ソフトウェアエンジニアリングシンポジウム 2014
IPSJ/SIGSE Software Engineering Symposium (SES2014)

大規模OSS開発における不具合修正時間の短縮化を目的としたバグトリアージ手法

柏 祐太郎[1,a)]　大平 雅雄[1,b)]　阿萬 裕久[2,c)]　亀井 靖高[3,d)]

概要：本論文では，大規模 OSS 開発における不具合修正時間の短縮化を目的としたバグトリアージ手法を提案する．既存手法の多くは，不具合に対する開発者の適性のみを考慮するため，ごく一部の開発者に修正タスクが集中するという問題があった．既存手法に対し提案手法は，開発者の適性に加えて，開発者が一定期間内に修正作業に使える時間の上限を考慮している点に特徴がある．Mozilla Firefox および Eclipse Platform プロジェクトを対象としたケーススタディを行った結果，提案手法について以下の 3 つの効果を確認した．(1) 一部の開発者へタスクが集中するという問題を緩和できること．(2) プロジェクト全体としての不具合修正時間を，人手によるバグトリアージに比べて 50–83%，既存手法と比べて 34–38% 削減できること．(3) 提案手法で用いた 2 つの設定，プリファレンス（開発者の適性）と上限（開発者が取り組むことの出来る時間の上限）が，タスクの分散効果にそれぞれ同程度寄与すること．上記の知見に加え，本論文では，提案手法の適用範囲および妥当性について，追加の分析結果に基づき議論する．

キーワード：バグトリアージ，大規模 OSS 開発，0-1 整数計画法

1. はじめに

近年の大規模システム開発では，試験工程のみならず運用工程においても多数の不具合が検出される．多くの場合，不具合管理システムを用いて，不具合の再現方法や修正方法を詳細に記録し，不具合は漏れのないように管理される．不具合管理システムに報告された不具合一つ一つに対して，重要度や優先度を設定し，開発者に修正タスクを割当てることをバグトリアージと呼ぶ [1].

しかしながら，大量に不具合が報告される現状では，個々の不具合に対して適切にバグトリアージを行なうことは容易ではない．実際，大規模オープンソース開発プロジェクトの Eclipse や Mozilla では，約 4 割の不具合に対して，担当者の再割当てが行われており [2]，人手によるバグトリアージには限界があることが知られている．担当者の再割当は，人的リソースを浪費するだけではなく，不具合の修正作業を滞らせるため，出来る限り生じないようにすることが望ましい．そのため現在，バグトリアージを支援するための研究が盛んに行われている [1–15].

先行研究で提案されている手法のほとんどは，個々の不具合に対して確実かつ迅速に修正できる開発者を推薦することを目的としている．過去の不具合報告とその修正履歴に基づいて，新規に報告された不具合に対して適任の担当者を推薦することで，再割当を起こしにくくすることが狙いである．しかし，既存手法は，個々の不具合修正の難易度や手間を考慮しないため，ごく一部の開発者にタスク割当てを集中させる傾向にある．優秀な開発者でも不具合修正に取り組める時間は有限であるため，既存手法は現実的でないと考えられる．

そこで本研究では，0-1 整数計画法に基づく不具合修正タスクの割当て手法を提案する．不具合の割当て問題を，開発者と不具合の組合せ問題として捉え，個々の開発者が修正作業に使える時間に制約条件を課すことでタスク割当てを最適化し，プロジェクト全体としての不具合修正活動の効率化を目指す．本論文では，以下の Research Question に取り組み，提案手法の有用性について議論する．

RQ1: 既存手法は，特定の開発者へ負荷を集中させる傾向があるか？提案手法ではその問題を緩和できるか？ 代表的な既存のバグトリアージ手法 [3] を用いて，既存手法が一部の開発者にタスクを集中して割当てる可能性があることを確かめる．また，提案手法を用いることでタスク集中

[1] 和歌山大学 Wakayama University
[2] 愛媛大学 Ehime University
[3] 九州大学 Kyushu University
[a)] s141015@sys.wakayama-u.ac.jp
[b)] masao@sys.wakayama-u.ac.jp
[c)] aman@ehime-u.ac.jp
[d)] kamei@ait.kyushu-u.ac.jp

を緩和できることを確かめる．

RQ2: 提案手法は，プロジェクト全体の不具合修正時間の短縮化に寄与するか？ 0-1 整数計画法に基づくタスク割当の最適化によって，プロジェクト全体として不具合修正時間を短縮できるかどうかを実験的に検証する．すなわち，一部の開発者へのタスク集中を緩和することが，プロジェクト全体の不具合修正活動を効率化できることを示す．

RQ3: 提案手法で用いる各種設定が，タスク割当ての最適化にどのように寄与するか？ 提案手法の特徴は主に，(1) 不具合に対する開発者の適性（**プリファレンス**）を数値化し，適性の高い不具合をできるだけ多く割当てることと，(2) 一部の開発者にタスクが集中するのを避けるために，一定期間内に修正作業に使える時間に**上限**を設けることにある．プリファレンス有無，上限の有無により 4 つのモデルを構築し，修正時間短縮化の効果を詳細に調べる．

以降，2 章では，バグトリアージ支援に関する関連研究について紹介し，本研究との違いを明らかにする．3 章では，0-1 整数計画法に基づくバグトリアージ手法を提案する．4 章では，提案手法の有用性を確認するための実験について説明し，5 章で実験の結果を示す．6 章では，実験結果と追加実験の結果に基づいて提案手法の適用範囲および妥当性について考察する．最後に 7 章でまとめと今後の課題について述べ，本論文を結ぶ．

2. バグトリアージ

2.1 現状のバグトリアージにおける問題点

OSS 開発におけるバグトリアージは一般に，Bugzilla[*1] などの不具合管理システム (BTS: Bug Tracking System) を用いて行われる．BTS の管理者は，プロジェクトに所属する多数の開発者から当該不具合を修正するのに適任の開発者を決定し，不具合修正タスクを割当てる必要がある．しかし，多くの場合，BTS 管理者が最初に割当てた開発者では不具合を修正できずに，担当者を変更（再割当）して不具合が修正されている．担当者の再割当ては不具合が最終的に修正されるまでの時間を大きく滞らせる [16]．そのため，近年の大規模 OSS 開発では，再割当の頻発がプロジェクト全体としての不具合修正の長期化を引き起すという問題がある [2]．実際，Eclipse や Mozilla では，約 4 割の不具合に対して担当者の再割当てが行われており，1 度の担当者の変更で修正時間が平均約 50 日遅れるとされている [2]．再割当による不具合修正時間の長期化は，大規模 OSS プロジェクトにおいて解決すべき喫緊の課題であるとされており，バグトリアージを支援する手法がこれまで多数提案されてきた [1–12]．

[*1] Bugzilla: http://www.bugzilla.org/

2.2 既存のバグトリアージ手法とその問題点

既存のバグトリアージ手法の目的は，担当者の再割当が頻発しないようにあらかじめ最も適任と思われる開発者にタスクを割当てることである．既存手法は主に，機械学習に基づく方法を採用している [1,3,5,7,8]．具体的には，開発者が不具合報告に記述したタイトルと概要からなるテキストデータを入力として，各開発者が過去に用いた単語の出現頻度を算出し，機械学習のアルゴリズム（例えば SVM [17]）を適用することで，各不具合に対して開発者を推薦するためのモデルを得る．構築したモデルに従うことで，比較的高精度（約 70-75%程度）に新規に報告された不具合の修正に対応可能な開発者を推薦できる [3]．

しかしながら，1 章でも述べたように，既存手法は，一定期間内に開発者が不具合修正に取り組める時間を考慮しない．また，個々の不具合修正の難易度や手間を考慮しない．そのため，必ずしも有能な開発者が担当する必要のない軽微な不具合であっても，優先して有能な開発者に割当てる可能性が高い．結果として，既存手法は一定期間内に修正作業に使える時間を越えて一部の有能な開発者にタスク割当てを行う恐れがある．有能な開発者以外の開発者にも対応可能な軽微な不具合を，その他の開発者へ分散して割当てることで，プロジェクト全体としての不具合修正活動をさらに効率化できる余地があるといえる．

3. 不具合修正時間の効率化を目的としたバグトリアージ手法

本研究では，開発者が一定期間内に修正作業に使える時間に上限を設けた上で，プロジェクト全体の不具合修正効率にとって最適となる開発者と不具合の組み合わせを求める新たなバグトリアージ手法を提案する．そのアプローチとして，本研究では，不具合と開発者の組合せ問題をマルチナップサック問題 [18] と見なす．マルチナップサック問題（Multiple knapsack problem）とは，重みと利得を持つ複数のアイテムを，最大重量が決まっている複数のナップサックに入れる際，ナップサックに入れたアイテムの総価値が最大となるようなアイテムの組合せを求める問題である．本研究では，ナップサックの集合をプロジェクト，各ナップサックの最大重量を各開発者が修正作業に使える時間の上限，アイテムを不具合，重みを不具合の修正に必要な時間，利得を不具合に対する開発者の適性を数値化したもの（プリファレンス）として考え，マルチナップサック問題の解法である 0-1 整数計画法 [19] を用いる．

3.1 0-1 整数計画法

0-1 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り，ナップサック問題の解法として知られている．0-1 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改めて注目さ

表 1 本論文で用いる用語一覧

用語	記号	意味
カテゴリ	k	不具合票の「コンポーネント」と「優先度」で分類したもの.
プリファレンス	P_{ij}	修正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. P_{ij} とは開発者 D_i がカテゴリ k の不具合修正タスク B_j に対するプリファレンスを示している. $P_{ij} = \frac{\text{カテゴリ}k\text{ における開発者}D_i\text{の修正数}}{\text{カテゴリ}k\text{ における修正タスクの総数}}$
コスト	C_{ij}	開発者 D_i が不具合修正タスク B_j の修正に必要とする時間. 過去に開発者 D_i がカテゴリ k の不具合修正タスクを完了するのに要した修正時間の中央値とした.
上限	L	タスクの集中を防ぐために設定する値
割当て可能時間	T_i	一定期間内に修正作業に使える時間 開発者 D_i の担当可能時間を示す. T_i = 上限 $L - \sum_{j=1}^{n} C_{ij} * x_{ij}$

れている最適化手法であり，生産問題やスケジューリング問題といったオペレーションズリサーチ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 [20]. マルチナップサック問題は以下の 0-1 整数計画問題として定式化できる.

$$Maximize: \quad \sum_{i=1}^{m}\sum_{j=1}^{n} v_j x_{ij} \tag{1}$$

$$Subject\ to: \quad \sum_{j=1}^{n} w_j x_{ij} \leq c_i \quad (i = 1, 2, ..., m) \tag{2}$$

$$x_j \in \{0, 1\} \quad (j = 1, 2, ..., n) \tag{3}$$

ここで v_j および w_j はそれぞれ j 番目のアイテムの利得と重みを表しており，x_{ij} は i 番目のナップサックに j 番目のアイテムを入れるか否か（選択しない：0，選択する：1）を表している. (1) 式は，選択されたアイテムにおける利得の総和を表し，この値を最大化することを目的とする. 一方，(2) 式は i 番目のナップサックの重量制限を表した制約条件であり，選択されたアイテムの総重量が最大重量（c_i）以下でなければならないことを表している. (3) 式は既に述べた x_{ij} に関する制約であり，この値が 0 または 1 のいずれかしか許されないことを示している.

(2), (3) 式の制約の下で (1) 式の値が最大となる x_{ij} の組み合わせを見つけ出すことがマルチナップサック問題の目的である. 定式化された 0-1 整数計画問題は，lp_solve [*2] といったソルバーで容易に解くことができる.

3.2 0-1 整数計画法に基づくタスク割当て

3.2.1 用語定義

まず，以降の議論を円滑に行うために，本論文で用いる用語を定義する. 表 1 には，用語の一覧をまとめる.

カテゴリ（タスクの分類）：本研究では，開発者 D_i $(i = 1, 2, ..., m)$ には，不具合修正タスク B_j $(j = 1, 2, ..., n)$ に対する適性が存在すると想定する. そこでまず，個々の修正タスクをカテゴリ k として分類する. カテゴリ k は，不具合票の「コンポーネント」と「優先度」で定義される. 例えば，対象コンポーネントを “UI” とする不具合票が 2 件あり，それぞれ優先度が “P1” と “P5” とされている場合は別々のカテゴリの修正タスクとして区別される. 同様に，同じ優先度であってもコンポーネントが異なる場合は，別カテゴリとして区別される. 修正タスクの分類にコンポーネントと優先度を用いる理由は，不具合修正時間がコンポーネントと優先度に依存することを示した先行研究の知見によるものである [21]. ただし，通常 5 段階で表現される優先度は，デフォルトの “P3” が用いられる場合が非常に多いため，本研究では，優先度が “P1” と “P2” を優先度 “高”， “P3” を優先度 “中”, “P4” と “P5” を優先度 “低” とし 3 段階に分けた.

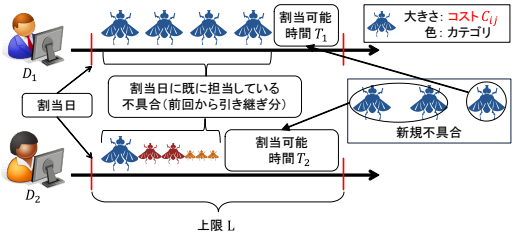

図 1 一定期間内に修正作業に使える時間の求め方

プリファレンス（開発者の適性）：0-1 整数計画法の目的関数では，目的変数に係数を設定する. 本研究では係数として，プリファレンス P を用いる. プリファレンスとは，修正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，カテゴリ k の修正タスクの総数に対する開発者 D_i の修正実績数の比をプリファレンスとする. 例えば，“UI” コンポーネントを対象とする優先度 “高” の修正タスクが過去に 10 件存在し，その内，開発者 D_i が 5 件を担当したことがあれば，開発者 D_i に対するカテゴリ k ($UI*$ 高) の不具合 B_j のプリファレンス P_{ij} は，0.5 として計算される.

不具合の修正コスト：不具合修正に必要な時間はどの開発者がどの不具合を修正するかによって異なる. 本研究では，開発者 D_i が不具合修正タスク B_j を修正する際に必要とされる時間を**不具合修正コスト** C_{ij} と定義する. C_{ij} は過去に開発者 D_i がカテゴリ k の不具合修正タスクに要した時間の中央値とした. コストの算出に算術平均ではなく，中央値を用いた理由は，OSS 開発における不具合修正時間の分布に偏りがあるためである.

なお，先行研究 [7] と同様に，過去の修正タスク個々の修正時間は，以下の式から求めた. 割当日時は不具合修正を完了させた開発者に割当てられた日時を指す. 本研究の修正時間にはその開発者に割当てられる以前の修正時間（再

[*2] lp_solve 5.5: http://lpsolve.sourceforge.net/5.5/

割当の時間）は含めないとする．

$$\text{修正時間（日）} = \text{修正完了日時} - \text{割当日時} + 1\text{ 日} \quad (4)$$
*ただし，小数点以下は切り捨てる

上限：開発者が一定期間内に修正作業に使える時間には限りがあると考えるのが自然である．本研究では，開発者 D_i が修正作業に使える時間を考慮した不具合の割当てを行う．図 1 は，修正作業に使える時間の求め方を示した概略図である．修正作業に使える時間は**割当可能時間** T_i から求める．割当可能時間 T_i は，あらかじめ設定する**上限** L（時間）と，新規の修正タスク割当時点で既に開発者 D_i が担当している不具合の修正コスト C_{ij} から求める．

新規に割当てる修正タスクのコストの合計が T_i を超えないようにすることで，特定の開発者へ修正タスクが極端に集中するのを防ぐ効果を期待できる．なお，上限 L はプロジェクトによって大きさを変えることができる．

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．

目的変数：x_{ij} とは，開発者 D_i に不具合 B_j を担当させるかどうかを表し，0 の場合は開発者 D_i に不具合 B_j を割当てないことを，1 の場合は割当てることを意味する．

$$x_{ij} \in \{0, 1\} \quad (5)$$

目的関数：各不具合に対する各開発者のプリファレンスと目的変数の積の総和を最大化する．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ことを意味する．

$$Maximize : \sum_{i=1}^{m}\sum_{j=1}^{n} P_{ij}x_{ij} \quad (6)$$

制約条件：本研究では，目的関数に対する制約条件として，以下の 2 つを課す．(7) 式は，一部の有能な開発者にタスクが集中するのを防ぐための制約である．(8) 式は，同一の不具合を複数人の開発者が同時に修正することを避けるための制約である．

- 各開発者に割当てるタスクに要するコストの合計は割当可能時間を超えないこと

$$\sum_{j=1}^{n} C_{ij}x_{ij} \leq T_i \qquad (i = 1, 2, ..., m) \quad (7)$$

- 1 つの不具合を担当する開発者は 1 人以下であること

$$\sum_{i=1}^{m} x_{ij} \leq 1 \qquad (j = 1, 2, ..., n) \quad (8)$$

以上のように，本論文ではバグトリアージを一種のマルチナップサック問題として定式化し，これを解くことで計画の最適化を行う．なお，本論文では開発者の一人一人の適性を反映できるよう，どの開発者がどの不具合を担当するかによってプリファレンスやコストが異なるというモデルになっている点に注意されたい．つまり，一般的なマルチナップサック問題（3.1 節参照）と目的関数や制約条件のかたちが若干異なっている（v_j が P_{ij}，w_j が C_{ij} となっている）．

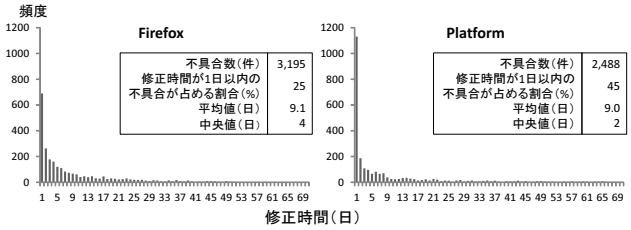

図 2 不具合修正時間のヒストグラムと統計量

3.3 提案手法の適用手順

提案手法の適用手順は以下の通りである．

Step 1: パラメータの設定 手法の適用前にあらかじめ上限 L を設定する．L の値で各開発者の割当可能時間 T_i を初期化する．

Step 2: プリファレンスとコストの算出 開発者 D_i がカテゴリ k の不具合修正タスク B_j に必要なコスト C_{ij} とプリファレンス P_{ij} をすべて算出する．

Step 3: 割当て待ち不具合の追加 新たに報告された不具合を割当て待ち不具合として待機させる．

Step 4: 0-1 整数計画法の適用 前節で述べた 0-1 整数計画法を用いて，割当て待ち不具合を開発者に割当てる．割当てられた不具合は割当て待ち不具合から外す．

Step 5: T_i の更新 Step 4 で割当てられてた不具合のコスト分だけ各開発者の割当可能時間 T_i を減らす．

Step 6: 次の割当て日に進む（Step.3 へ） 次の割当て日（n 日後）まで時間を進め，各開発者の T_i を n（日）増やす（ただし，T_i は Step 1 で設定した L より大きくしない）．割当終了予定日になるまで Step 3 以降を繰り返す．

4. ケーススタディ

本章では，Research Question に取り組むために行ったケーススタディについて述べる．

4.1 準備

4.1.1 データセット

本論文では，2 つの大規模 OSS プロジェクト (Mozilla Firefox, Eclipse Platform) を対象にしたケーススタディを行う．両プロジェクトは既存研究の多くが分析対象とされており [1–5, 7, 10, 13, 15, 21]，本ケーススタディで得られる結果の妥当性を確保できる．

表 2 に本ケーススタディで用いたデータセットの概要を，図 2 にデータセット全体の不具合修正時間のヒストグ

表 2　データセット

プロジェクト	データセット	期間	解決済み不具合（件）	対象不具合数（件）	コンポーネント数	カテゴリー数
Firefox	学習データ	2002/10/1〜2011/9/30	10,165	2,536	28	60
	評価データ	2011/10/1〜2012/9/30	1,648	659	31	39
Platform	学習データ	2002/10/1〜2011/9/30	21,802	1,910	19	28
	評価データ	2011/10/1〜2012/9/30	932	578	17	23

表 3　データセットに含まれるアクティブな開発者

プロジェクト	全開発者	アクティブ開発者
Firefox	734	21
Platform	301	23

ラムおよび統計量を示す．

各プロジェクトから収集した不具合データの内，修正 (FIXED) された不具合，かつ，修正時間および修正者を特定できる不具合のみを用いている．なお，再割当が発生した不具合については，最後に不具合修正を完了させた開発者が修正に取り組んだ時間のみを計測している．また，不具合報告の中には，数年間放置された後に修正される不具合も存在するため，各不具合の修正時間の分布を箱ひげ図で確認し，外れ値となる不具合はデータセットから除外している．

本ケーススタディでは，既存手法および提案手法により修正タスクを開発者に割当てる実験を行なうが，OSS プロジェクトの開発者は比較的短期間でプロジェクトを去ることが知られている [22] ため，各プロジェクトに在籍したすべての開発者を対象としてタスク割当を行うのは現実的ではない．また，すべての開発者が活発に不具合修正を行っている訳ではない [21] ため，修正タスクを担当できる見込みのある開発者のみにタスクを割当てる必要がある．そこで本研究では，既存手法および提案手法の最初のタスク割当て日を基準とし，半年以内に 6 回以上（一ヶ月に 1 回程度を想定）の修正タスクを完了させた開発者を「アクティブ開発者」と定義し，タスク割当ての対象とする（表 3）．なお，タスク割当ての精度を保証するために，対象の開発者以外が修正した不具合報告はデータセットから除外した．

本ケーススタディでは，2002 年 10 月 1 日〜2011 年 9 月 31 日の間に報告された不具合を学習データ，データセットの最後の 1 年である 2011 年 10 月 1 日〜2012 年 9 月 30 日の間に報告された不具合を評価データとして，既存手法および提案手法を比較する．

4.1.2　実験の手順

本ケーススタディでは，既存手法および提案手法によりタスクを割当てる実験を行ない，得られた割当結果を用いて修正時間を算出する．実験の概要を図 3 に示す．

本実験では，実験上の日付を用意し，その日付に従って不具合報告を再現する．不具合データには報告日時が記されており，報告日時が実験上の日付と同じであれば報告されたと見なし，該当する不具合を提案手法および既存手法で割当てていく．該当する不具合の割当が済めば，実験上の日付を進め，再び該当する不具合の割当てを繰り返す．本実験では 365 日分の割当てを行なう．

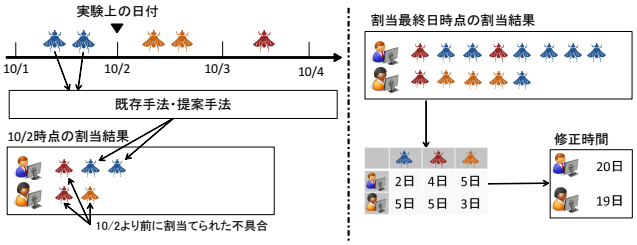

図 3　実験の概要

全ての日数分の割当が終了すると，次に修正時間の算出を行なう（図 3 右）．修正時間は学習データから個々の開発者における各カテゴリの不具合の修正時間の中央値（すなわち，コスト C_{ij}）とする．

4.1.3　実験環境と設定

本ケーススタディでは，0-1 整数計画法を用いてタスク割当て問題の解を求めるために，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 を用いた．また，CPU: Intel Core i7 2.30 GHz，OS: CentOS 6.2，メモリ: 2GB の計算機を用いて実行した．

提案手法を適用するためには，あらかじめ上限 L を設定する必要がある．本研究ではデータセットに含まれる不具合の修正に要した時間の第 3 四分位値を求め，その値を切り上げた値とし，Firefox では L=10，Platform では L=9 と設定した．また，既存手法には，Anvik らの機械学習に基づくバグトリアージ手法の内，最も精度の高い推薦を行える SVM ベースの手法 [3] を用いた．

4.2　実験と結果

RQ1: 既存手法は，特定の開発者へ負荷を集中させる傾向があるか？提案手法ではその問題を緩和できるか？

動機: 既存手法は一部の有能な開発者に集中して修正タスクを割当てる可能性がある．既存手法と提案手法で一部の開発者に修正タスクが集中しすぎないかどうかを確認する．

アプローチ: テストデータにおける全期間（365 日間）と不具合報告の最も多かった月（繁忙期：31 日間）の 2 つの期間を用いて，既存手法と提案手法によるタスク割当の結果を比較する．繁忙期を設定した理由は，修正作業に使え

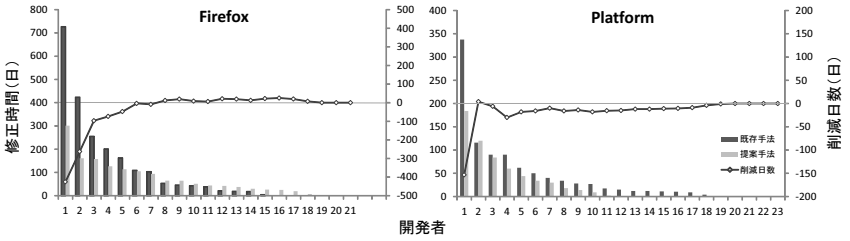

図 4 全期間における各開発者の修正時間（既存手法 vs. 提案手法）

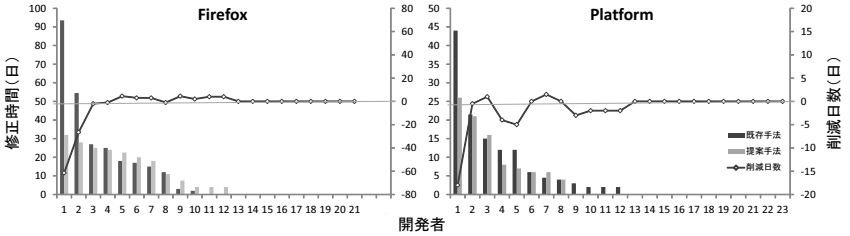

図 5 繁忙期における各開発者の修正時間（既存手法 vs. 提案手法）

る時間の上限を考慮しない既存手法と上限を考慮する提案手法とで，割当て結果に顕著な違いが見られると考えたためである．

結果: 図 4 および図 5 はそれぞれ，テストデータにおける全期間および繁忙期に各アクティブ開発者が取り組んだタスク量（修正時間）を示している [*3]．なお，既存手法と提案手法とでは開発者に割当てるタスク数が異なるため，図中に示した結果は，両手法で共通に割当てられたタスクから修正時間を算出し比較したものである（割当て結果の詳細については表 4 を参照されたい）．

図 4 の全期間での比較では，既存手法を用いた場合，Firefox では 2 名の開発者に合計で設定期間（365 日）以上を要するタスクが割当てられており，一部の開発者に多くの負荷がかかっていることが見て取れる．また，Platform においても 1 名の開発者に設定期間に近いタスクが割当てられている．図 5 の繁忙期では，Firefox で 2 名，Platform で 1 名の開発者に合計で設定期間（31 日）以上を要するタスクが割当てられている．

一方，提案手法を用いた場合，全期間と繁忙期の両期間において，開発者間でタスク量のばらつきは見られるものの，設定期間以上のタスク割当ては生じていない．また，提案手法を用いることで，既存手法において負荷が大きい開発者ほど，修正時間が大きく削減されている（右側の縦軸）ことが分かる．

これらの結果から，提案手法は開発者が修正できるタスク量を考慮して不具合割当を行っており，一部の開発者へタスクが集中するのを緩和できるといえる．

[*3] 横軸の番号は，開発者のタスク量（修正時間）を降順に並べた時の順番を表すものであり，開発者を特定するものではないことに注意されたい．図 4 および図 5 以外の結果においても同様である．

既存のバグトリアージ手法は，一部の開発者に負荷を集中させる傾向がある．既存手法に比べ，提案手法は負荷が集中するのを緩和する効果がある．

RQ2: 提案手法は，プロジェクト全体の不具合修正時間の短縮化に寄与するか？

動機: RQ1 の結果から，提案手法は既存手法に比べ，特定の開発者へのタスク集中を緩和できることが分かった．しかし，プロジェクト全体の不具合修正時間を短縮化できないのであれば，タスク割当を分散させる意義が薄れてしまう．そこで，提案手法がプロジェクト全体の不具合修正時間を短縮化できるかどうかを検証する．

アプローチ: 現状のタスク割当て（OSS プロジェクトで行われている実際のタスク割当て）による修正時間，既存手法と提案手法で実験して得られた修正時間をそれぞれ比較する．

結果: 表 4 に，現状の割当方法による割当結果，既存手法および提案手法を用いた割当結果を示す．表中の「平均修正時間」とは，タスク 1 件あたりの修正時間（平均値）である．また，前述した通り，それぞれの方法で割当てられるタスクの数が異なるため，3 つの方法で共通して割当てたタスクのみにした結果を「共通化後」として示している．比較のため，以下では共通化後の結果について議論する．

Firefox では，プロジェクト全体としての合計修正時間は，現状の割当方法で 2,920 日， 既存手法で 2,220 日，提案手法で 1,471 日となり，既存手法は現状の割当方法に比べて約 24%，提案手法は現状の割当方法に比べて約 50%の修正時間を削減できることが分かった．また，提案手法を用いた場合，既存手法に比べ約 34%の修正時間を削減できることが分かった．

Platform ではプロジェクト全体としての合計修正時間は，現状の割当方法で 3,433 日， 既存手法で 967 日，提案

表 4 現状の割当方法，既存手法および提案手法によるタスク割当結果の比較

プロジェクト		Firefox			Platform		
手法		現状の割当方法	既存手法	提案手法	現状の割当方法	既存手法	提案手法
割当てた不具合（件）	共通化前	659	439	440	578	386	572
	共通化後	377			385		
割当てた開発者（人）	共通化前	16	17	18	20	19	12
	共通化後	16	15	18	20	19	11
合計修正時間（日）	共通化前	5,816	3,091	1,685	4,893	995	901
	共通化後	2,920	2,220	1,471	3,433	967	599
平均修正時間（日）	共通化前	8.8	7.0	3.8	8.5	2.6	1.6
	共通化後	7.7	5.9	3.9	8.9	2.5	1.6

表 5 条件の有無による 4 つのモデル

	L：無	L：有
P：無	**モデル A:** プリファレンスと上限を設定せず，修正時間の短い開発者に不具合を優先的に割当てるモデル	**モデル B:** 上限を設定した条件下で，修正時間の短い開発者に不具合を優先的に割当てるモデル
P：有	**モデル C:** プリファレンスのみ設定し，カテゴリ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	**モデル D:** 上限を設定した条件下で，プリファレンスの合計が最大となるように不具合を割当てるモデル (提案手法)

手法で 599 日となり，既存手法は現状の割当方法に比べて約 72%，提案手法は現状の割当方法に比べて約 83%の修正時間を削減できることが分かった．また，提案手法を用いた場合，既存手法に比べ約 38%の修正時間を削減できることが分かった．

提案手法は，現状のタスク割当て方法に比べ 50%から 83%，既存手法に比べ 34%から 38%不具合修正時間を削減できることが分かった．

RQ3: 提案手法で用いる設定（プリファレンスと上限）が，タスク割当ての最適化にどのように寄与するか？

動機: RQ2 より，提案手法がプロジェクト全体の不具合修正時間の短縮化に効果があることが分かった．しかし，修正タスクに対する開発者の適性と，一定期間内に修正作業に使える時間の上限を考慮するためにそれぞれ用いたプリファレンス P と上限 L が，タスク集中の回避にどのように寄与するかの詳細については不明なままである．

アプローチ: P の有無，L の有無により 4 種類のモデルを構築し，それぞれのモデルを比較することで，タスク割当て問題におけるプリファレンスおよび上限の効果を調べる．各モデルの特徴は，表 5 のようにまとめることができる．

結果 表 6 に，各モデルを用いてタスク割当てを行った結果を示す．以下では，P と L がタスク集中の回避にどのように寄与したのかを，表 6 の結果から議論する．

モデル A とモデル B との比較: プリファレンスを設定せず上限の有無のみで比較する．モデル A（L 無し）に比べモデル B（L 有り）のタスク割当て人数は，両プロジェクトにおいて増加している（Firefox：1 人から 15 人，Platform：1 人から 14 人）．上限の設定はタスク割当てを分散させる効果があるといえる．

モデル A とモデル C との比較: 上限を設定せずプリファレンスの有無のみで比較する．モデル A（P 無し）に比べモデル C（P 有り）のタスク割当て人数は，全プロジェクトにおいて増加している（Firefox：1 人から 9 人，Platform：1 人から 12 人）．プリファレンスの設定により，修正実績数だけではなく修正タスクへの開発者の適性が反映される．上限の設定と同様に，プリファレンスの設定はタスク割当てを分散させる効果がある．

モデル B とモデル D との比較: 上限を設定した上で，プリファレンスの有無で比較する．モデル B（P 無し）に比べモデル D（P 有り）のタスク割当て人数は，Firefox では増加し，Platform では割当人数が減少した．（Firefox：15 人から 18 人，Platform：14 人から 12 人）．上限が先に設定されている場合は，プリファレンスの設定はタスクの分散に必ずしも有効ではない．

モデル C とモデル D との比較: プリファレンスを設定し，上限の有無で比較する．モデル C（L 無し）に比べモデル D（L 有り）のタスク割当て人数は，Platform では変化は無かったが，Firefox では増加している（Firefox：9 人から 18 人，Platform：12 人から 12 人）．モデル D では，プリファレンスの設定に加え，上限の設定により，さらにタスク割当てを分散できたことを示している．プリファレンスを設定した上で，上限を設定することはタスク分散の効果を高める可能性がある．

プリファレンスおよび上限の設定は，タスク割当ての分散に効果がある．特に上限の設定がプリファレンスの設定よりタスクの分散に大きく寄与する．

5. 考察

5.1 提案手法の適用範囲

本研究ではプロジェクト全体での不具合修正時間の短縮化を目的とするバグトリアージ手法を提案した．しかし，

表 6 モデルごとのタスク割当ての結果

プロジェクト		Firefox				Platform			
モデル		A	B	C	D	A	B	C	D
適性		無	無	有	有	無	無	有	有
上限		無	有	無	有	無	有	無	有
割当てた不具合（件）	共通化前	659	659	559	440	578	578	575	572
	共通化後	440	440	440	440	572	572	572	572
割当てた開発者（人）	共通化前	1	15	10	18	1	14	12	12
	共通化後	1	15	9	18	1	14	12	12
合計修正時間（日）	共通化前	4,284	3,066	4,897	1,685	5,780	1,186	997	901
	共通化後	2,860	2,022	2,115	1,685	5,720	1,174	910	901
平均修正時間（日）	共通化前	6.5	4.7	8.8	3.8	10.0	2.1	1.7	1.6
	共通化後	6.5	4.6	4.8	3.8	10.0	2.1	1.6	1.6

OSS 開発では，時期ごとに不具合の報告量や開発者数，不具合修正に必要な時間は異なるため，提案手法がプロジェクトのあらゆる状況に適しているとは限らない．本節では，提案手法がどのような状況に適しているかを，追加分析の結果とともに考察する．

5.1.1 不具合報告状況への対応

提案手法は，特定の開発者へタスクが集中するのを回避しつつ，可能な限り適任の開発者にタスクを割当てる．プロジェクトへ報告される不具合が多い状況と少ない状況とで，開発者の負荷がどのように変化するかを確認する．

図 6 は，不具合報告の最も多かった月（繁忙期 31 日間）と，最も少なかった月（閑散期 31 日間）のそれぞれで，各開発者に割当てられたタスクの量（修正時間）を示したグラフである．なお，提案手法によりタスクが割当てられなかった一部の開発者は図から省略している．タスクを割当てられる開発者の数は，両プロジェクトで共通して，繁忙期に比べて閑散期の方が少ないことが見て取れる．また，繁忙期に比べて閑散期では，修正時間のばらつき大きいことが見て取れる．

図 6 のような結果になった理由は次のように考えられる．閑散期には，開発者に割当てられるタスク量が上限 L に達しない．そのため，従来手法と同様，最も適任の開発者にのみタスクを割当てることができる．一方，繁忙期には，開発者に割当てられるタスク量が上限 L に達する開発者が多いため，上限 L を越える分のタスクが次に適任の開発者に分散して割当てられる．

これらの結果から，閑散期に提案手法を適用する場合には，最も適任の開発者にのみタスクを割当てるという，従来のバグトリアージ手法と同様の利点があり，繁忙期に提案手法を適用する場合には，プリファレンスが大きい開発者を中心にタスクを割当てつつ残りのタスクをその他の開発者に分散させることで開発者の負荷状況をコントロールできるといえる．したがって，繁忙期と閑散期が存在するプロジェクト（例えば，メジャーバージョンのリリースの前後で不具合報告数が大きく増減するプロジェクトなど）

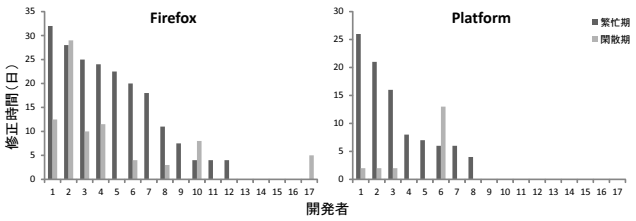

図 6 提案手法を用いた割当結果（繁忙期 vs. 閑散期）

において，提案手法は特に有用であると考える．一方，いずれの開発者も上限 L に達することのないような小規模プロジェクトにおいては，開発者推薦の精度のみを追求した従来のバグトリアージ手法（ [3] など）を利用するのが適していると思われる．

5.1.2 開発者の負荷状況への対応

ケーススタディでは，上限 L の大きさをデータセットに含まれる不具合修正時間の第 3 四分位値（Firefox では 10 日，Platform では 9 日）として固定した．上限 L の大きさによって開発者の負荷は大きく変わると考えられるため，L の値を変化させてその効果を詳細に調べる．

図 7 は，上限 L の値を変化させた時の開発者のタスク量を示したものである．上限 L を大きくした時の開発者のタスク量の変化は一様であったため，紙面の都合上，10，20，30（日）の値だけを示す．

Firefox では，L が大きくなるにつれて，タスク量の最も大きい開発者とその他の開発者とのタスク量との差が小さくることが見て取れる．一方，Platform では，各開発者のタスク量は多少変化するものの，L の変化によりタスク量の分布は大きく変わらないという結果となった．

図 8 は，上限 L と割当てた不具合数，合計修正時間の関係を示したものである．図から見て取れるように，Firefox では，上限 L が大きくなるにつれ，割当て不具合数および合計修正時間が増えるのに対し，Platform では，上限 L が大きくなっても，割当て不具合数にはほとんど変化は見られない．また，合計修正時間についても，Firefox ほどの大きな変動はみられない．Platform では，多くのタスクのコ

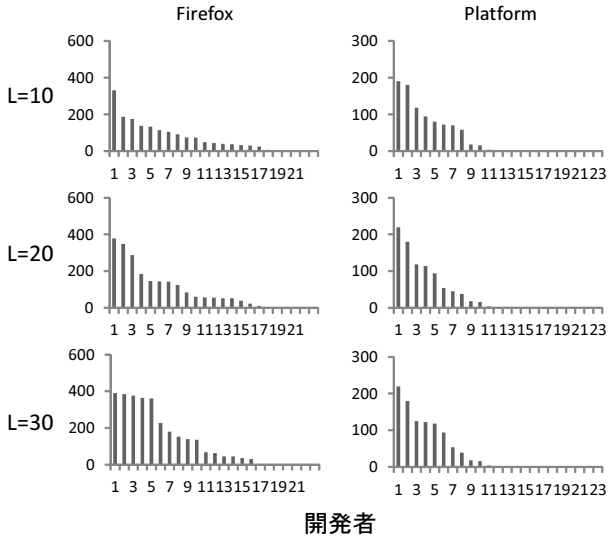

図 7 L の大きさ別割当結果

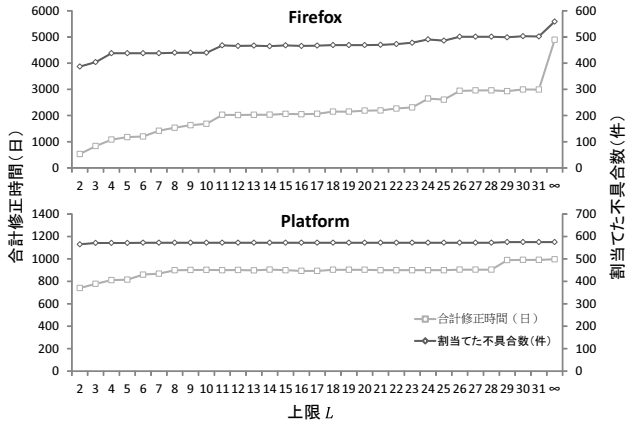

図 8 L の大きさと割当てた不具合数および合計修正時間の関係

ストは L より小さいため，L を変化させても一部の開発者に優先して割当てを行うという状況はあまり改善しないことが原因と考えられる．一方，Firefox では，Platform に比べてタスクのコストが L より大きい場合が多く，L を大きくすることで割当て可能な開発者が増えたと考えられる．

以上の結果から，上限 L は本来，開発者への負担を加味しながらプロジェクト管理者が決定すべきであるが，不具合修正に要するタスク量の分布によって，Firefox のように L を大きくすることが望ましいプロジェクトが存在することが分かった．また，Platform のように，不具合修正に要するタスク量が小さいものが多くを占める場合は，L を大きくしても負荷分散の効果が得られない場合もあり，実験で行ったような第 3 四分位を用いた機械的な設定ではなく，過去の不具合修正タスクを調査して L を設定するのが望ましいといえる．

5.2 修正時間算出方法の妥当性

まず，修正時間の算出方法の妥当性について検証する．

表 7 コスト算出方法の妥当性の検証

	現状の割当方法（日）	実験（日）	差（日）
Firefox	4,451	4,660	-209
Platform	4,787	1,982	2,806

表 7 は，現状の割当方法で実際に要した不具合修正時間の合計と，現状の割当方法で割当てられた修正タスクを提案手法の修正時間算出方法（不具合修正のコスト C_{ij}：開発者 D_i がカテゴリ k の不具合修正タスク B_j を完了するのに要した時間の中央値）で求めた不具合修正時間の合計とを比較したものである [*4]．

まず，Firefox では，提案手法の方が約 209 日とやや大きく修正時間を算出しているものの，約 4%の誤差であり修正時間の算出方法については概ね妥当であったといえる．

一方，Platform では，提案手法の方が修正時間を 2,800 日以上小さく見積もっている．同じ修正時間算出方法を用いたケーススステディにおいての既存手法と提案手法とを比較した結果と結論自体には影響を与えないが，提案手法を実際に Platform に適用する際には，本論文における修正時間の算出方法は大きな問題があるといえる．実際に，実験の誤差（2,806 日）を RQ2 の提案手法の合計修正時間に足し合わせると，3,405 日になり（599 日＋ 2,806 日），現状の割当方法の修正時間（3,433 日）との差は 28 日（1 %の削減）に留まっている．

Platform において修正時間の見積り誤差が大きくなった理由は，Platform では不具合修正時間の分散が大きかったことに起因するものと考えている．図 2 で示したように，Platform では，約 45%の不具合が 1 日以下で修正されている．一方，修正時間の平均値は 9 日である．そのため，不具合修正のコストが 1 日とした開発者を多く想定することとなり，修正時間が過剰に短く算出されたものと思われる．前述した図 8 からも，コストの過小見積りが原因となって，L が小さな場合でもほとんどの不具合が割当てられていることが見て取れる．

以上から，Platform のようなコストの小さな不具合が多くを占めるプロジェクトでは，本論文で用いた修正時間算出方法には限界があるといえる．実験の精度向上および実際の適用を考える上で，修正時間（コスト）算出方法は今後改良する必要がある．

6. おわりに

本研究では，大規模 OSS 開発における不具合修正時間の削減を目的としたバグトリアージ手法を提案した．提案手法は，0-1 整数計画法に基づいてタスク割当てを最適化する点に特徴がある．

[*4] 提案手法の修正時間算出方法に基づくため，現状の割当方法で割当てられたタスクの修正時間を計算できない場合がある．計算可能なタスクのみを対象として比較しているため，表 7 の比較結果は，表 4 の合計修正時間とは異なる数値が示されている．

Mozilla Firefox および Eclipse Platform プロジェクトを対象としたケーススタディを行った結果，提案手法について以下の 3 つの効果が確認できた．

- 特定の開発者へタスクが集中するという問題を緩和できること
- プロジェクト全体としての不具合修正時間を，人手によるバグトリアージに比べて 50–83%，既存手法と比べて 34–38% 削減できること
- 提案手法で用いた設定であるプリファレンス（開発者の適性）と上限（開発者が取り組むことの出来る時間の上限）がそれぞれ，タスクの分散効果に同程度寄与すること

本研究の今後の課題は，不具合修正コストの算出方法を改良すること，また，機械学習による方法を用いてプリファレンスの精度を向上させることなどが挙げられる．

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

参考文献

[1] Anvik, J., Hiew, L. and Murphy, G. C.: Who should fix this bug?, *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pp. 361–370 (2006).

[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] 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).

[4] 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).

[5] 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).

[6] 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).

[7] Park, J., Lee, M., Kim, Jinhan, H. S. 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).

[8] Lin, Z., Shu, F., Yang, Y., Hu, C. and Wang, Q.: An empirical study on bug assignment automation using Chinese bug data, *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM'09)*, pp. 451–455 (2009).

[9] Bortis, G. and van der Hoek, A.: PorchLight: A Tag-based Approach to Bug Triaging, *Proceedings of the 2013 International Conference on Software Engineering (ICSE'13)*, pp. 342–351 (2013).

[10] Tamrawi, A., Nguyen, T. T., Al-Kofahi, J. M. and Nguyen, T. N.: Fuzzy Set and Cache-based Approach for Bug Triaging, *Proceedings of The joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11)*, pp. 365–375 (2011).

[11] Shokripour, R., Anvik, J., Kasirun, Z. M. and Zamani, S.: Why So Complicated? Simple Term Filtering and Weighting for Location-based Bug Report Assignment Recommendation, *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13)*, pp. 2–11 (2013).

[12] Naguib, H., Narayan, N., Brgge, B. and Helal, D.: Bug report assignee recommendation using activity profiles., *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13)*, pp. 22–30 (2013).

[13] Sun, C., Lo, D., Wang, X., Jiang, J. and Khoo, S.-C.: A discriminative model approach for accurate duplicate bug report retrieval, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, pp. 45–54 (2010).

[14] 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).

[15] 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).

[16] Ohira, M., Hassan, A. E., Osawa, N. and Matsumoto, K.: The Impact of Bug Management Patterns on Bug Fixing: A Case Study of Eclipse Projects, *Proceedings of 28th IEEE International Conference on Software Maintenance (ICSM'12)*, pp. 264–273 (2012).

[17] 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).

[18] Martello, S. and Toth, P.: *Knapsack Problems: Algorithms and Computer Implementations*, John Wiley & Sons, Inc., New York, NY, USA (1990).

[19] 福島雅夫：数理計画法入門，朝倉書店，東京 (1996).

[20] 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).

[21] 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).

[22] Bird, C., Gourley, A., Devanbu, P., Swaminathan, A. and Hsu, G.: Open Borders? Immigration in Open Source Projects, *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR'07)*, p. 6 (2007).