博士論文

ソフトウェア更新に追従する
コーディング規約改訂支援

上田 裕己

2022年3月14日

奈良先端科学技術大学院大学
先端科学技術研究科

本論文は奈良先端科学技術大学院大学先端科学技術研究科に
博士 (工学) 授与の要件として提出した博士論文である．

上田 裕己

審査委員：

松本 健一 教授	（主指導教員）
笠原 正治 教授	（副指導教員）
石尾 隆 准教授	（副指導教員）
Raula Gaikovina Kula 助教	（副指導教員）
伊原 彰紀 講師	（和歌山大学）

ソフトウェア更新に追従する コーディング規約改訂支援*

上田 裕己

内容梗概

本論文は，ソフトウェアの継続的な品質維持を目的として，ソフトウェアプロジェクトの変更に追従するよう，コーディング規約を最適化および抽出する手法を提案する．ソフトウェア開発において，ソースコードの理解に要する時間を削減する取り組みの一つにコーディング規約の利用がある．組織がコーディング規約を採用することで，ソースコードの可読性を向上させるだけでなく，将来的にバグを発生させやすい記述を予防する効果がある．

一方で，言語仕様の頻繁な変更やソフトウェアプロジェクトの多様性拡大により，自身が扱うプロジェクトに適合したコーディング規約を選定することは困難である．またコードレビューへの分析により，開発者は規約での自動検出が困難な修正作業を繰り返し行っていることを確認している．

本論文では，コーディング規約の誤用と規約不足を解決するために，コーディング規約の改訂を支援する以下の手法を提案する．

(1) 既存コーディング規約の最適化手法

開発者が実際に遵守しているコーディング規約に基づき，静的解析ツールで検証するコーディング規約を自動設定する手法を提案する．実験の結果，提案手法を用いることで，開発者が手動で設定した場合よりも静的解析ツールによる誤検出を削減できることを確認した．

(2) 新規コーディング規約の抽出手法

開発者が行ったソフトウェア変更を規約として抽出する手法を提案する．実験

*奈良先端科学技術大学院大学 情報科学研究科 博士論文, 2022 年 3 月 14 日.

の結果，提案手法を用いて自動生成した修正パッチのうち，80% (8/10) がソフトウェアプロジェクトに採用されたことを確認した．

以上，ソフトウェア開発において，ソースコードの一貫性を保持するための手法を提案し，実験的な評価を行った．本研究で得られた成果はソフトウェア開発において，開発者の方針を定型化し，ソフトウェア製品の品質向上に貢献すると考えられる．

キーワード

コーディング規約, 静的ソースコード解析, コードレビュー，ソースコードエディタ，正規表現

Revising Coding Convention to Mirror Software Updates*

Yuki Ueda

Abstract

This thesis proposes a method for optimizing and extracting coding conventions to follow the changes in a software project for continuous software maintenance. In software development, coding conventions are one way to reduce the time required to understand source code. The adoption of coding conventions by an organization not only improves the readability of source code but also prevents descriptions that are likely to cause bugs in the future.

On the other hand, frequent changes in language specifications and the increasing diversity of software projects make it difficult to select appropriate coding conventions. In addition, analysis of code reviews confirms that developers repeatedly perform modifications that are difficult to detect automatically with coding conventions.

The thesis proposes the following method to support the coding convention revising to solve the misuse and lack of conventions.

(1) Optimization of Existing Coding Conventions: This thesis proposes a method to automatically set coding conventions to be verified by a static analysis tool based on the coding conventions followed by developers. As a result of experiments, we confirmed that the proposed method reduces the number of false positives by the static analysis tool compared to the case where the developer manually sets the conventions.

*Doctoral Dissertation, Graduate School of Information Science, Nara Institute of Science and Technology, March 14, 2022.

(2) Extraction of New Coding Conventions: This thesis proposes a method for extracting software changes made by developers as conventions. As a result of the experiments, we confirmed that the software project adopted 80 % (8/10) of the patches automatically generated using the proposed method. The results obtained in this study will contribute to improving the quality of software products by formalizing the developer's policy in software development.

Keywords:

coding convention, static source code analysis, code review, source code editor, regular expression

関連発表論文

査読付学術論文

- 上田裕己, 石尾隆, 伊原彰紀, 松本健一. コードレビュー作業において頻繁に修正されるソースコード改善内容の分析. コンピュータ ソフトウェア, 37巻, 2号, pp. 2_76-2_85, 2020, 3章

- Yuki Ueda, Akinori Ihara, Takashi Ishio, Toshiki Hirao, Kenichi Matsumoto. How are IF-Conditional Statements Fixed Through Peer CodeReview?. IEICE TRANSACTIONS on Information and Systems, 101(11), pp. 2720-2729, 2018, 3章

査読付国際会議発表

- Yuki Ueda, Takashi Ishio, Kenichi Matsumoto. Automatically Customizing Static Analysis Tools to Coding Rules Really Followed by Developers. In Proc. The 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'21), pp. 541-545, 2021, 4章

- Yuki Ueda, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. Mining Source Code Improvement Patterns from Similar Code Review Works. In Proc. The 13th International Workshop on Software Clones (IWSC ' 19), pp. 13-19, 2019, 3章

- Yuki Ueda, Akinori Ihara, Takashi Ishio, and Kenichi Matsumoto. Impact of Coding Style Checker on Code Review -A case study on the OpenStack projects-. In Proc. The 9th International Workshop on Empirical Software Engineering in Practice (IWESEP ' 18), pp. 31-36, 2018, 3章

- Yuki Ueda, Akinori Ihara, Toshiki Hirao, Takashi Ishio, and Kenichi Matsumoto. How is IF Statement Fixed through Code Review? - a Case Study

of Qt Project -. In Proc. The 8th IEEE International Workshop on Program Debugging (IWPD ’17), pp. 207-213, 2017, 3 章

その他

- 上田 裕己 , 石尾 隆 , 松本 健一. 静的解析ツールの誤検出および検出漏れの最小化支援. 研究報告ソフトウェア工学（SE-206）, 2020-SE-206 No.1, pp. 1-8, 2020 年, 4 章
- 上田 裕己, 伊原 彰紀, 石尾 隆, 松本 健一. コードレビューにおけるソースコード修正理由の自動説明に向けて. ソフトウェアエンジニアリングシンポジウム 2018 (SES ’18) ワークショップ, 2018 年, 5 章
- 上田 裕己, 伊原 彰紀, 石尾 隆, 松本 健一. コードレビューを通じて行われるコーディングスタイル修正の分析. 第 25 回ソフトウェア工学の基礎ワークショップ (FOSE ’18), pp. 52-63 2018 年, 3 章
- 上田 裕己, 伊原 彰紀, 石尾 隆, 桂川 大輝, 松本 健一. ソーシャルコーディングにおけるソースコード中の If 文自動検証システムの開発. マルチメディア, 分散, 協調とモバイル (DICOMO ’18) シンポジウム, pp. 850-856, 2018 年
- 上田 裕己, 伊原 彰紀, 平尾 俊貴, 石尾 隆, 松本 健一. コーディング規約改定によるコードレビュー中の軽微な変更の分析. ソフトウェアエンジニアリングシンポジウム 2017 (SES ’17),pp. 220-223, 2017 年, 3 章
- 上田 裕己, 伊原 彰紀, 平尾 俊貴, 石尾 隆, 松本 健一. コードレビューを通して変更依頼される If 文の分析 (ポスター). ソフトウェアエンジニアリングシンポジウム 2017 (SES ’17), 2017 年,

目 次

1. 緒論 **1**

2. 関連研究 **4**

2.1 コーディング規約と静的解析ツール 4

2.2 ソースコード評価指標 5

2.3 コードレビュー 5

2.4 コード変更抽出技術 6

3. コードレビューを対象としたコーディング規約の利用状況調査 **7**

3.1 緒言 7

3.2 ケーススタディ 9

3.2.1 データセット 9

3.2.2 RQ1-1: コードレビューを通してコード改善はどの程度行われるか？ 12

3.2.3 RQ1-2: 静的解析ツールによる規約違反はコードレビューを通して修正されているか？ 17

3.3 妥当性への脅威 21

3.3.1 内的妥当性 21

3.3.2 外的妥当性 21

3.4 結言 22

4. 静的解析ツール出力内容に基づく既存コーディング規約の最適化 **23**

4.1 緒言 23

4.2 本提案手法の立ち位置 24

4.3 提案手法の利用方法 27

4.4 ソフトウェアフレームワーク 28

4.5 ケーススタディ 29

4.5.1 データセット 30

4.5.2 RQ2-1: どの規約にソフトウェアプロジェクトは実際に準拠しているか？ . . . 30
4.5.3 RQ2-2: プロジェクトが従う規約は導入時から最新まで一貫してるか？ . . . 33
4.5.4 RQ2-3: 提案手法により規約違反の誤検出および検出漏れを削減可能か？ . . . 36
4.6 妥当性への脅威 . . . 38
4.6.1 内的妥当性 . . . 38
4.6.2 外的妥当性 . . . 39
4.7 結言 . . . 39

5. コード編集履歴を用いた新規コーディング規約セットの抽出 40
5.1 緒言 . . . 40
5.2 本提案手法の立ち位置 . . . 41
5.3 提案手法の利用方法 . . . 42
5.3.1 事前準備 . . . 42
5.3.2 Visual Studio Code . . . 43
5.3.3 コマンドラインインターフェース . . . 44
5.3.4 GitHub Actions . . . 45
5.4 ソフトウェアフレームワーク . . . 46
5.4.1 規約ファイル要素 . . . 46
5.4.2 正規表現規約の自動生成 . . . 50
5.5 ケーススタディ . . . 51
5.5.1 データセット . . . 51
5.5.2 RQ3-1 および RQ3-2 評価手法 . . . 53
5.5.3 RQ3-1: 抽出した規約セットにより既存のコード修正ツールよりも多くの不具合を修正可能になるか？ . . . 54
5.5.4 RQ3-2: どのような不具合に対して抽出した規約は効果的か？ . . . 56

5.5.5 RQ3-3: 抽出した規約により開発者に採用される修正提案が可能になるか？ 59
5.6 妥当性への脅威 61
5.6.1 内部妥当性 61
5.6.2 外部妥当性 62
5.7 結言 62

6. 結論 **64**
6.1 課題と展望 64

謝辞 **66**

参考文献 **67**

図 目 次

1 各調査質問に対応する分析対象 8
2 変更チャンクと変更ラベルの抽出方法 12
3 RQ1-1: コードレビューを通して実施されたコード改善とバグ修正の出現回数 (総変更数: 384 件，うちコード改善は 211 件) 16
4 本提案手法のインストール及び実行ワークフロー 28
5 RQ2-3: 規約違反数のカウント方法 37
6 Visual Studio Code への DevReplay 拡張機能インストール手順 45
7 Visual Studio Code 上でのソースコード修正および規約セット自動生成 46
8 プルリクエストにおける GitHub action 出力例 49
9 正規表現コーディング規約セット生成プロセス 51

表 目 次

1 対象プロジェクト概要 11
2 コード改善とみなす変更ラベル 14
3 RQ1-2: コードレビューを通して頻繁に静的解析ツール Pylint で検出される警告 (出現回数順) 18
4 RQ1-2: コードレビューを通して頻繁に静的解析ツール flake8 で検出される警告 (出現回数順) 19
5 RQ2: 評価対象プロジェクトのリリースバージョン数 31
6 RQ2-1: 対象プロジェクトで共通して準拠されている規約分布 . . 32
7 RQ2-2: 各リリースごとに提案手法を適用した規約および，プロジェクトが設定した規約の精度，網羅率，F 値 35
8 RQ2-3: 提案手法によって検出した ASAT 違反行数の予測 Precision, Recall, F 値 (n=405,094,892) 38
9 各ツールの機能比較 41
10 規約重要度と DevReplay 機能の対応 50
11 RQ3-3: 対象プロジェクト及び規約セット生成対象期間 53
12 RQ3-1: Codeflaw における既存ツールおよび提案手法の不具合修正精度および件数 55
13 RQ3-1: IntroClass におけるブラックテスト不具合への修正精度 . 56
14 RQ3-1: IntroClass におけるホワイトテスト不具合への修正精度 . 57
15 RQ3-2: Codeflaw における不具合種類ごとの修正件数 (n=651) . . 58
16 RQ3-2: IntroClass におけるブラックテスト不具合の修正精度 . . 58
17 RQ3-2: IntroClass におけるホワイトテスト不具合の修正精度 . . 59
18 RQ3-2: IntroClass における有効な不具合修正を行った規約セットとその頻度 60

Listings

1 no-template-curly-in-string 規約違反と違反修正例 25

2 no-empty 規約違反と違反修正例 26

3 RQ2-1: JavaScript における *no-unreachable* and *no-constant-condition* 規約違反例 33

4 修正対象ソースコード例 'target.py' 43

5 DevReplay 規約セットファイル例 ".devreplay.json" 43

6 Listings 5 を元に自動生成された ".devreplay.json". 規約要素型であった Listings 5 から規約リスト型に変更している 47

7 GitHub Action ファイル ".github/workflows/devreplay.yml" . . . 48

8 Codeflaws データセットにおけるコード修正例 52

9 RQ3-2:IntroClass における Leave One Out 手法で利用したソースコード修正例 61

10 RQ3-2: IntroClass における 5 program 手法で利用したソースコード修正例 62

11 RQ3-3: オープンソースプロジェクト開発者に採用されたソースコード変更内容 63

1. 緒論

コーディング規約（プログラミング作法，コーディングルール，プログラミングスタイルとも呼ぶ）はソフトウェア開発において，ソースコードの可読性向上に加え，メンテナンス性の確保を行う上で広く活用されていると報告されている [1, 2, 3, 4].

コーディング規約は，開発者へのドキュメントとしてだけでなく，ソフトウェア開発プロセスの一部としても利用されている．代表的な例として，静的解析ツールはソースコード編集中およびソースコード更新時に，コーディング規約に違反したソースコードを自動検出し，開発者に通知する [5]．静的解析ツールを用いることで，開発者はソフトウェア実行前に不具合を予防および削減することが可能になる．

静的解析ツールが検出するコーディング規約違反の中には，ソフトウェアプロジェクトにとって重要でないものも存在する．プロジェクトは，静的解析ツールの偽陽性を防止および削減するために，有効とするコーディング規約を選定する．この規約集合を，本論文では**規約セット**と呼ぶ．以降，本論文では，静的解析ツールで検出可能かつ，有効化および無効化が可能なコーディング規約を分析対象とする．

一方で，ソースコードの記述内容をプロジェクトの方針とすると，コーディング規約が常に有効に働くとは限らない．大きく２つの原因が報告されている．(1) **コーディング規約セットの形骸化**: コーディング規約違反の警告のうち，80%以上は開発者に修正されないことが報告されている [6, 7, 8]．そのため，本来有用な警告も無視されてしまうという点で，規約セット及びツールが形骸化する原因となる．また，多くの開発プロジェクトは静的解析ツールで用いるコーディング規約の有効化および無効化設定をそのプロジェクトの内容に関わらず，変更しないことが報告されている [9]．そのため，プロジェクトの方針に沿わない規約が運用され続けことになる．(2) **暗黙的なコーディング規約セットの運用**: ソフトウェアプロジェクトでは，類似した修正を一ヶ月以内に繰り返すことが報告されている [10]．こういった限定的なソースコード修正内容は，実装コストや有効期間およびプロジェクトの狭さから，既存の静的解析ツールで検出対象とならない．例

えば，開発者が行うソースコード変更内容の半数以上は類似しているものの，既存の静的解析ツールで運用されているコーディング規約セットには含まれていない (3 章)．そのため，既存のコーディング規約セットを用いるだけではプロジェクトの方針を十分に反映することができない．

そこで，本研究は，プロジェクトの方針とコーディング規約セットを一致させるために，コーディング規約の改訂を支援する以下の 3 点について研究を行った．

(a) 既存コーディング規約の利用状況調査 (3 章)
開発者がソースコードを検証するコードレビューにおいて，ソースコード中の欠陥の発見はもちろん，コーディング規約セットにも含まれるような空白の追加や，変数名の変更など可読性の改善を目的とした指摘も多数報告されている [11, 3]．しかしながら，従来研究では具体的な修正内容については議論されておらず，開発者が注意すべきコーディング規約違反は膨大なままである．本論文では，開発者が行うソースコード修正内容を定量的に分類し，コーディング規約セットとの差異を評価する．

(b) 静的解析ツール出力内容に基づく既存コーディング規約セットの選定手法 (4 章)
本論文では，プロジェクトにおけるコーディング規約の過不足削減を目標とする．具体的には，ソフトウェアプロジェクトのソースコードに合わせてコーディング規約セットを推薦するシステムを提案する．また，実験で本提案手法を用いることで，開発者が手動で作成した規約セットよりもプロジェクトにおける規約の過不足を削減可能であることを確認した．

(c) コード編集履歴を用いた新規コーディング規約の抽出手法 (5 章)
本論文では，既存のコーディング規約セットでは網羅しきれない，開発者やプロジェクト固有の暗黙的なコーディング規約を抽出し，繰り返し行われる修正作業を規約化および自動化するシステムを提案する．また，実験で実際のソフトウェアプロジェクトにおいて，本提案手法を用いて生成したソースコード変更提案のうち 80%(10 件中 8 件) が採用されたことを確認した．

以下，第 2 章では本論文の基礎となるコーディング規約に関する既存研究の説明を行う．次に，第 3 章で分析としてコードレビューにおけるコーディング規約の利用状況を調査する．また，開発者が実際に修正したソースコードの内容を自動分類する手法を提案する．更に第 4 章静的解析ツールを利用して，コーディング規約セットを選定する手法を提案する．実験により，手法により自動生成した規約の精度を評価する．第 5 章でソースコード変更履歴を用いて正規表現形式でコーディング規約を抽出する手法を提案し，ソフトウェアプロジェクトを対象とした実験によってその効果を評価する．最後に第 6 章で結論と，今後の展望を述べる．

2. 関連研究

2.1 コーディング規約と静的解析ツール

本論文ではコーディング規約を用いて，プロジェクトの一貫性を評価したり，ソースコードの修正方法を提案する．

通常コーディング規約は PEP8 [12]，CERT C [13] のようなプログラミング言語ごとに定義された規約を基に作成される．本論文では，規約の評価を定量化するためにコーディング規約は静的解析ツールによる違反を自動検出可能な規約にのみ注目する．

コーディング規約違反を自動検出するために，プログラミング言語に応じて，多くの静的解析ツール (Automated Static Analysis Tool，以降，**ASAT**) が開発されている．Java の場合は CheckStyle [14]，Python の場合は Pylint [15]，C ＃の場合は StyleCop [16]，Ruby の場合は RuboCop [17] がある．本論文が主に対象とする JavaScript 言語の場合，ESLint [18] が最も広く使用されている ASAT である．ESLint は，Google JavaScript スタイルガイド [19] や StandardJS [20] などの事前定義された主要なコーディング規約をサポートしている． また，開発者は自分の好みに一致するように特定の規約セットを構成できる．これらの ASAT はエディタ (Visual Studio Code，Atom) や統合開発環境 (IntelliJ IDEA, Eclipse) で自動実行することができ，Travis CI [21] や CircleCI [22] などの継続的インテグレーション環境でも利用されている．

ASAT の利用を容易にするために，複数の提案が行われている．UAV は開発者に ASAT 警告の理解を促進するために，Java の ASAT 警告を可視化するツールである [23]．また，C-3PR は ASAT による警告を GitHub 上で自動的に修正する機能を提供している [24]．

ASAT は広く利用されている一方で，検出された警告の 80%以上が開発者に修正されないといったように，誤検出の多さも指摘されている [6, 7, 8]．原因の一つとして，ASAT を利用するプロジェクトの多くが ASAT 導入後にコーディング規約を編集しないこと，つまりそのプロジェクトの開発者が興味を持たないコーディング規約違反も余分に検出していることが挙げられている [25]．本論文 4 章

では，誤検出の削減を目指してコーディング規約の最適化手法を提案する．

2.2 ソースコード評価指標

コーディング規約以外にも，ソースコードの可読性を評価するための様々な指標が提案されている．代表的なものとして，ネストの深さ [26] や複雑度 [27] が頻繁に利用されてきた．近年は，ソースコードの自然さ，つまりプロジェジェクト全体におけるソースコード記述の一貫性が主に機械学習を用いた推薦手法に利用されている [28]．NATURALIZE は統計的自然言語処理技術を用いて，一貫した識別子の命名規則や空白などのフォーマットを提案する [29]．以上の指標に違反したソースコードは，静的解析ツールによる検出が行えないため，本論文では分析対象としない．

2.3 コードレビュー

3 章では，コーディング規約の利用状況調査において，コードレビューを対象とする．コードレビューは静的解析ツールやコーディング規約が最も重用される開発プロセスであり，コーディング規約に基づいてリファクタリングを含むソースコード改善を行っている [11, 30, 31]．特に，レビューで行われている議論の 75% はソフトウェアの保守性に注目し，機能的な問題に関する議論は 15%である [1, 32]．オープンソースソフトウェア開発では， Gerrit [33] や Review Board [34] をはじめとしたコードレビュー管理システムを利用することで，開発者が投稿したパッチについて検証者と開発者間で円滑なコミュニケーションを行う [35]．また，パッチ開発者と検証者は互いにソースコードを改善するための議論を行う [36]．コードレビューはソフトウェアの信頼性確保のために効果的だが，多くの時間的コストを消費することが確認されている [37]．Panichella らはパッチ投稿前に静的解析ツールを導入することが，コードレビューで行われた修正作業の削減に貢献すると主張した [8]．本論文 3 章では，開発者がコードレビューで実際に行っているコーディング規約に関する修正の頻度や内容を定量的に分析する．

2.4 コード変更抽出技術

静的解析ツールの機能やコード修正ツールを改善するために，変更抽出手法が提案されている．変更抽出手法の利用目的として，ソースコード自動修正技術が提案されている [38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51]．例えば，PAR [52] や FixMiner [46]，SysEdMiner [53] は本論文 5 章と同様に Git リポジトリからコード変更パターンを検出し，開発者に修正を提案する．既存研究により抽象構文木形式で出力する変更パターンは，静的解析ツールのコーディング規約と異なり出力後の編集が困難となる．5 章では，開発者が編集可能な形式である正規表現での変更抽出手法を提案することで，将来の誤検出および検出漏れへの対応を可能にする．

また近年は，静的解析ツールの違反を自動修正するパターンの収集も行われている [54, 10, 55]．これらの静的解析ツールを用いた手法は，ツールの性能および に依存する．本論文 4 章では，静的解析ツールの設定を自動設定し，ツール偽陽性および偽陰性を削減することで，これらの研究の精度に貢献する．

3. コードレビューを対象としたコーディング規約の利用状況調査

3.1 緒言

コードレビューは，ソフトウェア開発プロセスの 1 つであり，パッチ開発者が機能追加や欠陥修正のためにソフトウェアプロジェクトに投稿したソースコード変更提案 (以降，**パッチ**) を，複数の開発者 (以降，**検証者**) が検証する作業である．コードレビューはソフトウェアの信頼性を確保するための重要な活動であり，Czerwonka らは，検証者がコードレビューで行う指摘のうち，15 %はパッチの機能的問題に関する内容であることを報告している [1].

オープンソースソフトウェア開発においては，多数のパッチ開発者がそれぞれ独自に記述したソースコードを投稿する．そのため，ソフトウェアプロジェクトのソースコード記述を一貫した可読性の高いものにするためにソースコードの改善が重要となる．Rigby らはコードレビューにおいてスタイルの問題に関する指摘が行われていることを報告しており [2]，Bacchelli らは開発者へのインタビューを通じて，可読性の改善などのソースコードの品質向上がコードレビューの大きな目的の 1 つであると確認している [3].

コードレビューは重要な作業であると同時に，検証者にとっては負荷の高い作業でもある．1 件のコードレビューに約 1 日，長い場合は数週間を必要とし，1 人の検証者が平均 6 時間 / 週を費やす [56, 35]．時間を費やす原因は，一度の検証ではパッチに含まれるすべての問題を発見または改善することが困難であり，繰り返しの検証が必要なためである [57].

コードレビューではソースコード中の欠陥の発見はもちろん，空白の追加や，変数名の変更など動作に大きな影響をもたない可読性の改善を目的とした指摘も多数報告されている [11]．本章は，コードレビューにおいて指摘されることが多い可読性の改善 (以降，**コード改善**) 内容を明らかにする．それにより，コードレビューにおいて，コード改善点発見に対する静的解析ツール (Automated Static Analysis Tool，以降，**ASAT**) の有用性と，将来の ASAT に求められる機能の発見を目指す.

パッチとして投稿されるソースコードの品質を向上させるために，プロジェクトはコーディング規約を定めている．一部のコーディング規約違反は，ASAT によって自動検出することが可能であり，パッチ開発者はツールを用いてパッチ投稿前にソースコードを検証し，プロジェクト内のソースコード記述方式に一貫性をもたせることができる．しかし，コードレビューにおいて実施されるソースコード改善のすべてがコーディング規約に含まれているわけではなく，検証者がコーディング規約以外に何を確認しているか，また検証者がどの程度コーディング規約違反を指摘しているかは明らかとなっていない．本章では，コードレビューを通して行われるソースコード改善と，ASAT が検出できる規約違反を明らかにする．図 1 に調査質問に対応する分析対象を示す．RQ1-1 で対象とするレビューを通した指摘と RQ1-2 で対象とする ASAT による警告内容の一部は重複するため，両方で各分析対象に基づいた議論を行う．具体的には，以下の 2 つの調査質問に対する分析を行う．

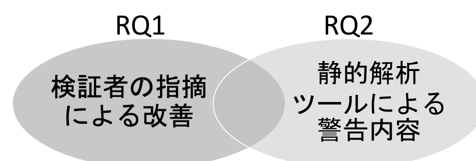

図 1: 各調査質問に対応する分析対象

- **RQ1-1: コードレビューを通してコード改善はどの程度行われるか？**パッチ投稿後にコードレビューを通して変更されたソースコードの変更内容を目視で確認し，検証者によって指摘されやすいソースコードの内容を明らかにする．また，各コード改善について，コードレビュー中の出現回数を比較することで，パッチ開発者がパッチ投稿前に特に注意すべきコードの問題と，そのうち ASAT の導入で検出が可能である問題を明らかにする．

- **RQ1-2: ASAT による警告はコードレビューを通して修正されているか？** パッチ投稿直後のソースコードに ASAT を実行し，コードレビューを通して修正されやすいコーディング規約違反を明らかにする．また，コードレ

ビュー前後での規約違反の数を比較することで，ASAT として実装されたコーディング規約がどの程度検証者の指摘方針を反映しているか否かを明らかにする．

調査対象として，コードレビュー管理システムである Gerrit 上で OpenStack が公開している Python プロジェクト群のコードレビューデータセット [58] と，GitHub 上で Microsoft，Google，Facebook が公開している Python プロジェクト群から収集したレビューデータセットを用いる．

3.2 ケーススタディ

本章では，コードレビュー管理システムおよび GitHub から収集したパッチに対して，コードレビューによって行われた変更を分析する．まず，パッチ開発者が作成し，プロジェクトに投稿したパッチを初版 $Patch_1$ とする．また，検証者による指摘とパッチ開発者による修正を繰り返してプロジェクトに採用されたパッチを最終版 $Patch_n$ とする．コードレビューによる変更内容を分析するため，初版 $Patch_1$ から修正されないままプロジェクトに採用されたパッチは分析対象から除外する．

コードレビューによる変更内容は多岐にわたるため，RQ1-1 では，パッチが対象プロジェクトに追加するソースコードのチャンク（断片）のうち，$Patch_1$ と $Patch_n$ の間で変更されたもの（以降，**変更チャンク**）の内容を目視で分析する．変更チャンクは `diff` コマンドで出力したソースコードの差分から得られる追加行と削除行によって構成されている．このとき，文字の置き換えなどの変更は追加と削除の組み合わせによって表現する．RQ1-2 については，$Patch_1$ を適用した状態のソースコードと $Patch_n$ を適用した状態のソースコードに対して ASAT を実行し，コーディング規約違反に関する調査を行う．

3.2.1 データセット

データセットとして，OpenStack，Google，Microsoft，Facebook が公開しているプロジェクト群を用いる．対象とするプロジェクトは，コードレビューの管

理に Gerrit Code Review や GitHub を用いて膨大なレビュー活動を公開している．本論文ではこれらのプロジェクトに投稿されたパッチから抽出した変更チャンクを分析対象とする．

また， 対象プロジェクトは Python 言語標準のコーディングスタイル PEP8 [59] に基づいた ASAT である Pylint を利用することによって，自動的に PEP8 に違反するソースコードを検出している．

Yang らが作成した Gerrit データセットに記録された OpenStack のコードレビューデータ [58] と Gerrit REST API [60] を利用し，OpenStack が Gerrit Code Review 上で公開しているプロジェクトから，パッチとそれに含まれる変更チャンクを収集する．また，GitHub API [61] を利用し，Google，Microsoft，Facebook が GitHub 上で公開しているプロジェクトからパッチとそれに含まれる変更チャンクを収集する．

表 1 に対象プロジェクトの概要を示す．RQ1-1 では各変更チャンクを目視で調査するため，変更チャンク数の偏りがでないようにすべてのプロジェクトの対象チャンクから層別サンプリングを行う．このとき，許容誤差は 5%，信頼度は 95% とする．RQ1-2 でも，一般性を保証するため，パッチ数の異なる Gerrit と GitHub のプロジェクトから 382 件ずつパッチをサンプリングする．

表 1: 対象プロジェクト概要

プロジェクト名	観測期間	パッチ数	変更チャンク数	サンプルパッチ数	サンプルチャンク数
OpenStack	2011 – 2015	68,174	61,673	382	144
Google	– 2018	5,780	74,454	238	174
Microsoft	– 2018	2,827	25,795	116	60
Facebook	– 2018	658	3,345	28	5
合計	—	76,459	165,257	764	384

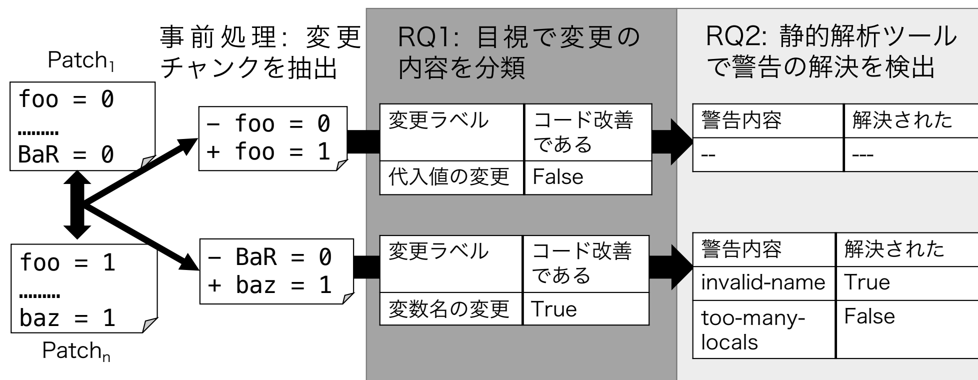

図 2: 変更チャンクと変更ラベルの抽出方法

3.2.2 RQ1-1: コードレビューを通してコード改善はどの程度行われるか？

本章では，コードレビューで頻繁に行われているコード改善の内容を明らかにする．コードレビュー前後のソースコード差分から $Patch_1$ と $Patch_n$ 間の変更チャンクを 384 件取り出す．また，変更内容を既存研究 [62] が定義したバグ修正パターン（Bug fix patterns）に基づきラベル付けを行い，コード改善であるか否かを目視で検出する．変更チャンクの抽出手順は以下の通りである．手順の例を図 2 に示す．

1. 事前処理として，$Patch_1$，$Patch_n$ のペアに `diff` コマンドを実行し，パッチ間の差分を得る．各パッチの内部に対応するチャンクが存在しており，かつ差分が存在しているものを変更チャンクとして抽出する．図 2 の例では，$Patch_1$ と $Patch_n$ の間で “`foo = 0`” が “`foo = 1`” に変更され，“`BaR = 0`” が “`baz = 1`” に変更されたという差分情報から，これらを 2 つの変更チャンクとして抽出している．

2. 各チャンクについて，目視で変更内容を確認し，ラベル付けをする．まず，チャンクが 1 種類のバグ修正パターンに該当する変更の場合は，その変更を機能的な変更とし，該当するバグ修正パターンをラベル付けする．チャンクが複数種類のバグ修正パターンを含む変更の場合には，その変更を大規模で機能的な変更とし，ラベル付を行わない．チャンクがバグ修正パターン

に当てはまらない場合は，その変更をコード改善と分類し，表 2 のラベルから 1 つ割り当てる．表 2 のコード改善ラベルでは，ASAT によって検出可能な改善とそうでない改善を分類しており，“ADD_REMOVE_SPACE”, “ADD_REMOVE_NEWLINE”,
“CHANGE_VALUE_STYLE” の 3 つが Pylint, Flake8 の規約違反として検出可能な改善のラベルである．

表 2: コード改善とみなす変更ラベル

改善名	コード改善内容
**ADD_REMOVE_NEWLINE	改行の追加削除
**ADD_REMOVE_SPACE	空白やインデントの追加削除
**CHANGE_VALUE_STYLE	変数名の大文字小文字，またはハイフン記号の変更
ABSTRACT_VALUE	変数の抽象化
ADD_REMOVE_ARRAY_ELEMENTS	配列要素の追加削除
ADD_REMOVE_DECORATOR	デコレータの追加削除
ADD_REMOVE_DICT_EELEMENTS	辞書要素の追加削除
ADD_REMOVE_RETURN	return 文の追加削除
CHANGE_COMENT	コメント文の変更
CHANGE_DEFINE_ORDER	変数を定義する順序の変更
CHANGE_FUNC_OBJECT	同一名の関数を呼び出すオブジェクトの変更
CHANGE_IMPORT	import 文で呼び出すライブラリの変更
CHANGE_STRING	文字列の変更
CHANGE_VALUE_NAME	“CHANGE_VALUE_STYLE” 以外の変数名の変更
UPDATE_VERSION	ライブラリのバージョン更新

** Pylint または Flake8 で自動検出可能な改善

図 3 に目視での分析によって分類し，2 回以上出現した各変更内容の出現回数を示す．ここで， 384 件の変更のうち，ラベル付を行うことのできなかった大規模で機能的な変更 6 件 (1.6%) は省略している．

コードレビューで行われた変更のうち，バグ修正パターンに当てはまらないコード改善は 384 件中 211 件 (54.9%) 件行われていた．この結果は，開発者がコード改善に注目していると回答した既存研究のインタビュー結果と合致する [3]．そのうち最も多く検出した改善内容は “CHANGE_VALUE_NAME”(変数名の変更) である．また，“ADD_REMOVE_SPACE” や “CHANGE_VALUE_STYLE” のように既存の ASAT で自動的に検出が可能であるにもかかわらずコードレビューで行われた変更は合計 384 件中 46 件 (11.5%) であった．

“CHANGE_STRING” は，自動的に検出できない典型例である．単純な例では小文字から大文字に変更するものや出力文字列の末尾にカンマを追加するといったものがある．また，複雑な例では，固有名詞や記法の修正，依存するファイルパスやライブラリのバージョン番号の変更といったプロジェクトの知識やファイル間の依存関係などを把握しないと変更できないものまであった．変数名や文字列の推薦を行う技術 [29] などを ASAT としてプロジェクトが採用することでより多くのコーディング規約違反を検出可能になる．

RQ1-1 に対する答えは以下の通りである．

コードレビューを通してコード改善はどの程度行われるか：目視で 384 件のソースコード変更履歴を分類した結果，コードレビューで行われるソースコードの変更内容 384 件中 211 件 (56.0%) がコード改善であることを確認した．特に多く出現した変更は “CHANGE_VALUE_NAME” である．

コード改善のうち “CHANGE_VALUE_STYLE” や “ADD_REMOVE_SPACE” のようにパッチ投稿前に既存の ASAT を利用することでパッチ投稿前に改善が可能な変更は 11.5% である．また，ASAT で検出可能な改善内容だけでなく，“CHANGE_VALUE_NAME”(43 件) や “CHANGE_STRING”(34 件) のように，自動検出が困難な改善が 384 件中 82 件 (20%) 行われた．既存の ASAT では検出が困難な，プロジェクトの固有名詞や出力文字列の記法に関する修正，各プロジェクトがコーディングガイドラインを定義し，開発者間で共有することでこれらの

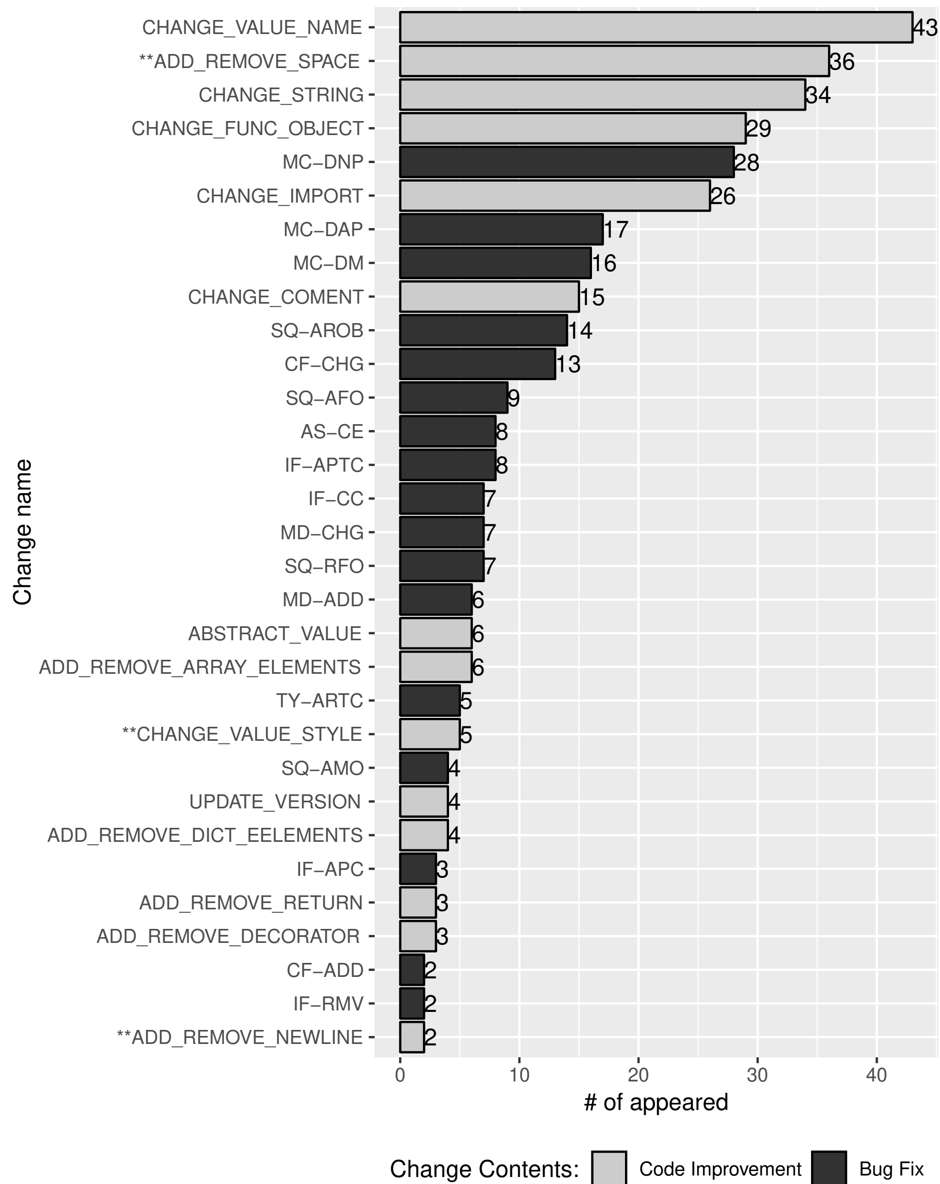

図 3: RQ1-1: コードレビューを通して実施されたコード改善とバグ修正の出現回数
(総変更数: 384 件，うちコード改善は 211 件)

問題は予防できる．また，将来 ASAT の機能を拡張，改善する際には表 2 で挙げた改善の自動検出が期待される．

3.2.3 RQ1-2: 静的解析ツールによる規約違反はコードレビューを通して修正されているか？

RQ1-1 で分類したコードレビューで行われるコード改善のうち，11.5% は ASAT の実行により，パッチ投稿前に改善可能であることを確認した．本節では，ASAT である Pylint と Flake8 を利用し，コードレビューを通して修正が行われたか否かに関わらず自動的な検出が可能なコーディング規約の違反を分析する．ASAT の警告の中にはプロジェクトの複数のファイルから影響を受けるものもあり，プロジェクトの規模による警告数の偏りがでないように，パッチ数の異なる Gerrit と GitHub のプロジェクトから 382 件ずつパッチをサンプリングする．次に，$Patch_1$ から検出可能な規約違反と修正された規約違反の出現回数をそれぞれ求める．ASAT が $Patch_1$ の状態のソースコードからコーディング規約違反を検出し，かつ $Patch_n$ のソースコードからコーディング規約違反を検出しなかったとき，コードレビューがその規約違反を修正したと判断する．$Patch_1$ と $Patch_n$ の両方の変更チャンク内から規約違反を検出した場合は，ASAT は検出可能であったが，コードレビューはその規約違反を修正しなかったと判断する．このとき，コード改善に注目するため，プログラミングエラーとして分類されている警告は無視する．図 2 の例では，$Patch_1$ の変数 `BaR` を含む行から変数名の命名規約違反である “invalid-name” が検出され，$Patch_n$ の変数 `baz` を含む行からは違反が検出されないため，コードレビューによる修正として扱う．

表 3 に，パッチ投稿時に最も多く検出した，10 種類の警告とその出現回数を ASAT ごとに示す．警告の出現回数が総パッチ数よりも多いのは，1 つのパッチが変更するソースコードに複数の警告が含まれる場合があるためである．対象としたパッチの 382 件中 51 件 (13.4%) は，パッチ投稿前に ASAT で検出が可能な修正である．パッチ開発者がパッチ投稿前に ASAT を利用することで，検証者のコストを下げることが可能になる．

表 3: RQ1-2: コードレビューを通して頻繁に静的解析ツール Pylint で検出される警告 (出現回数順)

Pylint 警告名	Pylint 警告内容	Gerrit 修正数/出現回数		GitHub 修正数/出現回数	
missing-docstring	関数やクラスにコメント文がない	10.4%	14 / 135	20.2%	41 / 203
invalid-name	命名規則違反	9.8%	10 / 102	17.4%	38 / 218
bad-continuation	可読性を下げる改行	16.4%	10 / 61	24.6%	35 / 142
no-self-use	クラス変数を利用しない関数	8.3%	5 / 60	13.0%	16 / 123
too-few-public-methods	public メソッドが少ないクラス	4.0%	2 / 50	13.0%	22 / 116
wrong-import-order	モジュールを読み込みがモジュール名のアルファベット順でない	2.6%	1 / 39	18.5%	23 / 124
unused-argument	使用されない引数の定義	2.0%	1 / 51	22.5%	23 / 102
fixme	解決されていない FIX ME コメント	5.9%	2 / 34	22.0%	24 / 109
too-many-arguments	1 関数に 10 以上の引数	13.2%	5 / 38	7.6%	8 / 105
too-many-locals	1 関数に 15 以上の変数定義	7.1%	2 / 28	11.8%	13 / 110

表 4: RQ1-2: コードレビューを通して頻繁に静的解析ツール flake8 で検出される警告 (出現回数順)

Flake8 警告名	Flake8 警告内容	Gerrit 修正数/出現回数		GitHub 修正数/出現回数	
E501	一行あたりに 80 文字以上記述されている	0.7%	1 / 147	50.5%	569 / 1,126
E114	コメント文のインデントが 4 文字の空白以外で構成されている	0.0%	0 / 134	44.0%	477 / 1,085
H405	複数行文字列中の不要な改行	11.1%	27 / 243	40.9%	375 / 917
H306	可読性を下げるモジュール読み込み順	2.0%	1 / 51	52.2%	554 / 1,061
H301	一行に 2 つ以上のモジュールの読み込み	0.6%	1 / 147	39.8%	303 / 762
H404	複数行文字列の行頭の改行	6.0%	16 / 270	31.0%	192 / 619
F821	定義されていないオブジェクトの利用	0.6%	1 / 167	27.5%	184 / 670
E302	関数定義の間に空行がない	0.0%	0 / 138	27.5%	167 / 646
F405	*を利用したモジュールの読み込み	0.0%	0 / 311	11.4%	54 / 473
F401	利用されないモジュールの読み込み	0.0%	0 / 96	28.4%	177 / 623

Pylint は合計 1,950 件の警告を検出しており，Flake8 は合計 11,015 件の警告を検出している．2 つのツール間での警告検出数に差が生じた原因を調査するため，目視でプロジェクトのリポジトリを確認した結果，Flake8 よりも Pylint を採用しているプロジェクトが多く，パッチ開発者が Flake8 の警告を見る機会がないことが考えられる．例えば，最も多くのパッチをもつ Google は，自身のコーディングスタイルガイドライン [63] で Pylint の利用を推奨しているが，Flake8 については言及していない．

GitHub で管理されたプロジェクトにおいて，警告の修正率は 7.6%から 52.2%であり，ツールが警告する規約違反の多くはコードレビューでは検証者に指摘されないまま，プロジェクトに統合されている．同様に，Gerrit で管理された OpenStack プロジェクトでは，ASAT による警告のうち修正される件数は多くとも “bad-continuation” の 61 件中 10 件 (16.4%) である．特に Flake8 で検出した警告のいくつかはコードレビューを通して全く修正されない．Pylint によって最も多く検出した警告 “invalid-name” （変数や関数の命名規則違反）は表 2 中の “CHANGE_VALUE_STYLE” の改善に対応するが，GitHub プロジェクトにおいて “invalid-name” の警告 218 件中 180 件 (82.5%) はレビューを通して修正されていない．

ASAT で検出した規約違反の修正がレビューで行われない理由として，ツールの警告内容が検証者の修正方針と合致していない可能性が考えられる．例えば，OpenStack プロジェクトは独自のコーディングガイドライン [64] に基づいてソースコードを記述している．Flake8 によって検出した警告のうち，一度も修正されていない警告が存在する原因の 1 つとして， OpenStack のコーディングガイドに記述されていない，または記述されていたとしても OpenStack 独自の例外を設けている [65] 点が挙げられる．そのため，プロジェクトが検証者に ASAT の利用を促した場合，方針に合わない警告内容を確認するコストが余分に発生する．ASAT は検出対象とするコーディング規約を選択することが可能であるにもかかわらず，多くのプロジェクトで対象とする規約の設定が変更されず運用されている事がわかっている [25]．警告の確認コストを削減するために，検証者のレビュー方針に合わせてツールを自動設定する技術が必要である．

RQ1-2 に対する答えは以下の通りである．

静的解析ツールによる警告はコードレビューを通して修正されているか：コードレビューを通して行われた変更のうち 13.4%のパッチはパッチ投稿前に ASAT を実行することで事前に修正が可能な修正である．一方で，パッチ投稿前に ASAT で検出可能な違反であっても警告の半分以上は，コードレビューを通して検証者に指摘されないままプロジェクトに統合されていた．検証者のレビュー方針と合致するよう，検証者，またはプロジェクト管理者は ASAT で検出する規約違反をプロジェクトに合わせて設定する必要がある．

3.3 妥当性への脅威

3.3.1 内的妥当性

目視によるソースコードの変更内容分析ではソースコードの差分，すなわち diff 形式のデータを使用して変更内容を分析している．ソースコードの差分情報の分析だけではソースコードの動作に与える影響の有無を確認できないパッチが存在する．したがって，RQ1-1 では，変更がコード改善であるか否かを diff 形式のデータだけで判断できない場合には Gerrit や GitHub で管理するソースコード全体を目視で確認した．今後はソフトウェアのテスト結果を用いることでソースコードの動作に与える影響の有無を自動的に判断する．

コード改善として “ADD_REMOVE_SPACE” や “CHANGE_VALUE_NAME” のように修正方法によってはソースコードの動作に影響を与える可能性のあるコード改善が存在する．目視での検出で影響がないことを確認したが，外部ファイルからの参照などを調査することで影響が発見されることも考えられる．

3.3.2 外的妥当性

本章ではコーディング規約違反の検出に Pylint と Flake8 を利用している．RQ1-1 の分析の結果，ソースコードの改善だけでなく，固有名詞や記法の修正が多く発生していることを確認した．今後の課題として，スペルチェックツールを利用し，文字列の修正方法を検証する．

本章では Python 言語で構成されたソースコードのみを対象としているため．Java 言語のようなインデントに意味を持たないプログラミング言語を対象とした場合に異なる分析結果になることが考えられる．例えば表 2 のラベルは本章で対象としたソフトウェアの変更内容に基づいて分類している．そのため，異なるソフトウェアや言語を対象とした場合，定義したラベルでの網羅は保証できない．今後は異なる言語を利用するプロジェクトも対象に分析する．

3.4 結言

本章ではコードレビューを通して行われるコード改善とその出現回数の分析と，ASAT によるコード改善への効果を分析した．その結果，対象プロジェクトのコードレビューで行われる変更のうち 56.0%はコード改善であることを明らかにした．レビューを通して修正が行われたパッチのうち，13.4% はレビューを行う前にパッチ開発者が ASAT で検出可能な修正である．例えば “ADD_REMOVE_SPACE”（空白やインデントの追加削除）のようなソースコードのスタイルの修正をコードレビューでの変更 384 件中，36 件行っている．一方で，“CHANGE_VALUE_NAME”（変数名の変更）や “CHANGE_STRING” （文字列の変更） のように，既存の ASAT で検出ができないコード改善が頻繁に実施されている．

続く 4 章で，実際に開発者が守る ASAT の規約のみを有効にするシステムを提案する．また，5 章で，ASAT での解決が困難なコード改善例をパターンとして抽出することで，パッチ投稿前に開発者が注意すべきソースコードとその改善方法を自動的に示すシステムの構築を目指す．

4. 静的解析ツール出力内容に基づく既存コーディング規約の最適化

4.1 緒言

ソースコードの可読性向上はソフトウェアの不具合発見および予防を容易にするための重要な作業である．多くのソフトウェアプロジェクトでは可読性の維持にかかる作業コスト削減のため，静的解析ツール (以降，**ASAT**) を利用している．多くのプログラミング言語では頻出する不具合の原因，スタイル，パフォーマンスの改善を目的としたコーディング規約（コーディング規約）が提案されており，ASAT はそれらのコーディング規約に違反するソースコード行を検出する．ASAT は，開発者が手作業で実行するだけでなく，ソースコード編集時やコードレビュー開始時に自動実行される形でも運用されている．

ASAT の特徴として高い拡張性がある．利用者は ASAT の設定ファイルを作成および編集することによって自動検出の対象となるコーディング規約の有効，無効を切り替えることができる．また ASAT の設定ファイルはプロジェクトに参加する開発者にプロジェクトの方針を明示することで，開発チーム内での知識共有にも貢献する．

ASAT は広く利用されている一方で，検出された警告の 80%以上が開発者に修正されないといったように，実質的な誤検出の多さも指摘されている [6, 7, 8]．原因の一つとして，ASAT を利用するプロジェクトの多くが ASAT 導入後にコーディング規約を編集しないこと，つまりそのプロジェクトの開発者が興味を持たないコーディング規約違反も余分に検出していることが挙げられている [25]．

本章では ASAT の規約精度の改善を目標とし，以下の 3 つの仮説のもと ASAT の規約を自動設定する手法を提案する．

- ASAT が標準で有効にしている規約集合や開発者が手動で設定した規約集合と，開発者が実際に準拠している規約には齟齬があり，そのために開発者にとって，規約違反の実質的な誤検出および検出漏れが発生する．

- あるバージョンで開発者が準拠している規約はその後のバージョンでも準

拠され続ける（規約は変化しない）．

- プロジェクトに含まれる一部のファイルだけが特定の規約を準拠する，ような局所的な規約は存在しない．

これらの仮説に基づき，本章では，ASAT が検査可能なすべてのコーディング規約のうち，ある時点で違反がなかった（対象プロジェクトのソースコードすべてが準拠している）規約をすべて有効化し，違反のある規約を無効化するように，ASAT を自動設定する手法を提案する．

本章の貢献は以下の 3 つである．

1. ASAT の誤検出を削減する手法の提案 (4.4 節)
2. JavaScript プロジェクトにおける ASAT による誤検出発生頻度および誤検出されやすい規約の調査 (4.5.2 節)
3. 提案手法で生成した規約セットと対象プロジェクトが作成した規約セットを比較した誤検出の評価 (4.5.3 節，4.5.4 節)

調査の結果，プロジェクトが明示的に有効にしている規約であっても，その 3.8% は実際には無視されていることを確認した．また，プロジェクトが有効にしている規約数よりも，実際に準拠している規約の数は 3.3 倍ほど存在することが判明した．

4.2 本提案手法の立ち位置

多くの ASAT はコーディング規約というソースコードの記述方法に制限を設ける規約を参照して実装されている．通常コーディング規約は利用するプログラミング言語ごとに定義されている．広く利用されているコーディング規約や ASAT として，Java のコーディング規約である Sun [66] への違反を検出する *Checkstyle* [31]，Python のコーディング規約違反を検出する *Pylint* [15]，*Flake8* [67]，*pydocstyle* [68] が存在する．利用するプログラミング言語に対応した ASAT を利用することで利用言語で頻繁に発生する実装のミスを前もって検出することが可

Listing 1: no-template-curly-in-string 規約違反と違反修正例

```
// 警告発生例
console.log("Hello␣${name}!");
console.log('Hello␣${name}!');
// ES5 以前の出力 > Hello World!
// ES6 以後の出力 > Hello ${name}!

// 警告修正例
console.log(`Hello ${name}!`);
// > Hello World!
```

能になる．本章では JavaScript のコーディング規約である EcmaScript [69] に基づきソースコードを検証する ESLint [18] を対象に分析を行う．

言語の仕様変化や機能追加に対応するため，ASAT は実装されたコーディング規約のうち，広く利用可能な規約を標準規約セットとして有効にしている．ESint の場合，241 の規約が検出可能な規約として実装されているが，開発者が ESLint の設定を編集しなければ 56 規約の標準規約セットのみを有効にしている．この規約セットは 2009 年時点の JavaScript 標準規約である ECMAScript 5 [70] に違反するソースコード行を検出する．標準規約である ECMAScript 5 が効果的な例として Internet Explorer (IE) に対応した Web アプリケーション開発がある．ESLint の設定を編集せず標準規約を利用することで，IE ではサポートされていない ECMAScript 6 の規約を無効化したソースコード検証を可能にする．

事前分析で確認した ASAT 標準規約の過不足例を 2 件示す．まず，ASAT の標準規約では検出されないにも関わらず多くのプロジェクトが準拠している実質的な検出漏れ規約例，2 つ目は，標準規約セットで検出するものの多くのプロジェクトが準拠しない実質的な誤検出規約例を示す．分析手法の詳細は RQ2-1 として 4.5.2 節に記述する．

まず，標準規約セットに含まれていないが広く準拠されている規約の一つ *no-template-curly-in-string* の例を Listing 1 に示す．テンプレートリテラルという文字列連結を省くための処理を行っており，JavaScript ではバッククオート（`）内に “`${variable}`” のような変数または式を含む文字列を作成することができる．

Listing 2: no-empty 規約違反と違反修正例

```
// 警告発生例
if (foo) {
}
try {
  doSomething();
} catch(ex) {

}

// 警告修正例
if (foo) {
  console.log(foo)
}
try {
  doSomething();
} catch(ex) {
  // エラーがなければ続行
}
```

ただし，ECMAScript 6（ES6）以降の JavaScript で，テンプレートリテラルを使用する場合，"${variable}"のような，誤った引用符を利用すると，挿入された式の値を含む文字列の代わりに素の文字列 ${variable} が出力されてしまう．最新版の JavaScript 言語を利用する開発者がこの誤りを検出するには，ESLint を設定し規約を有効にする必要がある．

次に，標準規約セットに含まれているものの広く準拠されていない規約として *no-emply* の例 Listing 2 に示す．*no-emply* はソースコードを読む際に混乱の原因となる空のブロックの作成を制限する．本章が対象としたプロジェクトでは *no-emply* のようなソースコードの動作に影響を持たない規約違反は修正されにくい．そのため，標準規約セットでは本来開発者が優先して修正を行う必要があったはずの規約違反に混入して，開発者にとって重要でない規約違反が検出されてしまう．

4.3 提案手法の利用方法

本章では，提案手法の具体的な実装として，エディタ上でのインタフェースを試作した．本提案手法の利用方法として，プロジェクト管理者がソフトウェア更新時に合わせて ASAT 設定ファイルを同期させることを想定している．このとき，利用者は Visual Studio Code (以下，VS Code) の拡張機能を通して本手法を利用する．以下に本手法を利用し，図 4 に示す ASAT 設定ファイル更新のワークフローを示す．

1. Visual Studio Code から本提案手法をインストールする

    1. アクティビティバー (図 4 左端) から拡張機能アイコン (上から 5 番目のアイコン) を選択する
    2. アクティビティバー隣にあるサイドバーの検索スペース (図 4 左上) に “linter-maintainer” と入力する
    3. “install” ボタンを選択する

4. Visual Studio Code エディタで ASAT 設定ファイルを開く．図では ESLint の規約設定ファイルである *.eslintrc.json* を開いている．

5. ファイル右上にある電球ボタンをクリックする．またはこの電球ボタン右の矢印をクリックすることで，以下の実行オプションを選択できる．

    - Add Rules: 対象となる設定ファイルのパス以下のファイルが遵守している規約のうち，設定ファイルで有効にされていない規約を有効化する．
    - Remove Rules: 対象となる設定ファイルのパス以下のファイルが違反している規約のうち，設定ファイルで有効になっていｄる規約を無効化する．
    - Add/Remove Rules: Add Rules と Remove Rules の両方を実行して規約を更新する．電球ボタンをクリックしたときも同様の動作を行う．

6. 誤検出もしくは検出漏れが存在した場合，ASAT 設定ファイル *.eslintrc.json* の内容が更新される．

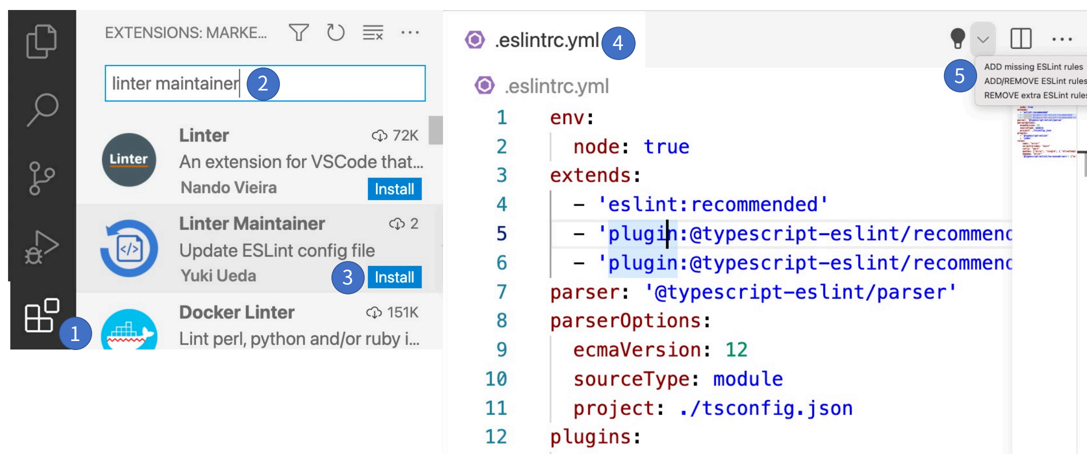

図 4: 本提案手法のインストール及び実行ワークフロー

以上の手順を，ソフトウェア変更時や利用する ASAT のバージョン更新時に実行することで，利用者は利用するコーディング規約とプロジェクトの内容を継続的に一貫させることができる．

4.4 ソフトウェアフレームワーク

本提案手法では，ASAT の精度向上を目指し，誤検出及び検出漏れの通知を行う．提案手法は，プロジェクトが利用している ASAT で利用可能な規約の集合 R_{All} とプロジェクトのソースコード S から，そのプロジェクトで有効にするべき規約の部分集合（すなわち，その ASAT にとっての設定ファイル）$Config(S)$ を生成する．

各規約 $r_i \in R_{All}$ を，与えられたソースコード集合 S からソースコード違反箇所の集合を取得する関数 $r_i(S)$ とみなすと，開発者が従っている（準拠する）規約とは $|r_i(S)| = 0$ であるような規約，開発者が従っていない（非準拠である）規約 $|r_i(S)| > 0$ であるような規約であると表現できる．提案手法が求める $Config(S)$

は，以下のように定義できる．

$$Config(S) \;=\; \{r_i \in R_{All} \;:\; |r_i(S)| = 0\}$$

提案手法を利用すると，あるバージョン j のソースコード S_j から規約 $Config(S_j)$ を生成し，それ以降のバージョンのソースコード S_{j+k} $(k \geq 1)$ に対して適用することができる．このとき検出される規約違反の行の集合を，以下のように表現する．

$$Violations(S_{j+k}) \;=\; \bigcup_{r_i \in Config(S_j)} r_i(S_{j+k})$$

この規約違反の行の集合は，バージョン j の時点で準拠していた規約に対する違反のみを検出するため，開発者が常に無視し続けるような規約の違反を含まなくなり，誤検出の件数が減少することが期待される．また，ASAT の標準規約セットでは有効ではない規約を有効にするため，開発者が目視で検出していた違反を自動的に検出できるようになる．一方で，開発者は準拠するつもりであったがそのバージョンで一時的に違反していた規約を無効化するため，新たな見逃しが発生する可能性もある．

4.5 ケーススタディ

本提案手法の有用性を定量評価するために，開発者が実際に利用している規約と提案手法によって更新した規約の精度を評価する．具体的には以下の調査課題に基づき評価する．

1. **RQ2-1: どの規約にソフトウェアプロジェクトは実際に準拠しているか？**
   事前分析として，提案手法が対象とする誤検出および検出漏れの規模を明らかにする．具体的には，プロジェクトが従う ASAT の規約とそうでない規約の件数を計測する．

2. **RQ2-2: プロジェクトが従う規約は導入時から最新まで一貫してるか？**
   開発者が規約の更新をどの程度の頻度で行えばよいか計測する．そのため

に，対象プロジェクトへ提案手法を実行し，バージョン間をまたいだ誤検出および検出漏れの増減を計測する．

3. **RQ2-3: 提案手法により規約違反の誤検出および検出漏れを削減可能か？** 提案手法の精度を評価するために，提案手法による規約を更新することで早期発見できる規約違反の件数を調査する．

4.5.1 データセット

本調査では，最も人気のある言語の一つである JavaScript を対象とする．また，対象とする ASAT として，JavaScript で最も利用されている ESLint [71] を採用する．プロジェクト間での規約を一貫させるために，ESLint のバージョンは 7.4.0 を利用している．

分析対象プロジェクトには，JavaScript のバグデータセット BugsJS に登録されているプロジェクトを 10 件を利用する．これらのプロジェクトは GitHub 上で最も人気のあるのサーバーサイドプロジェクトから選定されている．本章ではその中から，ESLint を採用しているプロジェクト 4 件を対象とする．

4 プロジェクトは合計で 643 リリースバージョンを持つ．本章ではそのうち，バージョン番号名が “x.y.z” となっている 561 バージョンを対象とし，それぞれ x, y, z の変更を Major, Minor, Maintenance リリースとする．このとき，“x.y.z-b” や “test-version” といった 82 件のバージョンは，時系列分析での順序付けが困難かつ，プロジェクトによって形式が異なるため除外している．

表 5 に各プロジェクトのリリースバージョン数を示す．Major リリースはバージョン名が “x.0.0” 形式のバージョンを対象とする．Minor リリースはは “x.y.0” 形式のバージョンであり，Major リリースバージョンを含む Maintainance リリースは “x.y.z” 形式のバージョンであり，Minor リリースバージョンを含む．

4.5.2 RQ2-1: どの規約にソフトウェアプロジェクトは実際に準拠しているか？

事前分析として，対象プロジェクトで利用されている規約と標準規約セットとの差異を調査する．すべての対象プロジェクトの 2020 年 7 月 1 日時点で最新版の

表 5: RQ2: 評価対象プロジェクトのリリースバージョン数

プロジェクト	Major	Minor	Maintenance	全て
bower	1	19	89	96
eslint	7	119	165	223
hexo	4	30	120	126
karma	5	29	187	198
合計	17	197	561	643

ソースコードから $Config(S)$ を抽出し，以下の 3 つの区分へと分類する．

- Followed: すべてのプロジェクトが準拠している規約．すなわち，4 つのプロジェクトの $Config(S)$ の共通部分．
- Unfollowed: すべてのプロジェクトが違反している規約．すなわち，R_{All} に含まれているが，どのプロジェクトの $Config(S)$ にも含まれなかった規約．
- Specific: プロジェクトによって準拠状態が異なる規約．すなわち，1 つから 3 つのプロジェクトの $Config(S)$ に含まれている規約．

また，各カテゴリの規約が，ASAT の標準設定での規約に含まれているかを調査する．表 6 に結果を示す．

Followed 規約 (64 件): まず，プロジェクトが実際に準拠している規約について 64 件のうち標準規約セットに含まれていない規約を 26 件 (40.6%) 確認した．代表的なものとして 4.2 節の Listing 1 でも示した *no-template-curly-in-string* や，再代入のない変数に const を付与する *prefer-const*，オブジェクトへのアクセスに角括弧でなくドットを利用する *dot-notation* がある．これらの規約に対する違反は JavaScript の実行時性能にも影響を及ぼす可能性があるため，開発者が意識的に準拠している可能性がある．

Unfollowed 規約 (95 件): 全ての分析対象プロジェクトが準拠していない Unfollowed 規約 95 件のうち，91 件は標準規約セットに含まれていない．標準規約セットに含まれている Unfollowed 規約は以下の 4 件である，

表 6: RQ2-1: 対象プロジェクトで共通して準拠されている規約分布

分類	規約数	標準規約数	非標準規約数
Followed	64	38	26
Unfollowed	95	4	91
Specific	82	15	67
合計	241	57	184

- no-empty: 空の構造文ブロックを生成しない
- no-undef: 関数内で宣言されていない変数を利用しない
- no-unused-vars: 利用しない変数を宣言しない
- no-extra-semi: 不要なセミコロンを書かない

これらはソースコードの可読性を向上させるものの，動作に影響がないため，開発者が見逃している可能性がある．

Specific 規約 (82 件): 最後に対象プロジェクトによって準拠状況が異なる規約を 82 件発見した．そのうち標準規約セットとして広く利用されている規約は 15 件該当する，Listings 3 に以下の規約違反例 2 件を示す.

- no-un-reachable: 実行されないコード行を削除する
- no-constant-condition: 条件式に固定値を利用しない

これらはソースコードの動作に関連するものの，実行されない．開発者が意図的に Listings 3 のような記述を行っている場合，これらの規約違反は開発者にとって誤検出となる.

RQ2-1 に対する答えは以下の通りである.

どの規約にソフトウェアプロジェクトは実際に準拠しているか?: 対象プロジェクトで共通して 134 件の規約が準拠されているが，そのうち標準規約セットに含まれていない，26 件の規約は ASAT の設定を編集しなければ検出できない．また，標準規約セットに含まれる規約のうち 19 件は一つ以上の対象プロジェクトで準拠されていないものの，プロジェクトの動作に影響を及ぼさない.

Listing 3: RQ2-1: JavaScript における *no-unreachable* and *no-constant-condition* 規約違反例

```
// no-unreachable 違反例
function foo() {
    if (Math.random() < 0.5) {
        return True;
    } else {
        throw new Error();
    }
    // 以下の行が実行されない
    return False;
}

// no-constant-conditon 違反例
// 以下の条件式が常に偽となる
if (false) {
    doSomethingUnfinished();
}
```

4.5.3 RQ2-2: プロジェクトが従う規約は導入時から最新まで一貫してるか？

本提案手法によって作成した規約セットの精度を評価する．ソフトウェアプロジェクトの時系列順に並んだバージョンを $S_1, S_2, \cdots, S_n$ とする．つまり，本提案手法による規約は以下のように評価する．$r \in Config(S_i)$ の場合，中間バージョン S_i $(1 \leq i < n)$ の規約 $r \in Config(S_n)$ を適切な規約とする．つまり，$r \notin Config(S_n)$ の場合，規約 $r \in Config(S_i)$ は誤検知とみなす．また，$r \in Config(S_n)$ の場合，規約 $r \in (R_{All} \setminus Config(S_i))$ は検出漏れとみなす．

評価では 3 つのリリースバージョン Major, Minor, Maintenance を利用する．各バージョン S_i に対して規約セット $Config(S_i)$ を生成し，それぞれの Precision(精度)，Recall (網羅率)，F 値を評価する．比較対象として S_i 時点で，プロジェクトが ASAT 設定ファイル内で定義している規約セットである $P(S_n)$ を用いる．

表 7 に各リリースバージョンごとに提案した規約セットの精度，網羅率，F 値を示す．結果より，最低でも Minor リリースごとに規約セットを更新することで規約

セットの精度を改善可能であることを確認した．更新頻度の中では，Maintenance リリースごとに更新した規約の精度が最も高い．一方，Major バージョンごとに更新した規約セットは karma プロジェクトにおいて，プロジェクトの規約よりも低い精度を記録した．この結果から，プロジェクトが実際に従うコーディング規約は時間の経過とともに変化することが考えられる．規約セットの精度を保つために，開発者は ASAT の規約セットを定期的に更新する必要がある．

プロジェクトが設定した規約の網羅率は提案手法のどのリリースタイミングと比較しても低い．この結果から，プロジェクトの開発者は従っている規約を把握していない，もしくは知っていても設定ファイルに反映していない．本提案手法を利用することで，これらの潜在的な規約を開発者に通知することができる．

RQ2-2 に対する答えは以下の通りである．

RQ2-2:プロジェクトが従う規約は導入時から最新まで一貫してるか?: 対象プロジェクトが手動で ASAT に設定している規約の網羅率は 0.16 から 0.39 であり，プロジェクトが従っている潜在的な規約を見逃している．提案手法を用いて，規約を最低でもメジャーバージョンごとの頻度で更新することで，規約違反検出の網羅率を 0.90 から 0.95 まで改善することができる．また，規約の更新頻度を頻繁にするほど，規約の精度および網羅率ともに改善する．

表 7: RQ2-2: 各リリースごとに提案手法を適用した規約および，プロジェクトが設定した規約の精度，網羅率，F 値

規約セット	精度: Precision		網羅率: Recall		F 値	
	平均値	中央値	平均値	中央値	平均値	中央値
bower (n=96)						
提案手法 (Major)	0.93	0.94	0.95	0.95	0.94	0.95
提案手法 (Minor)	0.99	0.99	0.98	0.99	0.98	0.99
提案手法 (Maintenance)	0.99	1.00	0.99	1.00	0.99	1.00
プロジェクトの規約セット	0.90	0.89	0.34	0.34	0.50	0.50
eslint (n=9,223)						
提案手法 (Major)	0.98	0.99	0.90	0.99	0.93	0.98
提案手法 (Minor)	0.99	1.00	0.98	1.00	0.98	1.00
提案手法 (Maintenance)	0.99	1.00	0.99	1.00	0.99	1.00
プロジェクトの規約セット	0.29	0.00	0.16	0.00	0.21	0.00
hexo (n=126)						
提案手法 (Major)	0.92	0.92	0.92	0.93	0.92	0.93
提案手法 (Minor)	0.98	0.99	0.98	1.00	0.98	1.00
提案手法 (Maintenance)	0.99	1.00	0.99	1.00	0.99	1.00
プロジェクトの規約セット	0.71	0.89	0.27	0.33	0.39	0.48
karma (n=198)						
提案手法 (Major)	0.77	0.70	0.90	0.88	0.82	0.78
提案手法 (Minor)	0.97	0.99	0.96	1.00	0.96	0.99
提案手法 (Maintenance)	0.99	1.00	0.99	1.00	0.99	1.00
プロジェクトの規約セット	0.83	0.81	0.39	0.39	0.53	0.53

4.5.4 RQ2-3: 提案手法により規約違反の誤検出および検出漏れを削減可能か？

RQ2-2では，規約の精度と網羅率を調査した．本分析では，規約違反の精度と網羅率を，提案手法によって生成した規約セットとプロジェクトが設定した規約セットで比較する．ソフトウェア更新の過程で開発者によって解決された違反の数をために，規約違反を $Resolved$ と $Unresolved$ に分類する．ある規約 r の違反がバージョン i でのファイル f_i で発生し，別のバージョンでの同一ファイル f_k $(i < k)$ で解決されている場合，$|r(f_k)| = 0$ とし，r を $Resolved$ に分類する．各 $|r(f_k)| = 0$ であるファイルのバージョンに，それまでの解決されたバージョンと現在のバージョンの間で検出した違反の最大値を $Resolved$ としてカウントする．図5に，規約 r の違反がファイル f で発生した際のカウント例を示す．図のバージョン $n-5$ から $n-3$ は規約に違反しているが，バージョン $n-2$ は規約に違反していない．この場合，バージョン間の違反の最大数である10を $Resolved$ 違反としてカウントする．開発者がバージョン $n-5$ と $n-4$ の間で他の違反を解決することもあるものの，*max-params* などの一部の違反が後続のバージョンで再表示される可能性があるため，これらはカウントしない．そのため本分析では，バージョン $n-2$ を解決したタイミングであるとする．提案手法がバージョン間で規約 r を有効にした場合，違反が検出されるため，True Positive としてカウントする．逆に提案手法がバージョン間で規約 r を無効にしていた場合，それらを False Negative，つまり検出漏れとしてカウントする．図5では，バージョン $n-1$ 発生した5つの規約違反は，最新バージョン n に残っているため，$Unresolved$ としてカウントする．提案手法がバージョン間で規約 r を有効にしていた場合，それらを False Positive，つまり誤検出としてカウントする．それ以外の場合は，検出されず，かつ修正もされないため，True Negative とする．

$$FalseNegative(i) = \sum_{k=i}^{n-1} \sum_{p=k}^{n} \sum_{r \in V(S_i)} max(|r(f_k)| - |r(f_{k+p})|, 0) \quad (1)$$
$$where\ V(S_i) = R_{All} \setminus Config(S_i)$$

提案手法を評価するために，以下のユースケースシナリオを想定する．開発者

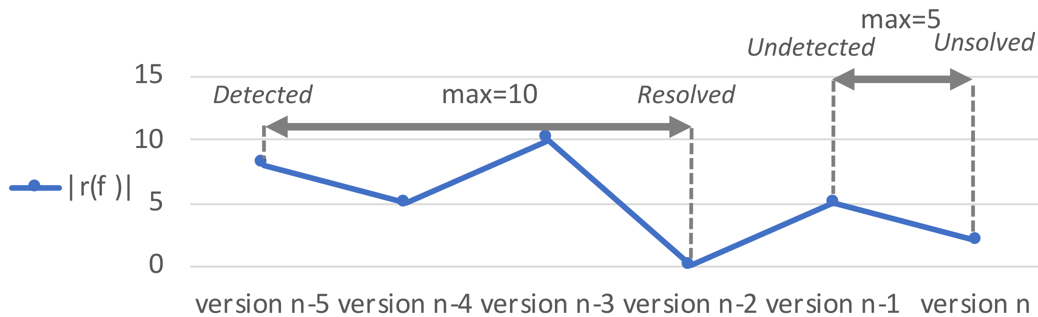

図 5: RQ2-3: 規約違反数のカウント方法

は，Major/Minor/Maintenance のすべてのバージョンで規約セット $Config(S_i)$ を更新する．生成された規約セットを適用して，すべての Maintenance リリースで違反を検出する．$Config(S_i)$ と検出された規約違反の履歴を使用して，上述した $Resolved$ および $Unresolved$ の違反を True Positive, True Negative, False Positive(誤検出), False Negative（検出漏れ）に分類する．

表 8 には，5 つの規約セットの精度，網羅率，および F 値を示す．Major，Minor，および Maintenance の行には，提案手法によって生成された規約セットの結果を示している．プロジェクトの規約セット行には，プロジェクトに含まれている規約セット $P(S_i)$ の結果を示す．標準規約セット行には，ESLint の標準規約セットの結果を示す．

Major リリースごとに規約を更新するとき，最も高い精度を得た．RQ2-2 と異なり，頻繁に更新することで，一時的な違反が原因で，必要な規約が誤って無効になることが考えられる．すべてのプロジェクトで，生成された規約セットは，手動で作成されたプロジェクトの規約セットよりも精度が高くなる．これは，提案手法が開発者が無視する規約を無効にしたことが原因となる．また，eslint と karma の 2 プロジェクトでは，提案手法を適応することで網羅率を向上させた．プロジェクトの開発者が ASAT のサポートなしで多くの違反を解決したことが原因である可能性がある．本提案手法を利用することにより，開発者は将来の違反を検出できる．

RQ2-3 に対する答えは以下の通りである．

表 8: RQ2-3: 提案手法によって検出した ASAT 違反行数の予測 Precision, Recall, F 値 (n=405,094,892)

	bower (n=4,174,637)			eslint (n=363,074,539)		
規約セット	Precision	Recall	F 値	Precision	Recall	F 値
提案手法 (Major)	0.95	0.02	0.05	0.27	0.03	0.06
提案手法 (Minor)	0.97	0.01	0.01	0.37	0.02	0.04
提案手法 (Maintenance)	0.95	0.01	0.01	0.44	0.02	0.04
プロジェクトの規約セット	0.80	0.04	0.07	0.16	0.00	0.00
標準規約セット	0.80	0.04	0.07	0.34	0.01	0.02

	hexo (n=14,416,765)			karma (n=23,428,951)		
規約セット	Precision	Recall	F 値	Precision	Recall	F 値
提案手法 (Major)	0.56	0.03	0.06	0.87	0.19	0.31
提案手法 (Minor)	0.46	0.02	0.03	0.84	0.05	0.10
提案手法 (Maintenance)	0.46	0.02	0.03	0.83	0.05	0.09
プロジェクトの規約セット	0.51	0.04	0.07	0.83	0.05	0.09
標準規約セット	0.51	0.04	0.07	0.55	0.01	0.02

RQ2-3:提案手法により規約違反の誤検出および検出漏れを削減可能か?: ASAT の標準規約を利用する場合と比較して，提案手法を利用することで，誤検出される規約違反を最大で 30%多く検出可能になる．

4.6 妥当性への脅威

4.6.1 内的妥当性

本手法は誤検出の発生を抑えるために，プロジェクト内の殆どのファイルが規約に準拠していたとしても，一つのファイルが準拠していなければその規約を無効とする．例えば検出漏れを最小化するために，過半数のファイルが準拠する規約を有効にする場合は，ASAT の Precision と Recall がトレードオフの関係になる．

RQ2-2 において，発生した規約違反を修正された行数を数え上げ，規約セットの精度を評価している．そのため，ファイルの改名の削除が発生した場合も規約違反の解決が行われたものとして余分に数え上げている．ただし，ファイル名の改名があったとしても，プロジェクト内での規約違反の総数は変化しない．

4.6.2 外的妥当性

プロジェクト間で規約の数を均等化するために，外部ライブラリの規約は除外した．また，本手法は JavaScript プロジェクトとそれに対応する ASAT である ESLint をサポートしている．将来の機能追加として Java ソースコードを検証する Pmd や Python に対応した Pylint のサポートを予定している．

4.7 結言

本章では，ソフトウェアプロジェクトのソースコードが準拠しているコーディング規約に基づき ASAT を自動設定する手法を提案した．本提案手法は，ASAT によって一箇所でも対象ソースコードから検出された規約違反を無効にし，検出されなかった違反を有効にする．実装としては，コマンドラインインターフェースおよびコードレビュー補助システムとして開発者が作成したソースコードを自動的に検査し，誤検出や検出漏れとなる規約を通知する機能を提供する．提案手法の性能評価として，JavaScript プロジェクトのバージョン 643 件と ESLint に実装されている 241 件の規約を対象に提案手法を適用することにより規約集合を生成した．本規約集合と開発者が手動で生成した規約集合を比較して，ASAT による誤検知の 90% を削減できることを確認した．

今後の課題として，開発履歴に基づき有効にする規約を順位付けすることで，開発者が現状の規約からの移行作業を行いやすくする．また，対象プロジェクトおよび対象言語を増やすことで手法および評価の一般性を得ることが挙げられる．現在著者らは TypeScript 言語に対応した ESLint 実装，および Java 言語に対応する PMD [72]，Python 言語に対応する Pylint [15] の規約を最適化する実装を公開している．

5. コード編集履歴を用いた新規コーディング規約セットの抽出

5.1 緒言

ソフトウェア開発において，メンテナンス作業は最も時間的コストの高い作業である [73, 74]．メンテナンス作業を削減するために，多くのソフトウェアプロジェクトが静的解析ツール (ASAT) を用いて，コーディング規約違反を自動的に検出している [75]．ASAT は現在，オープンソースソフトウェアと企業の両方で広く利用されている [76]．

ASAT が普及している一方で，実際の開発者は 80%以上の警告を無視していると報告されている [8]．つまり，ASAT の出力内容の多くは開発者にとって有用でなく，見逃されている．利用者は検出に利用する規約の無効化を行えるが，これを行うと将来の有用な推薦を削減する．また，規約セットを選定するにはツールとプロジェクト両方への理解が必要となる．

プロジェクト固有のコーディング規約違反を発見するため，開発者は検索/置換機能 (Find And Replace，以降 F/R) および `grep` コマンドを活用する [77]．検索/置換は多くのコーディングエディターに実装されており，`grep` コマンドも同様に多くの OS に事前インストールされている．ただし，利用者はこれらの機能もしくはコマンドを明示的に実行する必要がある．そのため，開発者は修正漏れなどが原因で同様の変更を一ヶ月以内に繰り返すことも報告されている [10]．

ASAT および F/R 機能の問題を解決するために，本論文では静的解析ツール DevReplay を提案する．DevReplay は自動的に開発者が従いやすいコーディング規約セットを生成する．本ツールは規約セットのカスタマイズ性を向上させた ASAT として実装している．例えば，開発者はマウスクリックで正規表現を用いたコーディング規約を生成することができる．また，本ツールは検索/置換機能に自動化機能を付属させている．DevReplay はコード変更の推薦をコードエディターおよびコードレビュープロセスで行う．

5.2 本提案手法の立ち位置

ASAT，検索/置換，DevReplay(提案手法) の 3 つのソースコード検証手法を紹介する．表 9 に各ツールが持つ機能の特徴を示す．

まず，ASAT について紹介する．ASAT はオープンソースと企業両方で広く利用されている．ASAT には 3 つの利点がある．(1) 自動化機能： 利用者はエディタや，コードレビュー，コマンドインターフェース上で自動的に ASAT を実行することができる．(2) 変更再現性： 利用者は ASAT を異なる開発環境でも同一の動作を実行することができる．(3) 規約汎化性能： ASAT が用いる規約は多くのプロジェクトで利用可能な汎用性の高い規約であり，長期間に渡ってツール開発者が洗練している．一方で，他ツールと比較した ASAT の欠点として，規約の特化性能がある．ASAT の規約はプロジェクトに特化していない．そのためいくつかの企業はそれぞれに特化した規約を作成しているものの，規約の作成に要求される技術は低くない．

機能	ASAT	検索/置換 (F/R)	DevReplay
自動化性能	x		x
変更再現性	x		x
規約汎化性能	x		
規約特化性能		x	x

表 9: 各ツールの機能比較

本論文では F/R の機能と ASAT のインターフェースを兼ねたツール DevReplay を提案する．DevReplay は以下の機能からなる．(1) 自動化性能：DevReplay はソースコードの検証をローカルのコードエディタとオンライン上のコードレビューシステムで自動実行する．利用者は適用するコード変更推薦を選択することができる．また，利用者はコード編集中に正規表現コーディング規約セットを自動生成することができる．(2) 変更再現性：既存 ASAT と同様に DevReplay はコーディング規約ファイルを参照してソースコードの修正を推薦する．利用者は規約セットファイルを複製することで，他プロジェクトや各インターフェイス

間でも一貫した修正推薦を得ることができる．(3) 規約特化性能：DevReplay は規約に正規表現を採用している．正規表現は多くの ASAT が採用している抽象構文木よりも読解および編集が容易な代わりに，表現範囲や厳密性に劣る．正規表現を用いることで，利用者はコーディング規約セットを容易に編集できる．一方で，DevReplay は検索/置換機能と同様に，規約の汎化性能を持っていない．規約の品質は利用者の正規表現に関する知識に依存する．利用者の練度による品質の不安定さを軽減するために，著者らは規約の調整機能 (5.4.1 節) と規約を自動生成する機能　(5.4.2 節) を提供している．

5.3 提案手法の利用方法

DevReplay は Node package manager (**npm**) パッケージとして配布している．修正提案内容の一貫性を保つために，DevReplay は 3 つのプラットフォームで動作する（Visual Studio Code (**VS Code**) エディタ, CLI, GitHub Actions）．本節では各プラットフォーム上での DevReplay 利用方法を紹介する．

5.3.1 事前準備

利用方法説明のために，本節ではサンプルソフトウェアプロジェクトを利用する．サンプルプロジェクトは以下のコマンドからダウンロードすることができる．

```
# GitHub からサンプルプロジェクトをクローンする
$ git clone https://github.com/devreplay/devreplay-usage-samples.git

# VS でプロジェクトを開く．Code ‘code‘ コマンドがない場合はアプリケーションから開く
$ code devreplay-usage-samples
```

サンプルプロジェクトには 2 つのファイルが含まれている．一つは Python プログラム “`target.py`” (Listings 4)．もう一つはコーディング規約セットファイル “`.devreplay.json`” (Listings 5) である．本節では “`.devreplay.json`” を用いて “`target.py`” にある 3 行のソースコードを 1 行に短縮する修正を行う．

Listing 4: 修正対象ソースコード例 'target.py'

```
1  a = 1
2  b = 2
3  c = 3
4
5  tmp=a
6  a=b
7  b=tmp
8
9  tmp=c
10 c=b
11 b=tmp
```

Listing 5: DevReplay 規約セットファイル例 ".devreplay.json"

```
1 {
2   "before": [
3     "tmp=a",
4     "a=b",
5     "b=tmp"
6   ],
7   "after": "a,b=b,a"
8 }
```

5.3.2 Visual Studio Code

コードエディタでの DevReplay の利用について，著者らは VS Code 拡張機能を提供している．VS Code は 2021 年現在最も普及しているソースコードエディタであり，Windows, macOS, 複数の Linux ディストリビューション上で動作する．

利用者は以下の手順で DevReplay 拡張機能をインストールする．図 6 にインストール時のスクリーンショットを示す.

1. アクティビティバー (図左端) から拡張機能アイコン (上から 5 番目のアイコン) を選択する
2. アクティビティバー隣にあるサイドバーの検索スペースに "devreplay" と入力する

3. 検索結果から “DevReplay” を選択する

4. “install” ボタンを選択する

拡張機能インストール後，VS Code は開いているファイルを自動的に検証する．図 7 では，DevReplay は 5-7 行目のソースコードに対して警告を出し，図下部のパネルに警告内容を表示している．利用者はソースコードファイル “`target.py`” の修正と規約の生成を以下の手順で行う．

1. **コード修正**: 利用者は 5 行目の電球をクリックし，ソースコード修正コマンド “*Fix to a,b=b,a*” を選択する．Listings 5 の規約は 3 行のコードを 1 行にする内容となっている．このとき，複数箇所が修正可能な場合は，DevReplay はすべての箇所を修正するコマンド “*Apply all fixes for X rules*” も提供する．

2. **規約セット生成**: DevReplay 拡張機能は図 7 右上のボタンを選択することで，規約セットを自動生成する．生成された規約セットは規約ファイル “`.devreplay.json`’ に蓄積される（Listings 6）．生成した規約セットを用いて，DevReplay は Listings 4 の 9-11 行目のように正規表現に合致するソースコードに対しても修正を提案する．

詳しい規約の内容については 5.4.1 節で紹介する．

5.3.3 コマンドラインインターフェース

VS Code 上で生成した規約セットは CLI 上でも再利用が可能である．CLI では，利用者は以下の *npm* コマンドから DevReplay をインストールする．

```
$ npm install -g devreplay
```

VS Code の場合に引き続き，本節では “`target.py`”(Listings 4) と “.devreplay.json” (Listings 5) を用いる．検証を行う場合は以下のコマンドを実行する．

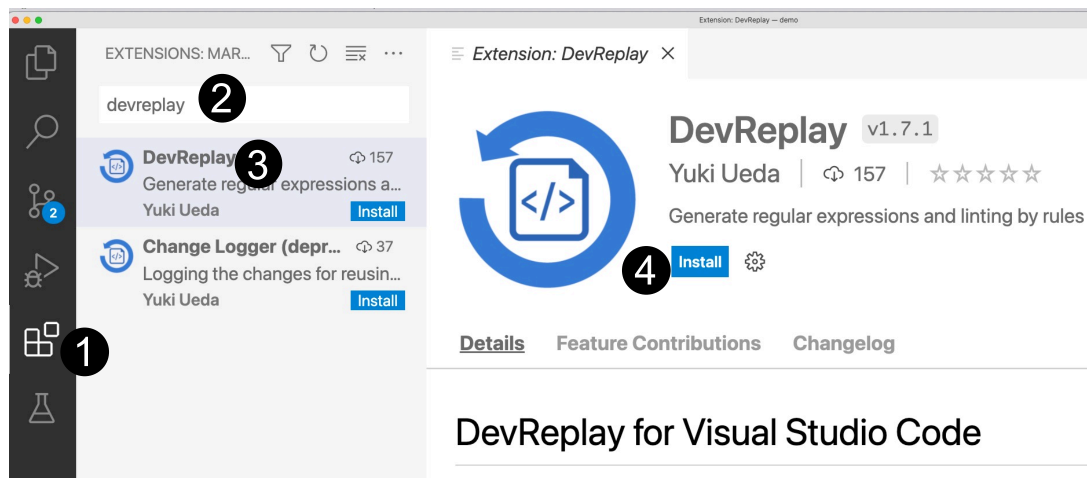

図 6: Visual Studio Code への DevReplay 拡張機能インストール手順

```
$ devreplay target.py .devreplay.json
```

修正を行う場合は，“`--fix`” オプションを追加する．例えば，“`target.py`” を修正する場合は次のコマンドを実行する．

```
$ devreplay --fix target.py .devreplay.json > target.py
```

このコマンドは VS Code 拡張機能の “*Apply all fixes for XX rules*” と同じ処理を行う．

5.3.4 GitHub Actions

利用者は DevReplay をコードレビュープロセス上で実行することができる．DevReplay を有効にするには GitHub Actions ファイル “`.github/workflows/devreplay.yml`”(Listings 7) を作成する．GitHub Actions は GitHub のサービスであり，継続的インテグレーションやデプロイ処理の自動化を支援する．GitHub 上でのプッシュ（更新）やプルリクエスト（変更提案）が発生するたびに，DevReplay は規約に合致するソースコードを通知する．図 8

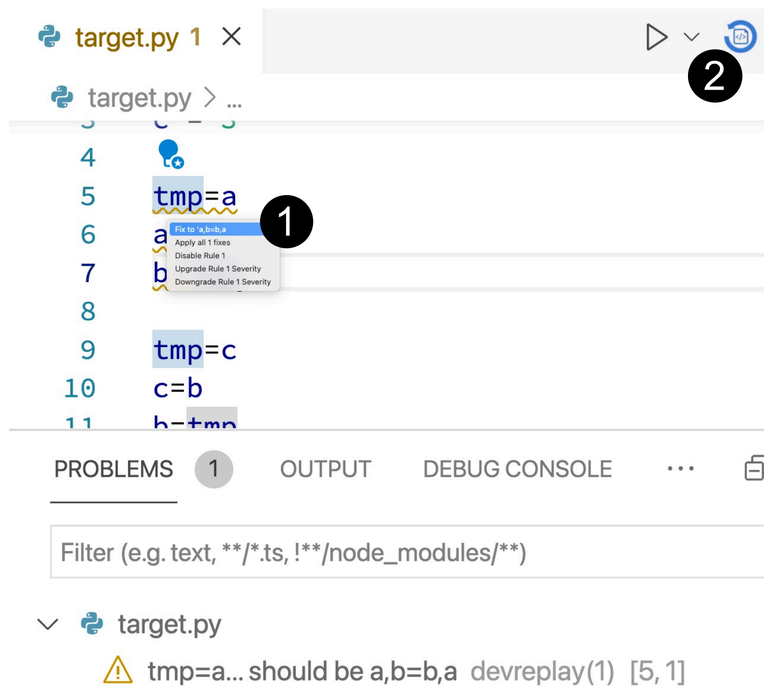

図 7: Visual Studio Code 上でのソースコード修正および規約セット自動生成

はプルリクエストに対する警告例を示している．

5.4 ソフトウェアフレームワーク

本節では，DEVREPLAY 規約の詳細と，規約セット自動生成プロセスの説明を行う．

5.4.1 規約ファイル要素

DEVREPLAY が利用する “`.devreplay.json`” は単一の規約要素，もしくは規約配列オブジェクトを持った JSON フォーマットによって構成される．

重要な要素として `before` と `after` プロパティがある．`before`，`after` プロパティは文字列，もしくは文字列配列からなる．`before` プロパティは DEVREPLAY を実行するために最低限必要となる．また，`after` プロパティを定義することで

Listing 6: Listings 5 を元に自動生成された “.devreplay.json”．規約要素型であったListings 5 から規約リスト型に変更している

```
 9    },
10    {
11      "before": [
12        "tmp=(?<a>\\w+)",
13        "\\k<a>=(?<b>\\w+)",
14        "\\k<b>=tmp"
15      ],
16      "after": "$1, $2=$2, $1",
17      "isRegex": true,
18      "matchCase": true
19    }
20  ]
```

ソースコードの修正機能を提供する．利用者が `after` プロパティを定義していない場合，DevReplay は単に警告のみをエディタ上で報告する．利用者は文字列配列フォーマットを利用することで，連続した複数行のコードを検索，置換することができる．

また，利用者は以下のプロパティをオプションとして追加することで，検索，置換機能をカスタマイズすることができる．このプロパティは VS Code エディタに実装されている検索オプション機能と同じ動作をする．

- `before` 関連オプション (真偽型)
  - 正規表現 (`isRegex`): ソースコード検索時に正規表現を使用する．自動生成時のコーディング規約はこのオプションを有効にしている（例：Listings 6）．この規約は 3 行のコードを短縮する．`before` プロパティにおいて，`.+` は 0 個以上の任意の連続した文字列, `\\s*`は 0 個以上の空白文字， and `\\w+`は 1 つ以上の英数字文字列を示す. `after` プロパティにおいて， `$1` は丸括弧で囲まれた 1 つ目の `before` プロパティ要素に対応する.
  - 完全一致 (`wholeWord`): 完全一致した単語のみを検索する (例：*editor*

Listing 7: GitHub Action ファイル “.github/workflows/devreplay.yml”

```
name: Lint
on: [push, pull_request]
jobs:
  devreplay:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-node@v2
        with:
          node-version: "14.x"
      - run: npm install -g devreplay
      - name: Run devreplay
        run: devreplay ./ .devreplay.json
```

は *editor*s に一致しない)

- 大文字小文字 (`matchCase`): 大文字小文字情報を考慮する．(例: *editor* は **E***ditor* に一致しない)

- `after` 関連オプション (真偽型)
  - 大文字小文字保管 (`preserveCase`): 置換時に大文字と小文字の状態を残す

- 出力表示関連オプション (文字列型)
  - メッセージ (`message`): VS Code 上や CLI 上で表示する警告メッセージを変更する．標準では “*before should be after*” もしくは “*before should be fixed*” (`after` プロパティを定義していない場合)
  - 重要度 (`severity`): 規約の重要度．表 10 に示すとおり，重要度を変更することで規約の機能を拡張もしくは削減することができる．また，VS Code 上のコマンド “*Disable Rule*”, “*Upgrade Rule Severity*”, “*Downgrade Rule Severity*” を用いることで重要度を更新することができる (図 7) ．標準では `warning` として処理している．

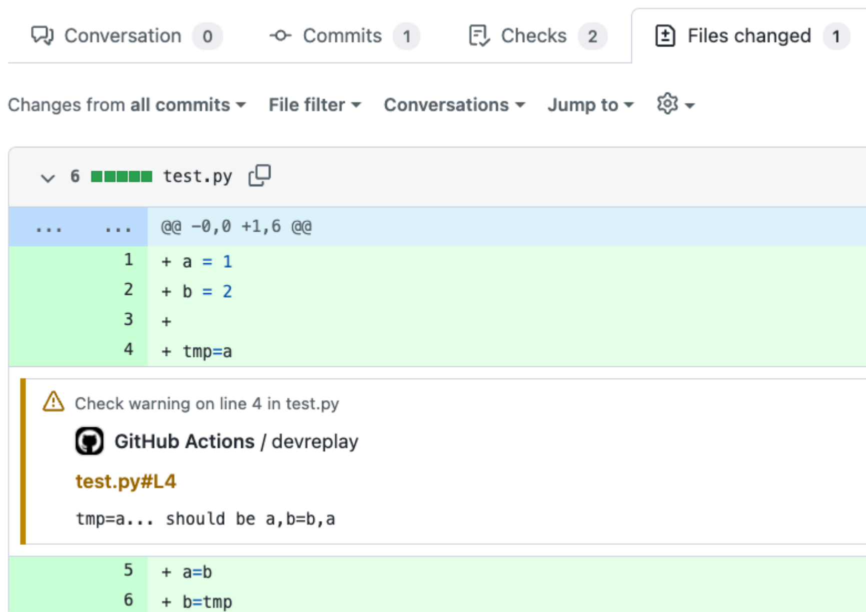

図 8: プルリクエストにおける GitHub action 出力例

- エディタ表示関連オプション (真偽型)
  - 不要なコード (`unnecessary`): 標準メッセージを “*before is unused*” に変更する．また，VS Code 上で警告時に表示される波線を文字の透過に変更する．
  - 非推奨なコード (`deprecated`): 標準メッセージを “*before is deprecated*” に変更する．また，VS Code 上で警告時に表示される波線を取り消し線に変更する．

利用者が対象プロジェクト内に規約ファイルを作成していない場合や CLI の引数にファイルを含めていない場合，DevReplay は既存ツールから引用した組み込みの規約を利用する．このとき，DevReplay は対象ファイルの拡張子から適用する規約を選択する．また，利用者は規約ファイルに規約の代わりにサポートしているプログラミング言語やフレームワーク名を記述することで，その規約を利用することができる（例： `"python"`）

表 10: 規約重要度と DevReplay 機能の対応

機能	error	warning	info	hint	off
ソースコード修正	x	x	x	x	
VS Code 上での警告波線	赤	黄	青		
GitHub Actions 上でのコメント	x	x	x		
GitHub Actions 上での警告	x	x			
VS Code 上でのビルド時警告	x				

5.4.2 正規表現規約の自動生成

規約を自動生成することで開発者の経験と合致したコーディング規約を生成することができる．図 9 に下記の規約セット生成プロセスを示す．

1. 最新バージョンのソースコードから変更されたコード片を `git diff` コマンドを用いて抽出する．
2. プログラム解析ツール tree-sitter を用いて最小変更抽象構文木を抽出する．
3. 変更前，変更後の抽象構文木から共通する識別子を抽出する．図では変数 t を抽出している．
4. 抽象構文木を以下のプロセスによって正規表現に変換する．
   - `before` プロパティに対して，共通する識別子をタグ付けされた任意の文字列を表す正規表現に変換する． 図 9 では変数 t を正規表現 $(< t > \backslash w+)$ に変換している． `after` プロパティに対して，識別子をドル記号付きのインデックスに変換する．
   - 空白を正規表現 $\backslash s*$ に変換する

以上の処理の後，DevReplay は正規表現を利用した規約セットファイル “`.devreplay.json`” を生成する．このときバックスラッシュ (\) は文字列処理のために \\ となる．

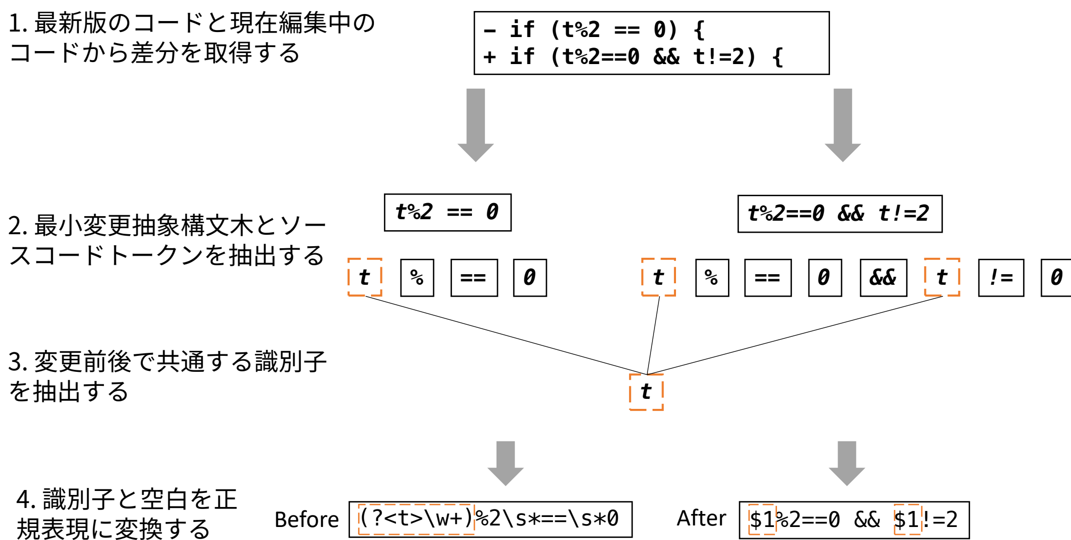

図 9: 正規表現コーディング規約セット生成プロセス

5.5 ケーススタディ

DevReplay の有用性を定量評価するために C 言語のベンチマークセット 2 件を用いて，不具合修正の精度を評価する．具体的には以下の調査課題に基づき評価する．

1. RQ3-1: 抽出した規約セットにより既存のコード修正ツールよりも多くの不具合を修正可能になるか？
2. RQ3-2: どのような不具合に対して抽出した規約は効果的か？

また，提案手法が実開発で利用可能であるか調査するために，次の調査課題を用いて評価する．
RQ3-3: 抽出した規約により開発者に採用される修正提案が可能になるか？

5.5.1 データセット

RQ3-1，RQ3-2 データセット： RQ3-1 と RQ3-2 では C 言語の不具合ベンチマークセット Codeflaws [78] と IntroClass [79] を用いる．また，DevReplay の

Listing 8: Codeflaws データセットにおけるコード修正例

```
// 比較演算子の入れ替え (ORRN)
- if (sum > n)
+ if (sum >= n)

// 真偽演算子の入れ替え (OLLN)
- if ((s[i] == '4') && (s[i] == '7'))
+ if ((s[i] == '4') || (s[i] == '7'))

// 条件式の追加削除 (OILN)
- if (t%2 == 0)
+ if (t%2 == 0 && t != 2)
```

精度を 4 種類の不具合修正ツール [80, 81, 82, 83] と比較する．これらのツールの精度評価は既存研究のものを利用する [84].

対象とするベンチマークデータセットはいずれも単一ファイルを対象としたものとなる．Codeflaws はプログラミングコンテストプラットフォームである Codeforces への投稿から 3,902 件の不具合 を 6 種類にカテゴリ分けしている．本分析ではこの内，既存研究で利用された 665 件の不具合を対象とする [84]. これらの不具合は 3 種類にカテゴリ分けされており “比較演算子の入れ替え” (ORRN)，“真偽演算子の入れ替え” (OLLN)，“条件式の追加削除” (OILN) となる．Listing 8 に各カテゴリの修正例を示す．

Introclass は学生の宿題によって作成された 6 種類のプログラムを扱っている．それぞれの課題は独立した自作の blackbox テストとテスト生成ツール KLEE [85] によって自動生成された whitebox テストによって確認されている．以下に各課題の概要を示す

- smallest: 4 つの値から最小値を求める
- median: 3 つの値から中央値を求める
- digits: 与えられた値の桁数を求める

表 11: RQ3-3: 対象プロジェクト及び規約セット生成対象期間

プロジェクト	言語	変更収集期間	RQ3-3 結果
VS Code	TypeScript	2020/01/01–2020/01/07	採用 [87]
			不採用 [88]
Flutter	Dart	2020/01/28–2020/02/04	不採用 [89]
React Native	JavaScript/Java	2020/01/17–2020/01/24	採用 [90]
Kubernetes	Go	2020/01/29–2020/02/05	採用 [91]
TensorFlow	C++/Python	2020/01/01–2020/01/07	採用 [92]
DefinitelyTyped	TypeScript	2020/01/01–2020/01/07	採用 [93]
Ansible	Python	2020/01/01–2020/01/07	採用 [94]
			採用 [95]
Home Assistant	Python	2020/02/06–2020/02/13	採用 [96]

- checksum: 文字列から特定の文字を数える
- grade: 与えられた値から学生の学年を求める

既存研究では “grade” プログラムを評価していないが，本論文ではDevReplay の汎化性能を評価するために対象とする．

RQ3-3 データセット: 表 11 に対象プロジェクトを示す．本プロジェクトでは最も人気のあるプロジェクト 10 件のうち，ソースコードを管理している 8 プロジェクトを対象とする [86]．プロジェクト参加者にとって記憶に新しい提案を行うために，本実験では一週間以内のソースコード変更内容から規約セットを生成し，そのなかで最も繰り返されている行われている変更を利用した．

5.5.2 RQ3-1 および RQ3-2 評価手法

時系列にデータを並べることができる Codeflaws について，前回までのコンテストデータを規約化したものは次回以降のコンテストでも再利用可能であると仮説を立てた．

次に，時系列データがない IntroClass について，2 つの利用ケースに基づいた仮説を立てた．1: 他の講義で学習した知識をパターン化したものは現在の課題で再利用可能である．2: 同じ講義を受けている学生の誤りのパターンは自分の課題にも適用可能である．

2 つのデータセットに対して，生成したコード修正提案を以下の 4 段階で評価した．

Same: 修正提案の中に実際に人間の開発者が直した修正と同一者が存在する．このとき空行や空白のみの違いもここに加える．**Success**: Same には該当しないものの与えられたテストケースにすべて成功しているものが修正提案の中に存在する．**No suggestion**: 生成した修正提案の中にコンパイルに成功したものが 1 件もない．**Test failed**: 上記 3 つに当てはまらない場合．つまりコンパイルに成功した修正提案があり，それらはすべてのテストケースには通らなかった．このとき， *Same* と *Success* に該当する課題を成功，*Test Failed* と *No suggestion* に該当するものを失敗とする．

CodeFlaws について，修正対象とするプログラミングコンテスト以前のコンテストからコーディング規約セットを生成する．そのため制限として，初期のコンテストに対しては規約セットを生成できないため修正は行えない．

IntroClass について，3 つの検証手法を利用する．(1) Leave-one-out（LOO）相互検証手法 [29] を使用して，同じ課題の別の投稿から規約セットを生成する．このときの規約セットを **LOO 規約**とする．LOO 規約を用いて，DevReplay の特化性能を評価する．(2) 対象でない 5 つの課題のプログラムから規約を生成する．この規約を **5 program 規約**とする．5 program 規約を用いて，ツールの汎化性能を評価する．(3) LOO と 5 program 規約セットの両方を使用して，手法の重複を評価する．

5.5.3 RQ3-1: 抽出した規約セットにより既存のコード修正ツールよりも多くの不具合を修正可能になるか？

[80, 81, 82, 83] まず，Codeflaws データセットについて，表 12 に既存の不具合修正ツールと比較した修正提案件数および修正成功件数を示す．このとき，必

表 12: RQ3-1: Codeflaw における既存ツールおよび提案手法の不具合修正精度および件数

ツール	Same/Success	Test failed/No suggestion
Angelix [80]	81 (12.4%)	570
CVC4 [81]	91 (14.0%)	560
Enum [82]	92 (14.1%)	559
Semfix [83]	56 (8.6%)	595
DevReplay (1 規約)	55 (8.0%)	596
DevReplay (3 規約)	90 (13.4%)	561
DevReplay (5 規約)	**101 (14.9%)**	550
DevReplay (全規約)	**136 (20.9%)**	515

要とする変更履歴データの量を評価するため，DevReplay で利用する規約の件数を複数パターン比較した．実験結果より，DevReplay は規約 5 件分のデータにより，既存ツールよりも多くの不具合修正が可能であることを確認した．

次に IntroClass について，表 13 に各課題に対する不具合修正件数を示す．対象としたデータセットのバージョンおよび不具合件数が異なるため，ここでは修正率で比較する．既存ツールよりも多くの修正を行えた例として *syllables* and *grades* がある．これらは文字列型のフォーマットに関する不具合が複数件あり，共通点を探す DevReplay で修正を行うことができた．修正が行えない例として，数値計算を行う *digit* が挙げられる．原因として，数値計算の内容は DevReplay では正規表現に変換が行えない点が考えられる．

これらの結果より，構文に関わるような初心者が陥りやすい典型的なミスには DevReplay は機能する．逆に動的なチェックが必要となる不具合に対しては既存ツールを利用することが推奨される．

RQ3-1 に対する答えは以下の通りである．

RQ3-1: 抽出した規約により既存のコード修正ツールよりも多くの不具合を修正可能になるか？: 提案手法 DevReplay は類似した修正の多い Codeflow(プログラミングコンテスト) の不具合を既存ツールよりも高い精度で修正した．同様に，

表 13: RQ3-1: IntroClass におけるブラックテスト不具合への修正精度

ブラックテスト	same/success 件数 (same 件数)						
プログラム	#	DevReplay	#	Angelix	CVC4	Enum	Semfix
smallest	68	8 (0)	56	37	39	29	45
		11.8%		66.1%	69.6%	51.8%	80.4%
median	64	1 (0)	54	38	28	27	44
		1.6%		70.4%	51.9%	50.0%	81.5%
digits	59	0 (0)	57	6	4	3	10
		0.00%		10.5%	7.0%	5.3%	17.5%
syllables	39	18 (1)	39	0	0	0	0
		46.1%		0.0%	0.0%	0.0%	0.0%
checksum	18	0 (0)	19	0	0	0	0
		0.00%		0.0%	0.0%	0.0%	0.0%
grade	96	14 (9)	–	–	–	–	–
		14.6%		–	–		
合計	344	41 (10)	225	81	71	59	99
		11.9%		36.0%	31.6%	26.2%	44.0%

IntroClass(プログラミング授業課題) に含まれる，文字列の修正も既存ツールでは修正できていない種類の不具合修正を行った．ただし，同データセットに含まれる数値計算の不具合の修正は行えない．そのため，既存手法との併用が推奨される．

5.5.4 RQ3-2: どのような不具合に対して抽出した規約は効果的か？

RQ3-2 について，表 15 に修正した不具合，Listing 8 に修正例を示す．DevReplay で修正可能な不具合の種類には偏りがある．修正可能な例として，ORRN (比較演算子の修正) の 14.9% (46/308) 件が人間の修正内容と同様の提案を行えている．つまり，異なる課題であっても 14.9% のコードは同一の修正が適用可

表 14: RQ3-1: IntroClass におけるホワイトテスト不具合への修正精度

ホワイトテスト	same/success 件数 (same 件数)						
プログラム	#	DevReplay	#	Angelix	CVC4	Enum	Semfix
smallest	49	4 (0)	41	37	37	36	37
		8.2%		90.2%	90.2%	87.8%	90.2%
median	52	1 (0)	45	35	36	23	38
		1.9%		77.8%	80.0%	51.1%	84.4%
digits	94	0 (0)	90	5	2	2	8
		0.0%		5.6%	2.2%	2.2%	8.9%
syllables	46	23 (1)‘	42	0	0	0	0
		50.0%		0.0%	0.0%	0.0%	0.0%
checksum	30	0 (0)	31	0	0	0	0
		0.00%		0.0%	0.0%	0.0%	0.0%
grade	95	14 (9)	–	–	–	–	–
		14.7%		–	–	–	–
合計	366	42 (10)	249	77	75	61	83
		11.5%		30.9%	30.1%	24.5%	33.3%

能であることが確認された．逆に OLLN (論理演算子) の修正はほとんど行えておらず，その原因も *No suggestion* (修正提案がない) となっている．原因として，他 2 種類の不具合と異なり，規約セット生成に用いた変更件数が少なく，提案が行いにくい点が挙げられる．

表 16，表 17 に Leave-one-out (LOO) 規約，5 program 規約，両方を用いた規約セットを用いた際のブラックテストおよびホワイトテスト結果を示す．それぞれ LOO は *grade*，5 program は *syllables* に機能した．また，両方の規約セットを用いた際に精度が向上している，つまり修正内容には重複は殆どないことが確認できた．以上の結果より，可能な限り対象の修正履歴と関連するプロダクトの修正履歴両方を利用することが推奨される．

Table 18 に最も修正に寄与した規約とその頻度を示す．また，Listing 9 と List-

表 15: RQ3-2: Codeflaw における不具合種類ごとの修正件数 (n=651)

不具合	件数	Same	Success	Test failed	No suggestion
OILN	325	0 (0.0%)	52 (16.0%)	152	121
OLLN	18	0 (0.0%)	1 (5.6%)	6	11
ORRN	308	46 (14.9%)	37 (12.0%)	127	98
合計	651	46 (7.1%)	90 (13.8%)	285 (43.8%)	230(33.8%)
		136 (20.9%)		515 (79.1%)	

表 16: RQ3-2: IntroClass におけるブラックテスト不具合の修正精度

プログラム	#	Leave-one-out	5 programs	LOO+5 programs
smallest	68	6(0): 8.8%	2(0): 2.9%	8(0): 11.8%
median	64	1(0): 1.6%	1(0): 1.6%	1(0): 1.6%
digits	59	0(0): 0.0%	0(0): 0.0%	0(0): 0.0%
syllables	39	4(0): 10.3%	17(0):**43.6%**	18(1): 46.2%
checksum	18	0(0): 0.0%	0(0): 0.0%	0(0): 0.0%
grade	96	14(9):**14.6%**	0(0): 0.0%	14(9): 14.6%
合計	344	25(9): 7.3%	20(0): 5.8%	41(10): 11.9%

ing 9 に規約による修正内容の具体例を示す．DevReplay は文字列リテラル (`FixPrintString`) や API の利用方法に関する修正 (`AlternativeFunctionCall`) を頻繁に行っている．具体的には “`Stdent`” を “`Student`” に修正するような局所的かつ論理式に影響しない変更を確認した．この結果から，DevReplay は Null ポイントチェックに関する修正が最も多いとされた既存研究 [97] やそれに基づいた不具合修正ツールと修正対象が異なる．

RQ3-2 に対する答えは以下の通りである．

RQ3-2: どのような不具合に対して抽出した規約は効果的か?: 提案手法による，修正の精度は規約セット生成に利用したコード変更履歴の件数に依存する．効果的な例として，文字列リテラルや API の誤用に対して既存ツールよりも多くの不

表 17: RQ3-2: IntroClass におけるホワイトテスト不具合の修正精度

プログラム	#	Leave-one-out	5 programs	LOO+5 programs
smallest	49	4(0): 8.1%	1(0): 2.0%	4(0): 8.1%
median	52	1(0): 1.9%	0(0): 0.0%	1(0): 1.9%
digits	94	0(0): 0.0%	0(0): 0.0%	0(0): 0.0%
syllables	46	4(0): 8.7%	23(0): **50.0%**	23(1): 50.0%
checksum	30	0(0): 0.0%	0(0): 0.0%	0(0): 0.0%
grade	95	14(9):**14.7%**	0(0): 0.0%	14(9): 14.7%
合計	366	23(9): 6.3%	24(0): 6.6%	42(10): 11.5%

具合を修正した．逆に，論理演算子の修正のような類似した変更が少ない不具合は，規約セット生成が行えないため修正を提案できない．本提案手法は既存手法と組み合わせることで，より多くの不具合修正が期待できる．

5.5.5 RQ3-3: 抽出した規約により開発者に採用される修正提案が可能になるか?

自動生成した規約セットの有用性を評価するために，表 11 に示すオープンソースプロジェクトに対して評価を行った．本分析では，DevReplay をコード変更推薦ツールとして用いる．

規約を用いて作成した修正提案生成後，各プロジェクトのプルリクエスト投稿規約に従って手動でプルリクエストを投稿した．その結果，投稿した修正提案の 80 % (10 件中，8 件) がプロジェクトの開発者に採用されたことを確認した．Listing 11 に生成したパッチ例を示す．メモリの使用量を削減する汎用性の高い修正に加え，プロジェクト内の命名規則と統一する修正を行った．また，本評価で対象としたプロジェクトはいずれも既存の静的解析ツールを利用しているが，提案した修正内容は検出していないことを確認した．

不採用となった 2 件の変更提案の不採用理由を，VS Code プロジェクトと Fluter プロジェクトの開発者から受けた．まず，本実験の手法について，著者らはプロ

表 18: RQ3-2: IntroClass における有効な不具合修正を行った規約セットとその頻度

Leave one out パターン	ホワイト	ブラック	合計
FixPrintString	13	13	26
FixBoundary	6	11	17
AddSpace	3	1	4
その他	1	0	1
5 Program パターン	**ホワイト**	**ブラック**	**合計**
AlternativeFunctionCall	13	13	26
AddReturnParentheses	9	3	12
AddSpace	2	1	3
AddDefinition	0	2	2
FixBoundary	0	1	1

ジェクト内で変更を適用可能なファイル全てに対して適用し，提案を行った．これにより，検証コストが増大し，変更を適用できないと判断された．もう一つの不採用理由として，DevReplay の仕様上ソースコードの動作を考慮しない点がある．提案した変更の多くはプロジェクトで報告されている不具合と関連付けしていない．そのため，採用されたプルリクエストであっても多くは議論されずに採用が決まっている．プロジェクトの検証者は，これらの変更がプロジェクトに影響を及ぼさない場合，不採択としている．これらの問題を回避するために，本ツールを利用する場合は，プロジェクト内部の開発者が規約セットを決定することが推奨される．

RQ3-3 に対する答えは以下の通りである．

RQ3-3: 抽出した規約セットにより開発者に採用される修正提案が可能になるか?: 提案手法 DevReplay により投稿した修正提案のうち 80%がプロジェクトの開発者に採用された．また，いずれの修正も既存の静的解析ツールで検出ができていないことを確認した．

Listing 9: RQ3-2:IntroClass における Leave One Out 手法で利用したソースコード修正例

```
// FixPrintString
- printf(\"Student has an F grade\\n\");
+ printf(\"Student has failed the course\\n\");

// FixBoundary
- if ($0 > $1)
+ if ($0 >= $1)
// または
- char $0[21];
+ char $0[20];

// AddSpace
- int main(){
+ int main (){
```

5.6 妥当性への脅威

5.6.1 内部妥当性

RQ3-1 と RQ3-2 の評価では，DevReplay が想定していた異なるユースケースを持つ既存ツールと比較した．既存のツールはバグ修正コマンドを開発者が能動的に実行する必要があるのに対して，DevReplay はコードエディタ上で自動的に修正提案を行う．したがって，開発者の実際の作業時間は 2 つのツール間で異なる．この分析では，各ツールの推薦の精度のみを比較した．

RQ3-3 では，このしきい値は，既存の調査方法 [43] に基づき 1 週間のコミット履歴からコード変更データを収集した．既存調査と本章での評価は，毎日数十のコミットがある大規模なソフトウェアプロジェクトを想定している．DevReplay の実装では，規約セット生成の対象とする期間は開発者がプロジェクトに合わせて調整できる．

Listing 10: RQ3-2: IntroClass における 5 program 手法で利用したソースコード修正例

```
// OFFN (AlternativeFunctionCall)
- scanf(\"%s\", $0);
+ fgets($0, sizeof($0), stdin);
//or
+ fgets($0,256, stdin);

// AddReturnParentheses
- return $0;
+ return ($0);

// AddDefinition
- int $0, $1;
+ int $0, $1, x;
// または
- int $0;
+ int $0 = 0;
```

5.6.2 外部妥当性

DEVREPLAY は，1 つの連続した行の変更を対象とし，コード編集中に正規表現を用いて動作する．そのため，Defects4j [98] のような大規模な変更を収集したデータセットは，範囲外となる．また．既存の不具合修正ツールとは異なり，DEVREPLAY の推薦ではテストケースやコードの動作を考慮しない．同様に，複数のファイルの変更も範囲外となる．

5.7 結言

本研究ではコーディング規約セットの抽出を目標として，正規表現を用いた規約の生成と再利用を行う静的解析ツール DEVREPLAY の提案とその評価を行った．本ツールを利用することで，利用者は個人がもつ修正方法の知見を開発チー

Listing 11: RQ3-3: オープンソースプロジェクト開発者に採用されたソースコード変更内容

```
// TensorFlow プロジェクトに採用された変更 (C++)
// メモリ使用量を削減する
// https://github.com/tensorflow/tensorflow/pull/35600
- runner_ = [](std::function<void()> fn) { fn(); };
+ runner_ = [](const std::function<void()>& fn) { fn(); };

// Ansible プロジェクトに採用された変更 (C++)
// プロジェクトの命名規則を統一する
// https://github.com/ansible/ansible/pull/66201
- name: myTestNameTag
+ Name: myTestNameTag
```

ム全体で共有することができる．また，評価実験で，複数言語のオープンソースプロジェクトに本ツールで作成した変更提案を投稿し，80%が採用されたことを確認した．

今後の展望として，正規表現の生成において，重複や不要な規約を削減する方法を改良することを目標とする．また，ユーザーインターフェースも生成した規約の編集を容易にするよう改良する．最後に，評価方法について，本論文では短期的に生成した規約セットを評価したが，長期的にプロジェクトに導入した際に行える自動修正の量を評価する．

6. 結論

本論文ではコーディング規約の定型化を目指して，既存のコーディング規約の利用状況調査を行い，既存のコーディング規約改善およびコーディング規約の発見を行うアプローチを提案し，実験的な評価を行った。

まず，コードレビューにおいて，頻繁に行われる可読性改善の種類を調査した．実験により，既存の静的解析ツールでは検出が不可能な修正が頻繁に行われていることを確認した．次に，開発者が実際に従っているコーディング規約に基づき，静的解析ツールを自動設定する手法を提案した．実験により，開発者が手動で設定するよりも静的解析ツールの誤検出および検出漏れを削減できることを確認した．次に，開発者のソースコード編集履歴から静的解析ツールを用いたコーディング規約を生成し，ソースコードを自動修正する手法を提案した．実験によって，自動生成したコーディング規約によってソフトウェアプロジェクトに採用されるソースコード修正を提案可能であることを確認した．

以上の分析及び手法提案により，開発者はソースコードの記述と一致したコーディング規約セットを，継続的に運用することが可能になる．

6.1 課題と展望

既存のコーディング規約最適化に関して短期的な２つの課題がある．(1) 時系列での分布調査：本論文では一定期間でのコーディング規約の利用状況を調査したが，静的解析ツールが採用するコーディング規約は，言語の更新に追従している．広い範囲のプロジェクトにおいて，時系列ごとに規約の調査および推薦を行うことで規約の流行り廃りを確認でき，古い規約を用いているプロジェクトの特定につながる．例えばプログラミング言語において，同じ言語であっても異なるバージョンを利用している場合がある．コーディング規約と時刻情報の組を収集することで，バージョンをソースコードの内容のみで特定したり，新バージョンへの書き換えが容易になると期待する．(2) 多様かつ複数の解析ツールの同時評価：本論文では言語ごとに単一の静的解析ツールの特定バージョンを用いて，分析及び推薦手法の提案を行った．実際の開発環境では目的に合わせて複数のツー

ルを並行して利用する．例えば，本論文が対象とした静的解析ツール Pylint [15] と flake8 [67] の規約は一部重複し，かつツールバージョンによって規約は増減する．規約の重複や過不足を削減するために，SARIF (Static Analysis Results Interchange Format) に代表される統一規格を用いて，複数ツールを同時に評価する手法が必要になると考える [99].

中期的な課題として，モジュールや開発者に特化した既存コーディング規約の選定がある．4 章ではコーディング規約の最適化手法の課題として，プロジェクトの全ファイルが従っている規約を推薦した．そのため，1 つでも規約に従っていないファイルがあった場合や，特定の開発者や一定期間のみが従っていない規約も有用でないと判断している．また，本論文では規約の利用状況をプロジェクトごとに分析したが，別の分析で規約違反はプロジェクト内で特定の開発者が繰り返し行っていることを確認している．モジュールや，開発者，期間ごとに規約の分布を調査することで，規約をより厳密に推薦し，コードの一貫性を高く保つことができると期待する．

長期的な課題としては，コーディング規約の評価に関する研究がある．本論文では，開発者が規約に基づく修正を受け入れるか否かで規約を評価した．現在利用されているコーディング規約は多岐にわたり，本研究手法を用いて選定したとしても，開発者は規約の有用性を検証することができない．将来的には新たなコストを開発者に課さないように，開発者に提示可能な規約の評価指標が必要である．

謝辞

本論文の執筆を進めるに当たり，多くの方々にご指導，ご協力を賜りました．ここにお名前を記させていただくと共に，深く感謝の意を示します．

主指導教員であり，本論文の主審査委員を努めて頂いた，松本 健一教授には，研究の進め方をはじめとして，多くのご指導・ご助言を賜りました．深く感謝いたします．

本稿の副指導教員，本論文の副審査委員を努めて頂いた，笠原 正治教授には，学内発表において研究の質を高める上で非常に貴重なご意見を頂きました．深く感謝いたします．

本稿の副指導教員，本論文の副審査委員を努めて頂いた，石尾 隆准教授には，論文の執筆や研究の方針をはじめとして常に適切な指導を賜り，多くのご助言を賜りました．深く感謝いたします．

本稿の副指導教員，本論文の副審査委員を努めて頂いた，Raula Gaikovina Kula 助教には，関連する最新の研究動向についてご指導いただきました．深く感謝いたします．

本論文の副審査委員を努めて頂いた，和歌山大学 ソーシャルソフトウェア工学研究室 伊原 彰紀講師には，入学前から，研究の方針や取り組み方についてご指導・ご助言していただきました．深く感謝いたします．

特に第 5 章の内容に関して，ミーティングなどを通じて有益なご意見をいただきました Square 松本 宗太郎氏，株式会社 Sider 角 幸一郎氏，株式会社リクルート 竹迫 良範氏に深く感謝いたします．

株式会社技術評論社 稲尾 尚徳氏には，技術文章について執筆の進め方から細かな校正方法まで，ご指導いただきました．深く感謝いたします．

奈良先端科学技術大学院大学 秘書 高岸 詔子氏には，本研究の遂行に当たり，さまざまなご配慮とご協力をいただきました．深く感謝いたします．

VS Code Meetup の方々，奈良先端科学技術大学院大学 先端科学研究科 ソフトウェア工学研究室の方々には，ミーティングなどを通じて様々なご助言をいただきました．特に幾谷 吉晴氏，中才 恵太郎氏には研究に限らず多くのご助言を賜りました．深く感謝いたします．

参考文献

[1] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. Code reviews do not find bugs: How the current code review best practice slows us down. In *Proc. the 37th International Conference on Software Engineering-Volume (ICSE'15)*, pp. 27–28, 2015.

[2] Peter C. Rigby and Margaret-Anne Storey. Understanding broadcast based peer review on open source software projects. In *Proc. the 33th International Conference on Software Engineering (ICSE'11)*, pp. 541–550, 2011.

[3] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proc. the 35th International Conference on Software Engineering (ICSE'13)*, pp. 712–721, 2013.

[4] Rodrigo Magalhães dos Santos and Marco Aurélio Gerosa. Impacts of coding practices on readability. In *Proc. the 26th Conference on Program Comprehension (ICPC'18)*, pp. 277–285, 2018.

[5] Lisa Nguyen Quang Do, James Wright, and Karim Ali. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering*, 2020.

[6] Ferdian Thung, David Lo, Lingxiao Jiang, Foyzur Rahman, Premkumar T Devanbu, et al. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In *Proc. the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*, pp. 50–59, 2012.

[7] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto. Are static analysis violations really fixed? a closer look at realistic usage of SonarQube. In *Proc. the 27th International Conference on Program Comprehension (ICPC'19)*, pp. 209–219, 2019.

[8] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Would static analysis tools help developers with code reviews? In *Proc. the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER'15)*, pp. 161–170, 2015.

[9] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C Gall. Context is king: The developer perspective on the usage of static analysis tools. In *Proc. the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER'18)*, pp. 38–49, 2018.

[10] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Shin Yoo, and Yves Le Traon. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*, 2018.

[11] Yida Tao, Donggyun Han, and Sunghun Kim. Writing acceptable patches: An empirical study of open source project patches. In *Proc. the 30th International Conference on Software Maintenance and Evolution (ICSME'14)*, pp. 271–280, 2014.

[12] Guido van Rossum, Barry Warsaw, and Nick Coghlan. Pep 8: style guide for python code. *Python. org*, 2001.

[13] Robert C Seacord. *The CERT C secure coding standard*. Pearson Education, 2008.

[14] Oliver Burn. Checkstyle. *https://checkstyle.org/*, 2021.

[15] Pylint. https://pypi.org/project/pylint/, 2020.

[16] StyleCop. https://github.com/stylecop/stylecop, 2020.

[17] robocop. https://docs.rubocop.org/rubocop/0.88/index.html, 2020.

[18] Nicholas C. Zakas, Brandon Mills, Toru Nagashima, and Milos Djermanovic. ESLint - Pluggable JavaScript linter. `https://eslint.org/`, 2020.

[19] Google JavaScript Style Guide. https://google.github.io/styleguide/jsguide.html, 2020.

[20] JavaScript standard style rules. https://standardjs.com/rules.html, 2020.

[21] Travis ci. `https://travis-ci.org/`. Accessed: 2021-11-22.

[22] Circleci. `https://circleci.com/`. Accessed: 2021-11-22.

[23] Tim Buckers, Clinton Cao, Michiel Doesburg, Boning Gong, Sunwei Wang, Moritz Beller, and Andy Zaidman. UAV: Warnings from multiple automated static analysis tools at a glance. In *Proc. the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER'17)*, pp. 472–476, 2017.

[24] Antônio Carvalho, Welder Luz, Diego Marcílio, Rodrigo Bonifácio, Gustavo Pinto, and Edna Dias Canedo. C-3PR: A bot for fixing static analysis violations via pull requests. In *Proc. the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER'20)*, pp. 161–171, 2020.

[25] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *Proc. the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, Vol. 1, pp. 470–481, 2016.

[26] Eliane S Wiese, Anna N Rafferty, Daniel M Kopta, and Jacqulyn M Anderson. Replicating novices' struggles with coding style. In *Proc. The 27th International Conference on Program Comprehension (ICPC'19)*, pp. 13–18, 2019.

[27] Ahmed E Hassan. Predicting faults using the complexity of code changes. In *Proc. The 31st international conference on software engineering (ICSE'09)*, pp. 78–88, 2009.

[28] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, Vol. 51, No. 4, pp. 1–37, 2018.

[29] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proc. the 22th International Symposium on Foundations of Software Engineering (FSE'14)*, pp. 281–293, 2014.

[30] Cathal Boogerd and Leon Moonen. Assessing the value of coding standards: An empirical study. In *Proc. the 24th IEEE International Conference on Software Maintenance (ICSM'08)*, pp. 277–286, 2008.

[31] Michael Smit, Barry Gergel, H James Hoover, and Eleni Stroulia. Code convention adherence in evolving software. In *Proc. the 27th IEEE International Conference on Software Maintenance (ICSM'11)*, pp. 504–507, 2011.

[32] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proc. the 11th Working Conference on Mining Software Repositories (MSR'14)*, pp. 202–211, 2014.

[33] Gerrit code review. `https://www.gerritcodereview.com/`. Accessed: 2019-12-17.

[34] Review rboard. `https://www.reviewboard.org`. Accessed: 2019-12-17.

[35] Peter C. Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proc. the 9th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pp. 202–212, 2013.

[36] Jason Tsay, Laura Dabbish, and James Herbsleb. Let ' s talk about it: Evaluating contributions through discussion in github. In *Proc. the 22th International Symposium on Foundations of Software Engineering (FSE'14)*, pp. 144–154, 2014.

[37] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proc. the 11th Working Conference on Mining Software Repositories (MSR'14)*, pp. 192–201, 2014.

[38] Quinn Hanam, Fernando S de M Brito, and Ali Mesbah. Discovering bug patterns in javascript. In *Proc. the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*, pp. 144–156, 2016.

[39] Miltiadis Allamanis, Earl T Barr, Christian Bird, Premkumar Devanbu, Mark Marron, and Charles Sutton. Mining semantic loop idioms. *IEEE Transactions on Software Engineering*, Vol. 44, pp. 651–668, 2018.

[40] Antoine Barbez, Foutse Khomh, and Yann-Gaël Guéhéneuc. Deep learning anti-patterns from code metrics history. *arXiv preprint arXiv:1910.07658*, 2019.

[41] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering*, Vol. 33, No. 11, 2007.

[42] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proc. the 36th International Conference on Software Engineering (ICSE'14)*, pp. 803–813, 2014.

[43] Hoan Anh Nguyen, Tien N Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *Proc. the 41st International Conference on Software Engineering (ICSE'19)*, pp. 819–830, 2019.

[44] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proc. the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*, pp. 727–739, 2017.

[45] Reudismam Sousa, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. Learning quick fixes from code repositories. In *Proc. the Brazilian Symposium on Software Engineering (SBES'21)*, pp. 74–83, 2021.

[46] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *arXiv preprint arXiv:1810.01791*, 2018.

[47] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. Elixir: effective object oriented program repair. In *Proc. the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 648–659, 2017.

[48] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. Api code recommendation using statistical learning from fine-grained changes. In *Proc. the 24th International Symposium on Foundations of Software Engineering (FSE'16)*, pp. 511–522, 2016.

[49] Na Meng, Miryung Kim, and Kathryn S McKinley. Sydit: creating and applying a program transformation from an example. In *Proc. the 19th Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11)*, pp. 440–443, 2011.

[50] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *Proc. the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, Vol. 1, pp. 213–224, 2016.

[51] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *Proc. the 41st International Conference on Software Engineering (ICSE'19)*, pp. 783–794, 2019.

[52] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proc. the 35th International Conference on Software Engineering (ICSE'13)*, pp. 802–811, 2013.

[53] Tim Molderez, Reinout Stevens, and Coen De Roover. Mining change histories for unknown systematic edits. In *Proc. the 14th International Conference on Mining Software Repositories (MSR'17)*, pp. 248–256, 2017.

[54] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*, pp. 1–12, 2019.

[55] Rohan Bavishi, Hiroaki Yoshida, and Mukul R Prasad. Phoenix: automated data-driven synthesis of repairs for static analysis violations. In *Proc. the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*, pp. 613–624, 2019.

[56] Amiangshu Bosu and Jeffrey C Carver. Impact of developer reputation on code review outcomes in oss projects: An empirical investigation. In *Proc. the 8th International Symposium on Empirical Software Engineering and Measurement (ESEM'14)*, pp. 1–10, 2014.

[57] Amiangshu Bosu and Jeffrey C Carver. Impact of peer code review on peer impression formation: A survey. In *Proc. the 7th International Symposium on Empirical Software Engineering and Measurement (ESEM'13)*, pp. 133–142, 2013.

[58] Xin Yang, Raula Gaikovina Kula, Norihiro Yoshida, and Hajimu Iida. Mining the modern code review repositories: A dataset of people, process and product. In *Proc. the 13th Working Conference on Mining Software Repositories (MSR'16)*, pp. 460–463, 2016.

[59] Pep 8 – style guide for python code. `https://www.python.org/dev/peps/pep-0008`. Accessed: 2019-12-17.

[60] Gerrit code review - rest api. `https://gerrit-review.googlesource.com/Documentation/rest-api.html`. Accessed: 2019-12-17.

[61] Github api v3. `https://developer.github.com/v3/`. Accessed: 2019-12-17.

[62] Kai Pan, Sunghun Kim, and E. James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, Vol. 14, No. 3, pp. 286–315, 2009.

[63] Google python style guide. `https://github.com/google/styleguide/blob/gh-pages/pyguide.md`. Accessed: 2019-12-17.

[64] Openstack style guidelines. `https://docs.openstack.org/hacking/latest/user/hacking.html`. Accessed: 2019-12-17.

[65] F405 への違反に関する例外. `https://docs.openstack.org/hacking/latest/user/hacking.html#imports`. Accessed: 2019-12-17.

[66] Sun Developer Network. Code conventions for the java programming language, 1999.

[67] flake8. https://pypi.org/project/flake8/, 2019.

[68] pydocstyle. https://pypi.org/project/pydocstyle/, 2020.

[69] Ecma International. Standard ECMA-262. https://www.ecma-international.org/publications/standards/Ecma-262.htm, 2020.

[70] Ecma International. ECMAScript Language Specification, 5.1 edition. https://www.ecma-international.org/ecma-262/5.1/, 2016.

[71] David Kavaler, Asher Trockman, Bogdan Vasilescu, and Vladimir Filkov. Tool choice matters: JavaScript quality assurance tools and usage outcomes in GitHub projects. In *Proc. the 41st International Conference on Software Engineering (ICSE'19)*, pp. 476–487, 2019.

[72] Pmd. https://pmd.github.io/. Accessed: 2021-10-25.

[73] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. Reversible debugging software "quantify the time and cost saved using reversible debuggers". 2013.

[74] Undo Software. Increasing software development productivity with reversible debugging. 2014.

[75] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proc. the 35th International Conference on Software Engineering (ICSE'13)*, pp. 672–681, 2013.

[76] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, Vol. 61, No. 4, pp. 58–66, 2018.

[77] Yoshiki Higo and Shinji Kusumoto. How often do unintended inconsistencies happen? deriving modification patterns and detecting overlooked code fragments. In *Proc. the 28th IEEE International Conference on Software Maintenance (ICSM'12)*, pp. 222–231. IEEE, 2012.

[78] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *Proc. the 39th International Conference on Software Engineering Companion (ICSE'17)*, pp. 180–182, 2017.

[79] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, Vol. 41, No. 12, pp. 1236–1256, 2015.

[80] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proc. the 38th International Conference on Software Engineering*, pp. 691–701, 2016.

[81] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. Counterexample-guided quantifier instantiation for synthesis in smt. In *International Conference on Computer Aided Verification*, pp. 198–216. Springer, 2015.

[82] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013.

[83] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proc. the 35th International Conference on Software Engineering (ICSE'13)*.

[84] Jifeng Xuan, He Jiang, Zhilei Ren, and Weiqin Zou. Developer prioritization

in bug repositories. In *Proc. the 34th International Conference on Software Engineering (ICSE'12)*, pp. 25–35, 2012.

[85] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. the 8th USENIX Symposium on Operating Systems Design and Implementation*, Vol. 8, pp. 209–224, 2008.

[86] The-State-of-the-Octoverse. https://octoverse.github.com, 2019.

[87] DevReplay's Pull Request for Visual Studio Code #87709. `https://github.com/microsoft/vscode/pull/87709`. Accessed: 2021-10-23.

[88] DevReplay's Pull Request for Visual Studio Code #88117. `https://github.com/microsoft/vscode/pull/88117`. Accessed: 2021-10-23.

[89] DevReplay's Pull Request for Flutter #50089. `https://github.com/flutter/flutter/pull/50089`. Accessed: 2021-10-23.

[90] DevReplay's Pull Request for React Native #87709. `https://github.com/facebook/react-native/pull/27850`. Accessed: 2021-10-23.

[91] DevReplay's Pull Request for Kubernetes #87838. `https://github.com/kubernetes/kubernetes/pull/87838`. Accessed: 2021-10-23.

[92] DevReplay's Pull Request for TensorFlow #35600. `https://github.com/tensorflow/tensorflow/pull/35600`. Accessed: 2021-10-23.

[93] DevReplay's Pull Request for DefinitelyTyped #41434. `https://github.com/DefinitelyTyped/DefinitelyTyped/pull/41434`. Accessed: 2021-10-23.

[94] DevReplay's Pull Request for Ansible #66201. `https://github.com/ansible/ansible/pull/66201`. Accessed: 2021-10-23.

[95] DevReplay's Pull Request for Ansible #66203. https://github.com/ansible/ansible/pull/66203. Accessed: 2021-10-23.

[96] DevReplay's Pull Request for Home Assistant #31783. https://github.com/home-assistant/home-assistant/pull/31783. Accessed: 2021-10-23.

[97] Martin Monperrus. A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *Proc. the 36th International Conference on Software Engineering (ICSE'14)*, pp. 234–242, 2014.

[98] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proc. the International Symposium on Software Testing and Analysis (ISSTA'14)*, pp. 437–440, 2014.

[99] Static Analysis Results Interchange Format (SARIF) Version 2.1.0. http://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html. Accessed: 2021-10-23.