様　式　Ｃ－１９、Ｆ－１９－１、Ｚ－１９　（共通）

科学研究費助成事業　　研究成果報告書

令和　元 年　 ６ 月　 ６ 日現在

機関番号：３４３１５
研究種目：基盤研究(B)（一般）
研究期間：2015～2018
課題番号：１５Ｈ０２６８５

研究課題名（和文）フレームベースリファクタリング環境の構築

研究課題名（英文）A study on Frame-Based Refactoring

研究代表者
丸山　勝久（MARUYAMA, KATSUHISA）

立命館大学・情報理工学部・教授

研究者番号：３０３３００１２

交付決定額（研究期間全体）：（直接経費）　　10,600,000円

研究成果の概要（和文）：本研究では、テストケースの集合で構成する空間的フレームを活用することで、リファクタリングにおける外部的振る舞いの保存に対する曖昧さが排除できることを示し、不足するテストケースを自動生成する手法を提案した。また、自動リファクタリングの適用途中に開発者による手動のコード編集を許可する時間的フレームを導入することで、リファクタリングの中断と再開による遅延適用を可能とするツールを構築し、使用性の観点からその有用性を示した。さらに、リファクタリングの適用とそれに伴うコード変化を正確かつ理解しやすい表現で記録するツールプラットフォームを構築した。

研究成果の学術的意義や社会的意義
社会の要求や技術の進歩に迅速に対応してソフトウェアを進化させていくために、リファクタリングは必須の作業である。本研究では、ソフトウェアの外部的振る舞いの保存という曖昧な概念をフレームにより明確に定義可能とすることで、自動リファクタリングにおける安全性を議論することを可能とした。さらに、外部的挙動をフレームで捉えることで、新たなリファクタリング支援手法の考案に成功した。これにより、ソフトウェア開発現場や保守現場において自動リファクタリングの適用機会が増加することが期待できる。

研究成果の概要（英文）：In this research study, we showed that a spatial frame defined as a set of test cases is capable of relaxing the definition of the behavior preservation in refactoring. Additionally, we proposed a mechanism that automatically generates new complementary test cases that are likely to help a programmer define spatial frames. A temporal frame separates the time period in which the behavior preservation should be guaranteed from the whole process of automated refactoring. To increase the applicability of automated refactoring, a postponable refactoring tool employing temporal frames allows a programmer to suspend the execution of the applied refactoring if its preconditions are not satisfied and to restart the suspended refactoring once all the preconditions are satisfied. We also developed a platform that can record accurate and intelligible textual changes of source code evolution, especially including code changes by refactoring.

研究分野：ソフトウェア工学

キーワード：ソフトウェアリファクタリング　ソフトウェア進化　技術的負債　不吉な臭い　ソフトウェア開発環境　プログラム解析　プログラム理解

様　式　Ｃ－１９、Ｆ－１９－１、Ｚ－１９、ＣＫ－１９（共通）

１．研究開始当初の背景

ソフトウェア進化においてソースコードの劣化は避けられない。このような状況において、リファクタリングは必須の作業である。リファクタリングとは、劣化した既存コードの外部的振る舞いを変えずに内部構造を変換することで、そのコードの理解容易性や変更容易性を改善する作業である。研究開始当初においても、さまざまな統合開発環境において自動リファクタリングが提供されていたが、実際の開発および保守現場における自動リファクタリングの利用率はそれほど高くないことが報告されていた。当然ながら、手動でリファクタリングを実施した場合、その作業途中でエラーが混入する可能性がある。社会に対して信頼性の高いソフトウェアを提供するためには、自動リファクタリングを開発現場や保守現場に根付かせていくべきである。

リファクタリングの特徴は、その適用前後でコードの外部的振る舞いが必ず保存されることである。これは、同一の入力値集合に対してリファクタリング前後で出力値集合が同一であることを指す。外部的振る舞いの保存が保証されると、変換前のコードが実行可能であれば、変換後のコードを実行した際も変換前と同一の実行結果が得られる。この特徴によって、開発者は安心してリファクタリングを実行できる。しかしながら、一方で、外部的振る舞いの保存に対する解釈は開発者によってまちまちであるため、開発者の期待する保証と自動リファクタリングの保証する保証に齟齬が生じることがある。このような齟齬の発生により、開発者はリファクタリングが外部的振る舞いの保存を保証するという主張に不信感を持ち、自動リファクタリングの利用を避ける傾向にあった。

また、当時のリファクタリングツールの多くは、外部的振る舞いの保存を保証するために、対象コードに関する事前条件を検査する手法を採用していた。この手法では、変換前のコードに対して事前条件を検査し、安全なコード変換だけを適用可能とすることで外部的振る舞いの保存を保証する。その一方で、安全なコード変換を達成するための前提条件は厳しくなる傾向になり、適用可能な自動リファクタリング操作は大きく制限されていた。

このような状況である一方で、アジャイルソフトウェア開発の促進という観点から、開発現場や保守現場におけるリファクタリングに対する期待は大きく、自動リファクタリングの浸透を加速させる技術を確立していくことが強く求められていた。

２．研究の目的

本研究では、開発者にとってツールが保証する外部的振る舞いの保存という概念に対する解釈が曖昧である点が、自動リファクタリングの普及を妨げる最大の原因であるとみなし、それをフレーム(frame)で定義することで曖昧さが排除できると考えた。フレームとは、リファクタリングにおいて開発者が意識しているもの(内部)を、意識していないもの(外部)から分離するための仕組みを指す。フレームは開発者の意識の内部と外部を分離する境界であるため、フレームの外部から見たフレーム内部の振る舞いがソフトウェアの外部的振る舞いと見なせる。リファクタリングやその適用に対してフレームを用意することで、同じソフトウェアに対してさまざまな外部的振る舞いを柔軟に定義することが可能となる。本研究では、自動リファクタリングにフレームの概念を導入することで、その適用可能性の向上を目指す。さらに、フレームを積極的に活用したリファクタリングツールをプログラミング環境に組み込むことで、実際のソフトウェア開発現場や保守現場における自動リファクタリングの浸透を目指す。

３．研究の方法

開発者は、必ずしもソフトウェア全体のすべての入出力で外部的振る舞いを捉えているわけではない。たとえば、ライブラリの提供者と利用者が明確に区別されている場合、ライブラリの提供する API(Application Programming Interface)が外部と内部の境界であり、その入出力で外部的振る舞いを捉えるのが一般的である。つまり、ある開発者にとっては、ソフトウェア全体ではなく、ライブラリの外部から見た入出力が同一であることだけが重要である。一方、ライブラリの更新と同時にライブラリを利用している開発者にとって、ライブラリの提供する API が必ずしも外部的振る舞いの保存を保証する必要はない。なぜなら、このような開発者は、ライブラリ内部のコード変更を即座に知ることができ、それに合わせてライブラリを利用するコードを自由に変更できるからである。このように、開発者の立場によって BP の解釈は異なると考えるのが妥当である。

利用者から見た外部的振る舞いも一意ではない。たとえば、Security-aware refactoring では、正当な権限を持つ利用者の振る舞いを保存しつつ、正当な権限を持たない利用者(攻撃者)の振る舞いを制限し、攻撃を無効化する。これは、通常の利用者から見た入出力と攻撃者から見た入出力を分離することではじめて実現できるリファクタリングである。本研究では、まず、このような空間的フレーム(図 1 における SF)を洗い出す。

リファクタリング時に開発者の意識する範囲は空間だけではない。通常の開発において、常に外部的振る舞いの保存を保証するコード変換だけを適用して、既存コードの改善を行うことは現実的でない。たとえば、コード変換の途中において、特定のクラスやメソッドに対するアクセス範囲の変更や、オブジェクトに対する参照の切り替えは頻繁に実行されている。このような事実に基づくと、リファクタリングとは、その作業全体において外部的振る舞いの保存を保証することであり、作業途中の一部区間ではそれを保証しなくてもよいと考えることができ、

リファクタリングの適用機会の増加が期待できる。ここでは、外部的振る舞いの保存に影響を与える区間を時間的フレーム(図 1 における TF)として洗い出す。

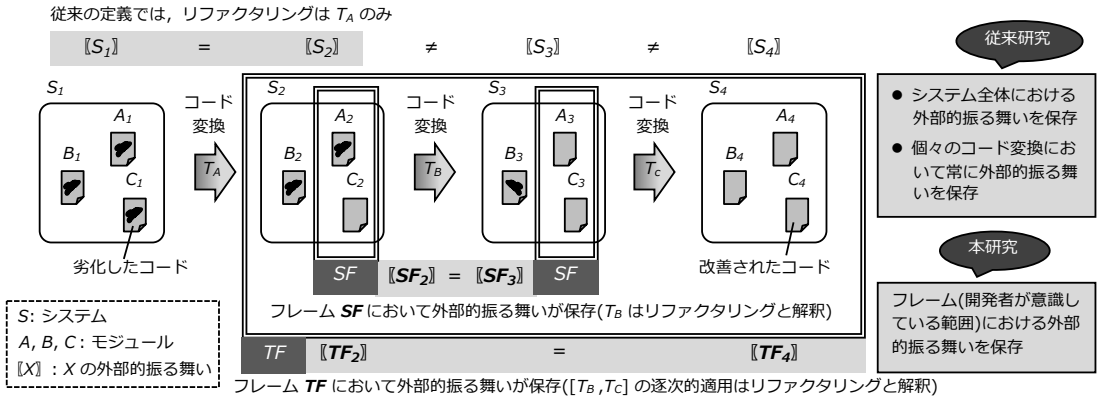

図 1: 本研究におけるアイデアと特徴

空間的フレームと時間的フレームについて概念を洗い出し、それらを一般化しただけでは、実際の開発および保守現場において自動リファクタリングの浸透は見込めない。そこで、これらのフレームを有効に活用したリファクタリング手法を提案し、プロトタイプツールとして実装する。

また、空間的フレームと時間的フレームの洗い出しにあたり、実際にどのようなリファクタリングが適用されているのかを調査することは必須である。その際、リファクタリング操作だけでなく、リファクタリング適用前後で行われたコードの編集操作を分析することになる。また、手動で適用された(と予測できる)リファクタリングが外部的振る舞いの保存を保証しているかどうかを確認する必要もある。このような観点から、既存のリファクタリング記録ツールの利用を検討したが、それは難しいことが分かった。そこで、本研究では、統合開発環境 Eclipse の Java エディタ上で実施されたリファクタリング操作を正確に収集するコード変化記録ツールも開発することとする。

４．研究成果

主に，以下の 3 つの研究成果を達成した。

(1) 空間的フレームの活用

一般的に、リファクタリング対象のモジュールにおけるすべての入出力に対して、外部的振る舞いが保存されていることを保証することは難しい。そのため、実際の開発現場では、十分なテストケースを用意し、テスト結果がリファクタリング前後で同一であれば、外部的振る舞いが保存されたと見なしている。本研究では、このような実践的な方法では、テスト結果が同一であるにもかかわらず、外部的振る舞いが変化する事例を示した。具体的には、テスト結果が同一であることを確認しながら、インラインメソッド(Inline Method)とメソッド抽出(Extract Method)リファクタリングを適用した場合に、外部的振る舞いが保存されたと見なせる場面と保存されていないと見なせる場面が混在することを示した。このように相反する場面が開発現場で混在すると、開発者が混乱する。たとえば、リファクタリングを適用した開発者が分散開発環境 Git のコミットメッセージに「リファクタリング」と記述すると、別の開発者は外部的振る舞いが保存されていることを期待する。しかしながら、コード変換により、外部的振る舞いの保存が保証されてないメソッドが出現している可能性があり、今後のソフトウェア改変においてバグが混入する可能性が高まる。

本研究では、このような現象が発生する原因が、リファクタリングにおけるコード変換の誤りや不完全さにあるのではなく、リファクタリング時におけるテストケースの損失やリファクタリングによって改変されたコードに対するテストケースの不完全さに起因することを明らかにした。その上で、テストケースの集合で空間的フレームを定義し、それを適切に管理することで、リファクタリングにおける外部的振る舞いの曖昧さを排除できる手法を提案した。開発者が空間的フレームを共有する仕組みを導入することで、特定のフレームに対して外部的振る舞いの保存を保証することが可能となり、外部的振る舞いに対する開発者間の解釈の違いによる混乱が避けられる。このような外部的振る舞いの保存に対する新たな概念を，リファクタリング研究コミュニティに持ち込んだという意味で本研究の意義は大きい。さらに、テストケースの不完全さを補うという観点から、メソッド抽出リファクタリングを適用した際、既存のテストコードから新規に抽出したテストコードを自動生成する仕組みを提案し、プロトタイプツールを完成させた。このようなツールの利用価値を広く開発および保守現場に示していくことが今後の課題である。

(2) 時間的フレームの活用

従来の自動リファクタリングツールにおける作業は大きく、前提条件の検査と実際のコード

変換に分けられる。たとえば、開発者がメソッド抽出リファクタリングを適用する場面を考える。まず，ツールは、指定したコード区間が構文として正しく切り出せるのかどうか、指定したコード区間で定義している変数の値を戻り値として定義できるのかどうかなどを、リファクタリングの適用に対する前提条件として検査する。これらの検査に通過した場合、抽出後のメソッドの名前の入力を促し､実際にコード変換を適用することで､メソッドの抽出を達成する。

ここで、従来のツールでは、前提条件の検査に失敗した場合、リファクタリングの適用作業のすべてが取り消される。外部的振る舞いの保存を保証するという立場からは、前提条件に違反するリファクタリングの適用を中止するという方針は自然である。その一方で、開発者から見ると、コード区間を指定する時点で前提条件を満たすかどうかを判断することは難しく、前提条件に違反したからといって一方的にリファクタリングが中止されるという方針は受け入れがたい。ツールによりリファクタリングが中止された場合、コード区間の指定などの作業を最初からやり直す必要があり、このような作業の繰り返しは開発の生産性を著しく低下させる。

そこで、本研究では、リファクタリングを中止するのではなく、その適用を延期するための仕組みを確立した。これを Postponable Refactoring と呼ぶ。従来の自動リファクタリングでは、外部的振る舞いの保存を保証するために、ツールによる適用の途中において、開発者による手動のコード編集ができるようになっていない。これに対して、Postponable Refactoring では、延期したリファクタリングの再開時点と延期されたリファクタリングの完了時点で時間的フレームを構成することで、実際には適用途中で実施されたコード編集をリファクタリングの適用前に実施されたと見なすことが可能である。つまり、実際の作業において自動リファクタリングの適用途中に手動のコード編集を許しているにもかかわらず、外部的振る舞いの保存は手動のコード編集に関与しないという立場を実現可能としている。これにより、開発者が意識する時間的フレームにおいて外部的振る舞いを保存したままで、その適用の柔軟性を向上させることに成功した。

Postponable Refactoring のプロトタイプツールの実装においては、統合開発環境 Eclipse に現在実装されているメソッド抽出リファクタリングに対して前提条件をすべて調査し、従来の前提条件違反(エラー)を延期により将来的に前提条件が満たされる可能性を持つ条件の違反(回復可能なエラー)とリファクタリングを最初からやり直さなければ前提条件を満たすことができない(回復不可能なエラー)に分類した。回復可能なエラーが延期対象である。

図 2 にプロトタイプツールの利用場面を示す。ツールは、リファクタリング適用時に前提条件を満たさないことを検知すると、ダイアログにより開発者に伝える。開発者は、ダイアログ内の「Postpone」ボタンを押すことでリファクタリング適用の延期を選択できる。延期が選択された場合、延期されたリファクタリングに関係するコード区間を特定し、それに対するコード編集の監視を開始する。開発者がコード変数を行うごとに、前提条件の再検査が実施され、延期されたリファクタリング操作の状態を推移させる。図 2 において、①は回復可能前提条件が満たされるのを待っている状態、②は前提情感が満たされたため再開を待っている状態、③は回復不可能なエラーが存在している状態を指す。②の状態のとき､開発者はメニューから「再開」を選択することができる（「中止」はどの状態でも実施可能である)。

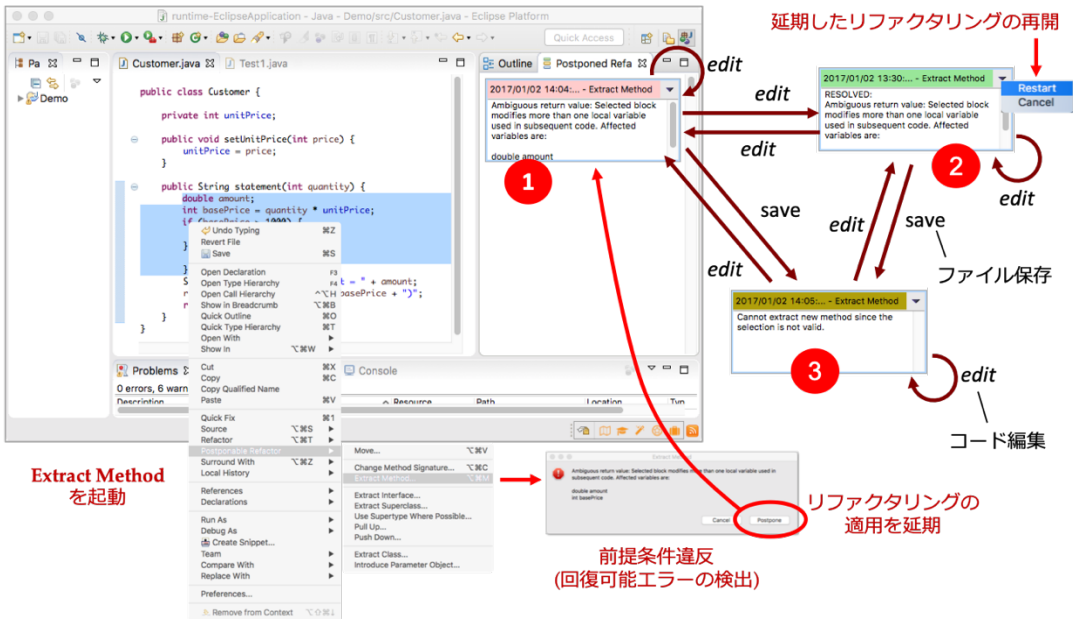

図 2: Postponable Refactoring ツールの利用場面

リファクタリングを延期させることで、開発者は最初からリファクタリング作業をやり直すことが避けられ、生産性の低下を防ぐことができる。このように、適用に関する障壁を取り除くことで、自動リファクタリングの適用を促進することができる。

(3) リファクタイングの分析環境

実際の開発および保守において、どのようなリファクタリングが適用されているのかを正確に分析するために、Eclipse のプロジェクト内部の Java ソースコードに適用された変更を記録する編集操作記録ツール ChangeMacroRecorder を開発した。

ここで、既存の編集操作記録ツールでは、開発者の編集や開発者の実施するコード変換アクションがソースファイルに閉じていることを前提にしており、複数のソースファイルに散らばったコード変化を正確に追跡することはできない。これに対して、多くのリファクタリング操作は同時に複数のソースファイルを更新する。このため、既存の編集操作記録ツールでは、リファクタリングによるコード変化を正確に追跡できない。

本研究で開発した ChangeMacroRecorder は、ソースファイルの内容を監視することで、リファクタリングが実施された際に影響を受けたソースファイルの変化を正確に追跡できる。さらに、適用したリファクタリング操作とそれによるコード変化を関連付けて記録する。これにより、リファクタリングによるコード変化を後で推測する必要がなくなる。これは、記録した編集操作を分析したり加工したりする作業の手間の削減につながる。

ChangeMacroRecorder は、リファクタリング操作だけでなくソースコード変更に関係するさまざまな編集操作を正確かつ理解しやすい形式で記録できるようになっている。このため、本研究のためのプロトタイプという位置付けにとどめず、ソースコード進化分析に携わる多くの研究者が利用可能なツールプラットフォームとして完成させた。

当初の研究計画に加えて、いくつかの発展的成果も得られた。フレームという観点からリファクタリングを見直すことで、リファクタリング適用の契機となるコードの臭い(code smell)を検出する手法の確立や、その除去における優先付け手法の確立という研究成果が得られた。また、編集操作記録ツールの実装に成功したことで、研究課題の広がりが達成できた。たとえば、コード編集履歴が、過去に適用されたリファクタリング操作の検出や分散協調開発環境におけるソースコードの競合解決に非常に有効であることが確認できた。

５．主な発表論文等

〔雑誌論文〕(計８件)

① 高橋 碧、 セーリム ナッタウット、 林 晋平、 佐伯 元司、情報検索に基づく Bug Localization への不吉な臭いの利用、情報処理学会論文誌、査読有、Vol.60、 No.4、pp.1040-1050、2019

② Shinpei Hayashi、 Fumiki Minami、 Motoshi Saeki、Detecting Architectural Violations Using Responsibility and Dependency Constraints of Components、IEICE Transactions on Information and Systems、査読有、Vol.E101-D、 No.7、pp.1780-1789、2018
DOI:10.1587/transinf.2017KBP0023

③ Natthawute Sae-Lim、 Shinpei Hayashi、 Motoshi Saeki、An Investigative Study on How Developers Filter and Prioritize Code Smells、IEICE Transactions on Information and Systems、査読有、Vol.E101-D、No.7、pp.1733-1742、2018
DOI:10.1587/transinf.2017KBP0006

④ Natthawute Sae-Lim、 Shinpei Hayashi、 Motoshi Saeki、Context-Based Approach to Prioritize Code Smells for Prefactoring、Journal of Software: Evolution and Process、査読有、Vol.30、No.6、pp.1-24、2018、DOI: 10.1002/smr.1886

⑤ Takayuki Omori、Katsuhisa Maruyama、Comparative Study between Two Approaches Using Edit Operations and Code Differences to Detect Past Refactorings、IEICE Transactions on Information and Systems、査読有、Vol.E101-D、pp.644-658、2018
DOI:10.1587/transinf.2017EDP7160

⑥ 西村 雄一、紙名 哲生、丸山 勝久、コードの編集履歴を用いた競合解決支援、情報処理学会論文誌、査読有、Vol.59、No.4、pp.1120-1136、2018

⑦ 林 晋平、 柳田 拓人、 佐伯 元司、 三村 秀典、クラス責務割当てのファジィ制約充足問題としての定式化、査読有、Vol.58、No.4、pp.795-806、2017

⑧ Katsuhisa Maruyama、Takayuki Omori、Shinpei Hayashi、Slicing Fine-Grained Code Change History、IEICE Transactions on Information and Systems、査読有、Vol.E99-D、No.3、pp.671-687、2016
DOI:10.1587/transinf.2015EDP7282

〔学会発表〕(計 33 件)

① Xiaoqian Xing、 Katsuhisa Maruyama、Automatic Software Merging Using Automated Program Repair、International Workshop on Intelligent Bug Fixing (IBF)、2019

② Takayuki Omori、 Katsuhisa Maruyama、 Atsushi Ohnishi、Summarizing Code Changes by Tracing an Operation History Graph、International Workshop on Mining and Analyzing Interaction Histories (MAINT'19)、2019

③ Aoi Takahashi、 Natthawute Sae-Lim、 Shinpei Hayashi、 Motoshi Saeki、A Preliminary

Study on Using Code Smells to Improve Bug Localization、International Conference on Program Comprehension (ICPC'18)、2018
④ Sarocha Sothornprapakorn、 Shinpei Hayashi、 Motoshi Saeki、Visualizing a Tangled Change for Supporting Its Decomposition and Commit Construction、Computer Software and Applications Conference (COMPSAC'18)、2018
⑤ Katsuhisa Maruyama 、 Shinpei Hayashi 、 Takayuki Omori 、ChangeMacroRecorder: Recording Fine-Grained Textual Changes of Source Code、International Conference on Software Analysis, Evolution, and Reengineering (SANER'18)、2018
⑥ 高橋 碧、 セーリム ナッタウット、 林 晋平、 佐伯 元司、情報検索に基づく Bug Localization への不吉な臭いの深刻度の利用、情報処理学会 ソフトウェア工学研究発表会、2018
⑦ 三宅皐、 紙名哲生、 丸山勝久、メソッド抽出リファクタリングにおけるテストケースの自動生成、情報処理学会 ソフトウェアエンジニアリングシンポジウム 2017、2017
⑧ Natthawute Sae-Lim、 Shinpei Hayashi、 Motoshi Saeki、How Do Developers Select and Prioritize Code Smells? A Preliminary Study、International Conference on Software Maintenance and Evolution (ICSME' 17)、2017
⑨ Katsuhisa Maruyama、 Shinpei Hayashi、A Tool Supporting Postponable Refactoring、International Conference on Software Engineering (ICSE'17)、2017
⑩ Katsuhisa Maruyama、 Shinpei Hayashi、 Norihiro Yoshida、 Eunjong Choi、Frame-Based Behavior Preservation in Refactoring、International Conference on Software Analysis, Evolution, and Reengineering (SANER' 17)、2017
⑪ Yuichi Nishimura、 Katsuhisa Maruyama、Supporting Merge Conflict Resolution by Using Fine-Grained Code Change History、23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER' 16)、2016
⑫ Natthawute Sae-Lim、 Shinpei Hayashi、 Motoshi Saeki、Context-Based Code Smells Prioritization for Prefactoring、International Conference on Program Comprehension (ICPC 2016)、2016
⑬ Jumpei Matsuda、 Shinpei Hayashi、 Motoshi Saeki、Hierarchical Categorization of Edit Operations for Separately Committing Large Refactoring Results、International Workshop on Principles of Software Evolution (IWPSE' 15)、2015
⑭ 丸山勝久、林晋平、吉田則裕、崔恩瀞、フレームベースリファクタリング～その概念と意義～、日本ソフトウェア科学会 ソフトウェア工学の基礎研究会 (FOSE2015)、2015

〔その他〕
ホームページ等
・MergeHelper: http://www.fse.cs.ritsumei.ac.jp/mergehelper/
・ChangeMacroRecorder: http://www.jtool.org/cmr/
・PostponableRefactoring: http://www.jtool.org/PostRefactor/

６．研究組織

(1)研究分担者
研究分担者氏名：林 晋平
ローマ字氏名： (HAYASHI SHINPEI)
所属研究機関名：東京工業大学
部局名：情報理工学院
職名：准教授
研究者番号（8桁）：40541975

(2)研究協力者
研究協力者氏名：大森 隆行
ローマ字氏名： (OMORI TAKAYUKI)