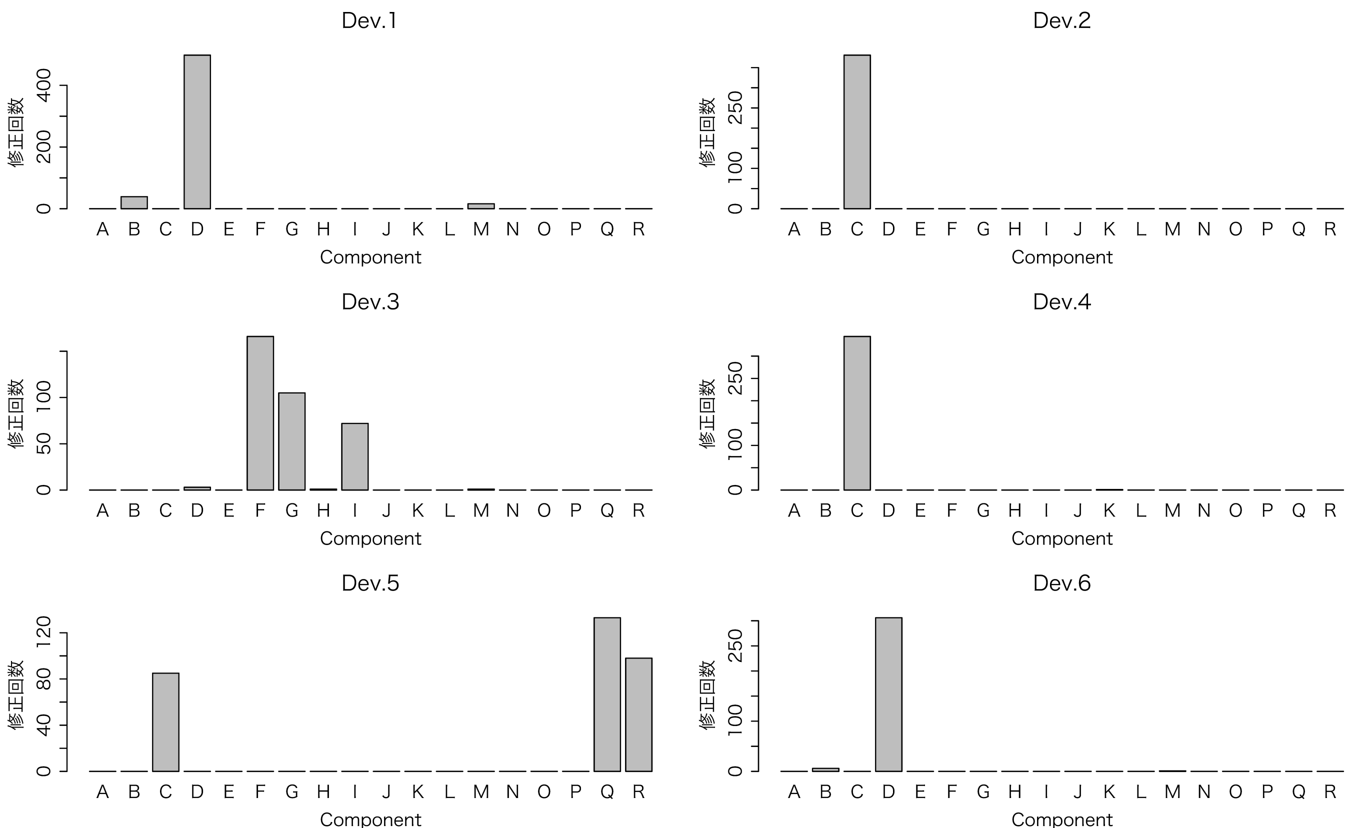

図 4　修正数上位 6 人のコンポーネント別修正数
Fig. 4　Number of bugs in each component, fixed by the top 6 developers

間が短くなるとは限らない．しかし，今回は整数計画法の導入によるタスク集中の回避に焦点を当てているためプリファレンスの妥当性についての本格的な議論は避けたいと考えている．そのため，プリファレンスの改良を今後の課題とする．

- 外的要因
  本研究では比較的に大規模の OSS プロジェクトである Eclipse Platform プロジェクトのデータを用いて不具合の割当を行なった．しかし，OSS プロジェクトによっては不具合の報告件数，開発者の人数や平均修正時間が大きく異なるため，提案手法が他のプロジェクトにも適用できるかは明らかにできていない．提案手法の一般性を確かめることを今後の課題とする．
- 構造的要因
  本研究では一定期間に開発者が修正できるタスクには限りがあることに着目し，整数計画法を適用することでタスク集中の回避を試みた．
  ここで疑問とされることは，多種にわたる数理計画の手法の中で，整数計画法を適用する事が妥当であるかどうかである．整数計画法以外の数理計画の代表的な手法は線形計画法や 0-1 整数計画法などが存在する．
  線形計画法は，整数計画法と違い，目的変数に実数をとることが出来る．タスクの割当て問題では目的変数は開発者に割当てる不具合数であり，少数単位で割当てが可能という事になる．少数単位のタスクの割当ては，不具合の割当て問題では不自然である．また，本研究の問題は不具合の再割当てによる不具合修正の長期化であるのに対し，再割当てを助長することになり，矛盾が発生する．従って本研究では線形計画法を用いなかった．
  次に 0-1 整数計画法だが，この手法は目的関数を 0 か 1 のみをとる，つまり，不具合の 1 つ 1 つを割当てる・割当てないを決定するという意味を持つ．この方法では不具合の 1 つ 1 つを識別する方法が必要である．今回は不具合の 1 つ 1 つを識別する方法は複雑となるため，今回は 0-1 整数計画法を用いる事を避けた．そのため，今後は 1 つ 1 つの不具合を識別した割当てを今後の展望とする．

7. まとめ

大規模オープンソースソフトウェア (OSS) 開発プロジェクトには, 大量の不具合が日常的に報告されている．不具合が大量に報告される状況下では，個々の不具合それぞれに対して適任の開発者を選ぶ事は容易ではない．修正すべき不具合には，セキュリティの脆弱性に関するような緊急性の高いものから，プロダクトのユーザビリティの改善に関するような緊急性の低いものまで様々であるためである．また，修正作業にあたって高度な技術的スキルを必要

とするものや，当該不具合に関する専門知識を必要とするものも含まれる．したがって，現状の大規模 OSS プロジェクトでは，必ずしも適任ではない開発者にも不具合修正タスクが割当てられる事が頻繁に生じており，その結果として，プロジェクト全体として不具合修正時間が長期化している．これまで，不具合修正タスクの割当を支援するために，不具合報告の内容をテキストマイニングし適任の開発者を推薦する方法や，適任ではない開発者に不具合修正タスクが割当てられるのを防ぐための手法などが提案されてきた．

しかしながら，既存の支援手法のほとんどは，各開発者が一定期間内に修正可能な不具合数の上限を考慮していないため，プロジェクト内の極めて有能な一部の開発者に不具合修正タスクが集中して割当たるという傾向があり，プロジェクト全体としては依然として不具合修正が効率的には行なえないのが現状である．本研究の目的は，開発者が一定期間内に修正可能なタスク量の上限を考慮しつつ，プロジェクト全体として効率よく不具合修正が行なえるようにするためのタスク割当支援手法を構築する事である．本研究で提案するタスク割当支援手法は，整数計画法を利用する事で，制約条件として各開発者のタスク量を考慮したタスク割当手法を構築する事が出来る．また，提案手法は，開発者が過去に担当した不具合の情報を用いるため，開発者のタスク量の上限に考慮しつつ適任の開発者にタスクを割当てることができる．

提案手法の評価を目的とする実験を行なった結果，提案手法は，(1) 実際の割当て結果より修正時間の短縮が可能なこと，(2) タスクの分散が可能であることを確認した．

謝辞 本研究の一部は，文部科学省科学研究補助金（基盤 (B):23300009）および（基盤 (C):24500041）による助成を受けた．

参考文献

[1] Bettenburg, N., Premraj, R., Zimmermann, T. and Kim, S.: Duplicate bug reports considered harmful ... really?, *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM '08)*, pp. 337–345 (2008).

[2] Jeong, G., Kim, S. and Zimmermann, T.: Improving bug triage with bug tossing graphs, *Proceedings of the the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*, pp. 111–120 (2009).

[3] Runeson, P., Alexandersson, M. and Nyholm, O.: Detection of Duplicate Defect Reports Using Natural Language Processing, *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pp. 499–510 (2007).

[4] Wang, X., Zhang, L., Xie, T., Anvik, J. and Sun, J.: An approach to detecting duplicate bug reports using natural language and execution information, *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pp. 461–470 (2008).

[5] Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R. and Zimmermann, T.: What makes a good bug report?, *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*, pp. 308–318 (2008).

[6] Breu, S., Premraj, R., Sillito, J. and Zimmermann, T.: Information needs in bug reports: improving cooperation between developers and users, *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work (CSCW '10)*, pp. 301–310 (2010).

[7] Hooimeijer, P. and Weimer, W.: Modeling bug report quality, *Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*, pp. 34–43 (2007).

[8] Anvik, J. and Murphy, G. C.: Reducing the effort of bug report triage: Recommenders for development-oriented decisions, *ACM Transactions on Software Engineering and Methodology (TOSEM '11)*, Vol. 20, No. 3, pp. 10:1–10:35 (2011).

[9] Bhattacharya, P. and Neamtiu, I.: Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging, *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM '10)*, pp. 1–10 (2010).

[10] Cubranic, D. and Murphy, G. C.: Automatic bug triage using text categorization, *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE '04)*, pp. 92–97 (2004).

[11] Park, J.-w., Lee, M.-W., Kim, Jinhan, H. S.-w. and Kim, S.: COSTRIAGE: A Cost-Aware Triage Algorithm for Bug Reporting Systems, *Proceedings of the Twenty-Fifth Conference on Artificial Intelligence (AAAI'11)* (2011).

[12] John, G. and Langley, P.: Estimating Continuous Distributions in Bayesian Classifiers, *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI '95)*, pp. 338–345 (1995).

[13] Gunn, S. R.: Support Vector Machines for Classification and Regression, Technical report, University of Southampton, Faculty of Engineering, Science and Mathematics; School of Electronics and Computer Science, University of Southampton (1998).

[14] Quinlan, J. R.: *C4.5: programs for machine learning*, Morgan Kaufmann Publishers Inc., San Francisco (1993).

[15] Mockus, A., Fielding, R. T. and Herbsleb, J. D.: Two Case Studies of Open Source Software Development: Apache and Mozilla, *ACM Transactions on Software Engineering and Methodology (TOSEM '02)*, Vol. 11, No. 3, pp. 309–346 (2002).

[16] Guo, P. J., Zimmermann, T., Nagappan, N. and Murphy, B.: "Not my bug!" and other reasons for software bug report reassignments, *Proceedings of the 2011 ACM Conference on Computer Supported Cooperative Work (CSCW '11)*, pp. 395–404 (2011).

[17] 阿萬裕久：論理的制約条件付き 0-1 計画モデルを用いた重点レビュー計画法，コンピュータソフトウェア（日本ソフトウェア科学会誌），Vol. 29,No.2, pp. 612–621 (2012).

[18] 福島雅夫：数理計画法入門，朝倉書店，東京 (1996).

© 2013 Information Processing Society of Japan