Vol.2020-SE-206 No.1
2020/11/27

静的解析ツールの誤検出および検出漏れの最小化支援

上田 裕己[a)]　石尾 隆[b)]　松本 健一[c)]

概要：静的解析ツールはプログラミング中に頻出するミスや悪習を自動検出する．静的解析ツールは企業及びオープンソースともに広く普及している一方で，検出した警告のうち 90%以上がが開発者に修正されない誤検出率の高さが指摘されている．原因の一つとして，静的解析ツールは開発者によって検出する警告を設定することを前提に設計されているにもかかわらず，開発者の多くは設定しない，もしくはツール導入時点から設定を変更しない点が指摘されている．本稿では静的解析ツールをソフトウェアプロジェクトのソースコードに合わせて自動設定する手法を提案する．本手法では，静的解析ツールが一度でも検出した警告を無効，そうでない設定を有効に自動設定する．4 件の JavaScript プロジェクトを対象にソフトウェアアップデートごとの静的解析ツールの誤検出を提案手法利用時とそうでない時で計測した．提案手法を利用することにより，開発者が将来修正するルール違反の 30%を早期検出した．

キーワード：静的解析ツール, コーディングスタイル, コーディングルール

1. はじめに

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 (Automated Static Analysis Tool，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ルール（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ルール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ルールの有効，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 90%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 [1,2]．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ルールを編集しないこと，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ルール違反も余分に検出していることが挙げられている [3]．

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 [4]．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 [5]．これらはコーディングルールの理解や修正作業の削減につながるが，誤検出自体を減らす仕組みではない．

また，コーディングルールを生成する技術として，空白や文字数などのスタイルを管理するコーディングルールを推測する手法も提案されている [6, 7]．本稿では，ソースコードのスタイルだけでなく，プログラミング言語特有の機能に関わるコーディングルールも対象とする ASAT を用いてコーディングルールを推測する．

本稿では ASAT のルール精度の改善を目標とし，以下の 3 つの仮説のもと ASAT のルール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ルール集合や開発者が手動で設定したルール集合と，開発者が実際に準拠しているルールには齟齬があり，そのためにルール違反の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ルールはその後のバージョンでも準拠され続ける（ルールは変化し

a) ueda.yuki.un7@is.naist.jp
b) ishio@is.naist.jp
c) matumoto@is.naist.jp

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ルールを準拠する，というような局所的なルールは存在しない．

これらの仮説に基づき，本稿では，ASAT が検査可能なすべてのコーディングルールのうち，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）ルールをすべて有効化し，違反のあるルールを無効化するように，ASAT を自動設定する手法を提案する．

本稿の貢献は以下の 3 つである．

(1) ASAT の誤検出を削減する手法の提案 (3 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ルールの調査 (4.1 節)
(3) 提案手法で生成したルールと対象プロジェクト開発者が作成したルールを比較した誤検出の評価 (4.2 節)

調査の結果，プロジェクトが明示的に有効にしているルールであっても，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ルール数よりも，実際に準拠しているルールの数は 3.3 倍ほど存在することが判明した．提案手法を用いて自動的に ASAT を設定することで，開発者が将来修正するルール違反のうち最大 30%を早期検出することが可能になる．

2. 研究の動機

2.1 静的解析ツールの現状

多くの ASAT はコーディングルールというソースコードの記述方法に制限を設けるルールを参照して実装されている．通常コーディングルール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ルールや ASAT として，Java のコーディングルールである Sun [8] への違反を検出する *Checkstyle* [9]，Python のコーディングルール違反を検出する *Pylint* [10]，*Flake8* [11]，*pydocstyle* [12] 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能になる．本稿では JavaScript のコーディングルールである EcmaScript [13] に基づきソースコードを検証する ESLint [14] 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ルールのうち，広く利用可能なルールを標準ルールとして有効にしている．ESint の場合，241 のルールが検出可能なルール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 ルールの標準ルールのみを有効にしている．このルール集合は 2009 年時点の JavaScript 標準ルールである ECMAScript 5 [15] に違反するソースコード行を検出する．標準ルールである 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ルールを利用することで，IE ではサポートされていない ECMAScript 6 のルールを無効化したソースコード検証を可能にする．

Listing 1: no-template-curly-in-string ルール違反と違反修正例

```
// 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!
```

2.2 標準ルールの過不足例

事前分析で確認した ASATs 標準ルールの過不足例を 2 件示す．まず，ASATs の標準ルール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検出漏れルール例，2 つ目は，標準ルールで検出するものの多くのプロジェクトが準拠しない誤検出ルール例を示す．分析手法の詳細は RQ1 として 4.1 節に記述する．

まず，標準ルールに含まれていないが広く準拠されているルールの一つ *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 “`${variable}`” のような変数または式を含む文字列を作成することができる．ただし，ECMAScript 6（ES6）以降の JavaScript で，テンプレートリテラルを使用する場合，`"${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の検出するには，ESLint を設定しルールを有効にする必要がある．

次に，標準ルールに含まれているものの広く準拠されていないルールとして *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．本稿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ルール違反は修正されにくい．そのため，標準ルール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ルール違反に混入して，開発者にとって重要でないルール違反が検出されてしまう．

3. 提案手法

提案手法は，プロジェクトが利用している ASAT で利

Listing 2: no-empty ルール違反と違反修正例

```
// 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
}
```

用可能なルールの集合 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ルールの部分集合（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．

各ルール $r_i \in R_{All}$ 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）ルールとは $|r_i(S)| = 0$ であるようなルール，開発者が従っていない（非準拠である）ルール $|r_i(S)| > 0$ であるようなルール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 からルール $Config(S_j)$ 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ルール違反の行の集合を，以下のように表現する．

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

このルール違反の行の集合は，バージョン j の時点で準拠していたルール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ルール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 の標準ルールでは有効ではないルール集合を有効にするため，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ルールを無効化するため，新たな見逃しが発生する可能性もある．

本稿では，提案手法の具体的な実装として，2 種類のインタフェースを試作した．1 つ目は，開発者がコードレビューに投稿したソースコードの自動検証である．投稿されたソースコードの更新によって ASAT のルールに過不足が発生する場合，つまり $Config(S)$ とプロジェクトが現在設定しているルールが一致しない場合に，GitHub 上で開発者に通知する機能を実装した．本機能を利用することで，開発者はルール違反だけでなく，新たに準拠されるようになったルールをコードレビュー時点で確認し，プロジェクトのルールを更新することができる．もう 1 つは，コマンドラインインターフェースとして $Config(S)$ を反映した ESLint ルール設定ファイルを自動生成する機能である．本機能を利用することで，開発者はプロジェクト開始直後や ESLint 導入直後にプロジェクトに，必要なルールを過不足なく定義する事ができる．

以下に本手法を利用したルールの誤検出，検出漏れ通知および設定ファイル生成のワークフローを示す．

1. 利用者の作業
   1-1. ソースコードの変更を git を用いて記録する．
   1-2. ソースコード管理サービス GitHub に変更を反映させる．
2. ASAT の実行
   2-1. 変更されたソースコードファイル，または全てのソースコードファイルに対して ASAT を実行する．
   2-2. 検出されたコーディングルール違反を出力する．
3. 提案手法の実行
   3-1. プロジェクトが現在有効にしているルール集合と，変更後のコードから得られる $Config(S)$ を比較し，コーディングルール違反に対応しなかった場合に無効化されるコーディングルールおよび新たに有効にできるルールの候補を出力する．
   3-2. 新たに有効にできるルールを用いて ASAT 用の新しい設定ファイルを生成する．

4. 評価実験

提案手法を評価するために以下の 2 つのリサーチクエスチョンに回答する．

RQ_1（事前分析）：**開発者が設定しているルールと開発者が実際に準拠しているルールはどの程度異なるか？** 事前分析として，ASAT の標準ルールがどの程度の誤検出と検出漏れを有しているか，つまり開発者が実際に準拠しているルールとの差分を調査する．更に追加の調査として頻繁に誤検出および検出漏れに該当するルールを特定する．

RQ_2: **提案手法によりどの程度ルール違反の誤検出が削減可能か?** 本提案手法で生成したルールと ASAT 標準ルール，開発者が設定したルールを用いて，どの程度将来の誤検出および検出漏れが発生するか比較評価する．ここでは，プロジェクトのバージョンアップデートごとにルール

表 1: 対象プロジェクトおよびプロジェクトの各バージョンアップデート数

プロジェクト	Major	Minor	Maintenance	合計
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

を更新し，将来のバージョンアップデート時でのルールのPrecision(精度) および Recall(再現率) を評価する．

表 1 に調査対象プロジェクトと，プロジェクトのバージョン数を示す．これらのプロジェクトは，Bugs JS [16] で用いられている 10 件の JavaScript プロジェクトのうち，ESLint [14] を採用しているプロジェクト 4 件を抽出した．分析対象ソースコードとして 2020 年 7 月 1 日時点の 4 プロジェクトから，合計で 643 件のバージョンアップデートごとのソースコードを抽出した．更新の単位の大小による影響を評価するために，バージョン番号が “x.0.0” であるようなバージョン（17 件）のみを集めた Major アップデートの履歴，バージョン番号が “x.y.0” であるようなバージョン（197 件，1.0.0 のような Major アップデートを含む）のみを集めた Minor アップデートの履歴，バージョン番号が “x.y.z” の形で表現されるバージョン（561 件）を集めた Maintenance アップデートという 3 種類の履歴を用いる．対象ブランチとして main または master ブランチと呼ばれるプロジェクトの主要ブランチからソースコード変更履歴を抽出する．また，バージョン番号が “x.y.z-b” や “test-version” のように名称から順序が定められなかったものはこの履歴からは取り除いた．

ASAT としては JavaScript ソースコードを検証する ESLint [14] を用いる．このとき，プロジェクト間およびプロジェクトバージョン間でのコーディングルール数とルールの検出基準を統一するために，ESLint は 2020 年 7 月時点で最新版のバージョン 7.4.0 を用いる．

4.1 RQ_1（事前分析）：開発者が設定しているルールと開発者が実際に準拠しているルールはどの程度異なるか？

4.1.1 RQ_1 分析手法

事前分析として，対象プロジェクトで利用されているルールと標準ルール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ルール．すなわち，4 つのプロジェクトの $Config(S)$ の共通部分．

表 2: RQ_1: 対象プロジェクトで共通して準拠されているルール分布

分類	ルール数	標準ルール数
Followed	134	51
Unfollowed	82	3
Specific	25	3
合計	241	57

- Unfollowed: すべてのプロジェクトが違反しているルール．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ルール．
- Specific: プロジェクトによって準拠状態が異なるルール．すなわち，1 つから 3 つのプロジェクトの $Config(S)$ に含まれているルール．

また，各カテゴリのルールが，ASAT の標準設定でのルールに含まれているかを調査する．

4.1.2 RQ_1 結果

表 2 に結果を示す．**Followed ルール (134 件)**: まず，プロジェクトが実際に準拠しているルールについて 134 件のうち標準ルールに含まれていないルールを 83 件確認した．代表的なものとして 2.2 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．これらのルールに対する違反は 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 ルール (82 件): 次に，標準ルールに含まれているにも関わらず，分析対象プロジェクトの半数以上が準拠していないルールは以下の 3 件である，

- 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

これらは Followed ルールと異なり，動作に影響がないためソースコードの不具合には該当しない．

Specific ルール (25 件): 最後に対象プロジェクトによって準拠状況が異なるルールを 25 件発見した．そのうち標準ルールとして広く利用されているルールは以下の 3 件である，

- no-unused-labels: 参照されないラベルを宣言しない
- no-useless-escape: 文字列処理内で必要のないエスケープ文を使わない
- no-extra-semi: 不要なセミコロンを書かない

Unfollowed で上げたルールと同様にこれらも共通してソースコードの動作に影響を及ぼさないため，開発者には無視されている可能性がある．

RQ_1 (事前分析) 結果まとめ: 対象プロジェクトで共通

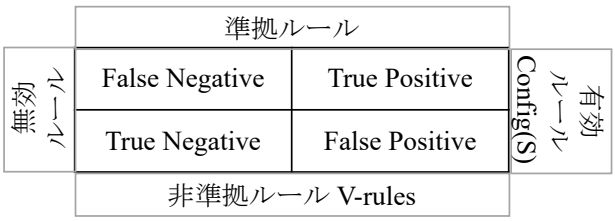

図 1: $RQ_{2.1}$: ルール設定およびプロジェクトに基づくルールの分類

して 134 のルールが準拠されているが，そのうち標準ルールに含まれていない，83 件のルールは ASAT の設定を編集しなければ検出できない．また，標準ルールに含まれるルールのうち 6 件は一つ以上の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．

4.2 RQ_2: 提案手法によりどの程度ルール違反の誤検出が削減可能か?

RQ_2 に関する調査は，さらに以下の小課題に分割して考える．

RQ2.1: 提案手法により次バージョンの時点での誤検出および検出漏れを削減可能か？

RQ2.2: 提案手法により将来修正されるルール違反を早期に発見可能か？

RQ2.3: 提案手法ではどの程度の頻度でルールの有効・無効を切り替える必要があるか？

4.2.1 RQ_2 分析手法

RQ2.1: ASAT で使われるルール集合 R_{All} を，図 1 に示すように 4 つに分割する．$Config(S)$ が準拠ルールであり，それ以外が非準拠ルール $(V-rules)$ である．また、対象プロジェクトにおいて開発者が設定ファイルで有効にしているルールが有効ルール，そうでないルールが無効ルールである．

有効ルールの集合を $P-rules$ と呼ぶことにすると，開発者が実際に準拠していて，ツールとして検出も有効になっているもの（すなわち $Config(S) \cap P-rules$）が，提案手法によるルール選択の True Positive である．開発者が準拠していないが $P-rules$ に含まれているものが提案手法の False Positive，開発者は準拠しているが $P-rules$ に含まれていないものが False Negative となる．

実際には，以下のような手順でこの計算を行った．

(1) 対象プロジェクトからのソースコードファイルおよび ASAT 設定ファイルを抽出する．

(2) ASAT 設定ファイルからプロジェクトが有効にしているルール (*P-rules*) を抽出する．

(3) ASAT の API から機能上利用可能なすべてのルール集合 (R_{All}) を抽出する．

(4) R_{All} を用いて ASAT を実行し，準拠するルール $Config(S)$ を抽出する．また，非準拠ルール V-rules $= R_{All} \setminus Config(S)$ で得られる．

(5) False positive ルール集合を *P-rules* および *V-rules* の共通集合として得る．

(6) False negative ルール集合を $Config(S) \setminus (R_{All} \setminus$ *P-rules* として得る．

(1) について，本稿で対象とする JavaScript プロジェクトでは 2 種類のファイルを抽出する．まず，ASAT の検証対象であるソースコードファイルを抽出する．ESLint の場合はファイル拡張子に.js をもつファイル群が収集対象ファイルとなる．もう一つは，プロジェクトが有効にしているルールを得るために ASAT の設定ファイルを抽出する．ESlint の場合ファイル名が ".eslintrc" からはじまる Json, Yaml または JavaScript ファイルが対応する．

次に (2) について，ASAT の設定ファイルからは標準ルールをはじめ，プロジェクトのベースとしているルール集合情報とプロジェクトが個別に有効・無効にしているルール情報が抽出可能となっている．提案ツールは ASAT の設定ファイルからプロジェクトが有効にしている ASAT ルール集合 *P-rules* を取り出す．

(3) について，ASAT の標準ルールから利用可能なルールすべてを抽出する．本稿では，ESLint に標準に含まれているすべてのルール [17] に含まれるすべてのルール集合 R_{All} を抽出する．本分析の制限として，ASAT はファイルごとに適用するルールを設定する機能を提供しているが，本稿ではプロジェクトで一貫したルール方針を抽出するため，すべてのファイルをルール適用範囲とする．

(4) では，*A-rules* を有効にした ASAT を対象ソースコードファイルに対して実行し，プロジェクトのソースコードが違反するルール集合 *V-rules* を抽出する．このとき対象ソースコードファイルのうち 1 ファイルでもルール違反を検出していれば，そのルールを他のファイルが準拠していたとしても *V-rules* として扱う．

(5) について，(3)，(4) で抽出した *P-rules* と *V-rules*，両方に含まれるルール，つまりプロジェクトが定義したにもかかわらず警告が発生したルール集合を誤検出のルール集合，False positive として抽出する．

最後に (6) では，R_{All} に含まれるルールのうち，*P-rules* と *V-rules* 両方に含まれないルールをプロジェクトが定義していないにもかかわらず警告が発生しないルール集合を検出漏れのルール集合，False Negative として抽出する．

抽出した False Positive および False Negative ルールを用いて，提案手法のルール選択の正確さを Precision (適合率)，Recall（再現率），F 値を求めることで評価する．

$$Precision = \frac{|TruePositive|}{|Config(S)|} \quad (1)$$

$$Recall = \frac{|TruePositive|}{|\text{準拠ルール}|} \quad (2)$$

$$\text{F 値} = \frac{2|Recall||Precision|}{|Recall| + |Precision|} \tag{3}$$

Precision 値が高いほど設定したルールの誤検出が少なく，Recall 値が高いほどルールの検出漏れが少ない．

$RQ_{2.2}$: $RQ_{2.1}$ は，提案手法で生成したルール集合が各バージョンで準拠されているルール集合と一致しているか否かを評価した．$RQ_{2.2}$ では，有効にしたルールが検出した具体的なコーディングルール違反が，あるバージョン i では発生したが $i+1$ 以降で修正されたとき，ルールを有効にしたことで検出する価値があったルール違反として True Positive に分類し，最新版まで修正されなかったルール違反を False Positive として評価する．また，実際の開発履歴では修正されているが，提案手法のルール無効化によって検知できなかったものを False Negative として評価する．

具体的な計算として，総ファイル数を m，バージョン数を n と定義し，バージョン i の対象ファイル f におけるルール r の違反数を $v(r, f_i)$ とする．検出漏れ修正ルール違反数 $FalseNegative(f_i)$ は，バージョン k から $k+p$ の期間発生したルール違反のうちバージョン i を基準にしたルールに含まれなかったルール $V(S_i)$ を用いて検出される警告行数の総和であり，以下のように定義される．

$$\begin{aligned} FalseNegative(f_i) = &\sum_{k=i}^{n}\sum_{p=k}^{n}\sum_{f=0}^{m}\sum_{r\in V(S_i)} \\ &max(v(r, f_k) - v(r, f_{k+p}), 0) \\ where\ V(S_i) = &R_{All} \setminus Config(S_i) \end{aligned} \tag{4}$$

誤検出修正ルール違反数 $FalsePositive(f_i)$ は，バージョン i 時点のルールで違反を検出したものの最終バージョンまで開発者によって修正されなかったルール $V(S_i)$ を用いて検出される警告の総数とする．

$$\begin{aligned} FalsePositive(f_i) = &\sum_{k=i}^{n}\sum_{f=0}^{m}\sum_{r\in F(S_i)} \\ &max(v(r, f_{i+1}), v(r, f_n)) \\ where\ F(S_i) = &\{r' \in Config(S_i) | v(r', f_n) > 0\} \end{aligned} \tag{5}$$

$RQ_{2.3}$: 提案手法によるルールの有効化や無効化を頻繁に行う場合，本来必要だったルールまで無効にしてしまう可能性がある．$RQ_{2.3}$ ではルールの一貫性を評価するために，提案手法を運用した場合のルールの変更頻度を以下の通りに分類する．

- Enabled_1: ルール集合作成時から最新版時点まで有効にしたルール
- Enabled_n: ルール集合作成時では無効だったがあるバージョン n 以降から最新版時点まで有効にしたルール

表 3: $RQ_{2.1}$: Precision (精度)，Recall (再現率)，F 値の平均値および中央値

ルール集合	Precision		Recall		F 値	
	mean	med	mean	med	mean	med
bower (n=96)						
Major	0.93	0.94	0.95	0.95	0.94	0.95
Minor	0.99	0.99	0.98	0.99	0.98	0.99
Maintenance	0.99	1.00	0.99	1.00	0.99	1.00
設定ルール	0.9	0.89	0.34	0.34	0.50	0.50
eslint (n=9,223)						
Major	0.98	0.99	0.90	0.99	0.93	0.98
Minor	0.99	1.00	0.98	1.00	0.98	1.00
Maintenance	0.99	1.00	0.99	1.00	0.99	1.00
設定ルール	0.29	0.00	0.16	0.00	0.21	0.00
hexo (n=126)						
Major	0.92	0.92	0.92	0.93	0.92	0.93
Minor	0.98	0.99	0.98	1.00	0.98	1.00
Maintenance	0.99	1.00	0.99	1.00	0.99	1.00
設定ルール	0.71	0.89	0.27	0.33	0.39	0.48
karma (n=198)						
Major	0.77	0.70	0.90	0.88	0.82	0.78
Minor	0.97	0.99	0.96	1.00	0.96	0.99
Maintenance	0.99	1.00	0.99	1.00	0.99	1.00
設定ルール	0.83	0.81	0.39	0.39	0.53	0.53

- Disabled_1:ルール集合作成時から最新版時点まで無効にしたルール
- Disabled_n: ルール集合作成時では有効だったがあるバージョン n から最新版時点まで無効にしたルール
- Unstable: ルール集合作成時点から最新版までの間に 2 回以上有効，無効状態を切り替えたルール

Unstable に該当するルールはプロジェクトが修正するにも関わらず，頻繁に違反修正が遅延しているルールとなる．つまり，Unstable が少ないプロジェクトほどルールに一貫性があり，次のバージョンアップデート前までにルール違反を修正している．また，Unstable ルールが発生したバージョンの粒度が細かいほどプロジェクトはそのルール違反を早く修正する．

4.2.2 RQ_2 結果

$RQ_{2.1}$: 表 3 に提案手法により Major，Minor，Maintenance アップデートごとにルールを更新した場合と，開発者が設定したルールを利用した場合のルール精度を示す．対象プロジェクトでは Precision, Recall, F-mesure の 3 つすべてがプロジェクト高い数値を記録した．特にルールの検出漏れを評価する Recall はプロジェクト開発者による設定ルールだと中央値が 0.39 以下となるが，提案手法を用いた場合は 0.88 以上となる．開発者が提案手法を利用する場合，少なくとも Major アップデートごとに設定を更新するだけでもルールの検出漏れを大きく削減することが可能になる．

表 4: $RQ_{2.2}$: 提案手法によって早期発見した ASAT 違反行数の予測 Precision, Recall, F 値 (n=405,094,892)

bower (n=4,174,637)			
ルール集合	Precision	Recall	F 値
Init	1.00	0.01	0.02
Major	0.95	0.02	0.05
Minor	0.97	0.01	0.01
Maintenance	0.95	0.01	0.01
設定ルール	0.80	0.04	0.07
標準ルール	0.80	0.04	0.07
eslint (n=363,074,539)			
ルール集合	Precision	Recall	F 値
Init	0.64	0.00	0.00
Major	0.27	0.03	0.06
Minor	0.37	0.02	0.04
Maintenance	0.44	0.02	0.04
設定ルール	0.16	0.00	0.00
標準ルール	0.34	0.01	0.02
hexo (n=14,416,765)			
ルール集合	Precision	Recall	F 値
Init	0.58	0.04	0.08
Major	0.56	0.03	0.06
Minor	0.46	0.02	0.03
Maintenance	0.46	0.02	0.03
設定ルール	0.51	0.04	0.07
標準ルール	0.51	0.04	0.07
karma (n=23,428,951)			
ルール集合	Precision	Recall	F 値
Init	0.97	0.32	0.48
Major	0.87	0.19	0.31
Minor	0.84	0.05	0.10
Maintenance	0.83	0.05	0.09
設定ルール	0.55	0.01	0.02
標準ルール	0.55	0.01	0.02

$RQ_{2.2}$: 表 4 に違反検出後に修正されたルールの違反行数での検出精度を示す．全プロジェクトで Precision 値は開発者が設定した設定ルールや ASAT 標準ルールよりも改善している．$RQ_{2.1}$ とは反対に更新頻度が粗いほど高い値を得ている．一方で Recall の値は設定ルールや標準ルールと比較して大きく改善されていない．開発者は最低でもプロジェクト作成時にルールを生成するだけでも，手動で作成するルールと同程度の検出漏れに抑えながら，修正されないルールを無効にすることが可能になる．

$RQ_{2.3}$：表 5 に Maintenance アップデート単位でルールを更新した場合に各ルールの切替回数とその分布，また，表 6 に Major アップデート単位でのルール切り替え回数と分布を示す．提案手法を利用する場合，Maintenance 単位なら合計で 10-68 回，Maintenance 単位でも 229-452 回のルール切り替えを自動で行うことが可能になる．bower 以外のプロジェクトで Enabled_n と Disabled_n に該当するルール極端に少ないため，それぞれ Enabled_1 と Disabled_1 に統合し，以下では，5 つの分類のうち，Unstable ルールと Enabled および Disabled の 4 分類が該当する Stable ルールの 2 つに分けて議論する．

表 5: $RQ_{2.3}$: Maintenance バージョンごとに本手法を利用した場合に有効または無効に切り替えるルール分布

	Enabled		Disabled		Unstable	ルール
	1	N	1	N		切替数
bower	111	4	54	11	61	229
	46.1%	1.7%	22.4%	4.6%	25.3%	26.6%
eslint	55	0	44	0	142	314
	22.8%	0.0%	18.3%	0.0%	58.9%	45.2%
hexo	99	0	39	0	103	286
	41.1%	0.0%	16.2%	0.0%	42.7%	36.0 %
karma	69	0	60	0	112	264
	28.6%	0.0%	24.9%	0.0%	46.5%	42.4%

表 6: $RQ_{2.3}$: Major バージョンごとに本手法を利用した場合に有効または無効に切り替えるルール分布

	Enabled		Disabled		Unstable	ルール
	1	N	1	N		切替回数
bower	149	2	82	8	0	10
	61.8%	0.8%	34.0%	3.3%	0.0 %	0.0 %
eslint	61	1	146	1	32	68
	25.3%	0.4%	60.6%	0.4%	13.3%	47.1%
hexo	115	2	53	1	70	148
	47.7%	0.8%	22.0%	0.4%	29.0%	47.3%
karma	132	1	82	3	23	50
	54.8%	0.4%	34.0%	1.2%	9.5%	46.0%

Unstable ルール: 61-142 件: 25.3-45.2%のルールが該当する．言い換えると対象プロジェクトでは，229 件以上のルール違反を ASAT による警告よりも遅延して，違反を修正している．表 6 と表 5 と比較すると，発生する Unstable ルールは Major バージョンアップデートでは 70 以下となり，多くの Unstable ルールは短い期間で修正されていることが確認できる．

Stable ルール: Enabled (55-115 件) and Disabled (39-65 件): Stable ルールのうち，少なくとも 55 のルールは Enabled に分類され，bower プロジェクトでは 111 件のルールがプロジェクト開始時から常に有効であり，ASAT で設定されているか否かに関わらず開発者が準拠している．

RQ_2 結果まとめ: (1) 提案手法を用いることでルールの検出の網羅率を 0.39 から 0.88 まで改善することができる．(2) 提案手法を利用することで将来修正されるルール違反の検出漏れを維持しながら，誤検出されるルール違反を標準ルールを利用する場合よりも最大で 30%多く検出可能になる．(3) 提案手法を用いることで，最大で 314 回のルール更新を自動化することが可能になる．また，61 から 142 件の検出から遅延して修正されるルール違反を事前に発見可能になる．

5. 妥当性への脅威

5.1 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ルール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ルールを無効とする．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ルール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．

RQ_2 において，発生したルール違反を修正された行数を数え上げ，ルール集合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ルール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ルール違反の総数は変化しない．

5.2 外的妥当性

プロジェクト間でルールの数を均等化するために，外部ライブラリのルールは除外した．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．

6. まとめ

本稿では，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ルールに基づき 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ルール違反を無効にし，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ルール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 件のルールを対象に提案手法を適用することによりルール集合を生成した．本ルール集合と開発者が手動で生成したルール集合を比較して，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，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．

謝辞

本研究は科研費 JP18H03221, JP20H05706, JP20J15163 の助成を得た．

参考文献

[1] Thung, F., Lo, D., Jiang, L., Rahman, F., Devanbu, P. T. et al.: To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools, *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*, IEEE, pp. 50–59 (2012).
[2] Marcilio, D., Bonifácio, R., Monteiro, E., Canedo, E., Luz, W. and Pinto, G.: Are static analysis violations really fixed? a closer look at realistic usage of sonarqube, *27th International Conference on Program Comprehension (ICPC'19)*, pp. 209–219 (2019).
[3] Beller, M., Bholanath, R., McIntosh, S. and Zaidman, A.: Analyzing the state of static analysis: A large-scale evaluation in open source software, *the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, Vol. 1, pp. 470–481 (2016).
[4] Buckers, T., Cao, C., Doesburg, M., Gong, B., Wang, S., Beller, M. and Zaidman, A.: UAV: Warnings from multiple Automated Static Analysis Tools at a glance, *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER'17)*, pp. 472–476 (2017).
[5] Carvalho, A., Luz, W., Marcílio, D., Bonifácio, R., Pinto, G. and Canedo, E. D.: C-3PR: A Bot for Fixing Static Analysis Violations via Pull Requests, *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER'20)*, pp. 161–171 (2020).
[6] Ochodek, M., Hebig, R., Meding, W., Frost, G. and Staron, M.: Recognizing lines of code violating company-specific coding guidelines using machine learning, *Empirical Software Engineering*, Vol. 25, No. 1, pp. 220–265 (2020).
[7] Higo, Y., Hayashi, S., Hata, H. and Nagappan, M.: Ammonia: an approach for deriving project-specific bug patterns, *Empirical Software Engineering*, pp. 1–29 (2020).
[8] Network, S. D.: Code conventions for the Java programming language (1999).
[9] Smit, M., Gergel, B., Hoover, H. J. and Stroulia, E.: Code convention adherence in evolving software, *Proc. ICSM*, pp. 504–507 (2011).
[10] Pylint: https://pypi.org/project/pylint/ (2020).
[11] flake8: https://pypi.org/project/flake8/ (2019).
[12] pydocstyle: https://pypi.org/project/pydocstyle/ (2020).
[13] Standard-ECMA-262: https://www.ecma-international.org/publications/standards/Ecma-262.htm (2020).
[14] ESlint: https://github.blog/2020-06-18-introducing-github-super-linter-one-linter-to-rule-them-all/ (2020).
[15] ECMAScript-Language-Specification: https://www.ecma-international.org/ecma-262/5.1/ (2016).
[16] Gyimesi, P., Vancsics, B., Stocco, A., Mazinanian, D., Árpád Beszédes, Ferenc, R. and Mesbah, A.: BugJS: A Benchmark of JavaScript Bugs, *Proc. the 12th International Conference on Software Testing, Verification and Validation (ICST'19)* (2019).
[17] ESLint-rules: https://eslint.org/docs/rules (2020).