# ソースコードの静的検査における警告の版間追跡ツール

桑原 寛明[1,a)]　渥美 紀寿[2,b)]

**概要**：本稿では，静的検査ツールが報告する警告を版間追跡し，過去の検査において開発者が確認した警告と同じ警告を将来の検査において報告しないツールを提案する．静的検査ツールを利用する際は，繰り返し検査を実行して報告される警告を確認し，必要に応じて対処することが重要である．しかし，報告された警告が false-positive であるため無視したい場合でも，ソースコードは変更されないため同じ警告が検査を行うたびに報告される．本ツールは，ソースコードの変更前後で行う検査における警告の版間追跡と，確認済み警告の記録により，過去の検査で確認した警告と同じ警告を将来の検査では警告として報告しない．本ツールを小規模なプログラム開発に適用し，検査と警告の確認を繰り返すことで総警告数に対し確認すべき警告の数が減少することを確認した．

# A Tool for Version Tracking of Alerts in Static Checking of Source Code

Hiroaki Kuwabara[1,a)]　Noritoshi Atsumi[2,b)]

***Abstract:*** In this paper, we describe a tool to suppress the false-positive alerts reported repeatedly in static checking of source code. This tool uses version tracking of alerts and developers' decision of false-positive alerts. It is important to check and fix source code repeatedly when we use static checkers. However, static checkers report false-positive alerts repeatedly since developers ignore them and can not modify the source code to suppress them. These alerts discourage developers from using static checkers. Our tool makes it possible to mark false-positive alerts and track the alerts detected before and after editing source code for selecting the alerts that should be reported to developers. This paper describes an application of our tool to small software development and the reduction of rate of alerts reported to developers.

## 1. はじめに

ソフトウェア開発において，コーディング規約の制定，静的検査ツールを用いたソースコードの静的検査，コードレビューなどが実施されている．これらは，開発されるソースコードの品質を向上させることを目的として実施される．

コーディング規約とは，開発者がソースコードを記述する際に守るべきルールの集合である．コーディング規約は，ソースコードの保守性や再利用性，信頼性の確保を目的として，ソフトウェアの開発プロジェクトごとに制定される．コーディング規約の例として，K&R スタイル [1] や GNU コーディングスタンダード [2] といったコーディングスタイルに関する規約や，MISRA-C[3] や CERT コーディングスタンダード [4] といった不具合を埋め込まないための規約が存在する．

ソースコードの静的検査では，プログラムを実行するのではなくソースコードを静的に解析することで，ソースコードに含まれる何らかの誤りを発見する．そのためのツールとして静的検査ツールが多数存在している．ソースコードがコーディング規約に従って記述されていることを確認するための QAC[5] や CX-Checker[6]，Checkstyle[7] といったツールや，不具合を検出するための Splint[8]，Cppcheck[9]，Clang Source Analyzer[10]，FindBugs[11]，PMD[12] といった様々なツールが存在する．

コードレビューでは，開発者が記述したソースコードがコーディング規約に準拠しているか，複雑すぎないか，不

[1] 立命館大学情報理工学部
Department of Computer Science, Ritsumeikan University
[2] 名古屋大学情報連携統括本部
Information & Communications, Nagoya University
[a)] kuwabara@cs.ritsumei.ac.jp
[b)] atsumi@nagoya-u.jp

```
  ...
  if ((err = SSLHashSHA1.update(&hashCtx,
                                &serverRandom)) != 0)
      goto fail;
  if ((err = SSLHashSHA1.update(&hashCtx,
                                &signedParams)) != 0)
      goto fail;
      goto fail;
  if ((err = SSLHashSHA1.final(&hashCtx,
                               &hashOut)) != 0)
      goto fail;
  ...
fail:
  SSLFreeBuffer(&signedHashes);
  SSLFreeBuffer(&hashCtx);
  return err;
```

**図 1** iOS7.0.6 より前のバージョンの不具合箇所

具合を含んでいないかといった点について，開発者が目視で確認する．静的検査ツールは万能ではなく，検出不可能な不具合も存在するため，コードレビューが必要とされる．この時，静的検査ツールを併用することで，コードレビューの重点を静的検査ツールでは検出できない問題に置くことが可能であり，コードレビューのコストを低減できる．

図 1 に iOS7.0.6 で修正された不具合 [13] の原因箇所のコード断片を示す．常に 3 つ目の goto 文によってエラー処理へとジャンプしてしまうため不具合の原因となっている．この例のように，静的検査ツールやコンパイラによって検出できる単純な不具合が製品のリリースバージョンに残されるということが発生している．このような問題を防ぐために，静的検査ツールによるソースコードの検査は有用である．

しかし，開発現場において静的検査ツールはあまり利用されていない．Johnson らの調査 [14] では，静的検査ツールが利用されない原因として以下の事柄が挙げられている．

- false-positive が非常に多い
- ツールが報告する警告の数が多過ぎて処理しきれない
- プロジェクトに適した検査を行うようにツールをカスタマイズすることが難しい
- 検査項目などの設定をチームで共有することが難しい
- ツールが出力する警告メッセージの内容が理解できない
- コーディング作業に適合したツール統合ができない

これらの問題点を解決するために，false-positive を削減する手法 [15], [16] や，警告を重要な順にランキングする手法 [17], [18] が提案されている．これらの研究では，静的検査ツールの利用率を向上させるために，ツールが行う検査の精度を改善するというアプローチを採用している．

一方，著者らは [19] において，静的検査ツールが報告した警告について，false-positive であるか否かに関わらず開発者が内容を確認した警告は将来の検査における警告から除外する手法を提案した．開発者に提示される警告数を削減することで，警告の効率的な確認を実現する．既存の静的検査ツールをそのまま利用する場合は，報告された警告と前回の検査から変更した箇所を照らし合わせて確認すべき警告を選別する必要があるが，提案手法を用いることで前回の検査から変更した箇所において発生した警告のみを抽出できる．いくつかのオープンソースソフトウェアのリリースバーションを対象とする調査の結果として，静的検査ツールが報告する警告の総数が膨大であっても，隣接するリリースバージョン間で新たに発生する警告の数はごく少数である傾向が強い．つまり，静的検査ツールを頻繁に実行し，かつ報告される警告をこまめに確認することが有益である可能性が高い．

本稿では，[19] における警告の除外手法を実際のソフトウェア開発に適用するためにツールとして実現する．警告の除外手法自体は具体的な静的検査ツールや適用時に用いられる開発環境には依存しないが，今回は Eclipse プラグインとしてツールを実装し，Java 言語を対象とする静的検査ツールを利用する．実装したツールを小規模なプログラムの開発に適用して得られた結果について述べる．

## 2. 静的検査ツール

静的検査ツールとは，プログラムを実行するのではなくソースコードを静的に解析することで，ソースコード中に残る欠陥や様々な性質を抽出するツールを指す．非常に多くの静的検査ツールが開発されているが，それぞれのツールはソースコードの解析手法や検出可能な欠陥の種類，検出精度などが異なる．例えば，パターンマッチを用いた静的検査ツールとして，RATS[20] や ITS4[21]，Flawfinder[22] などが存在する．これらのツールでは，バッファオーバーフローやレースコンディションなど不具合を引き起こす可能性が高い関数の使い方を記述したパターンを脆弱性データベースに格納し，パターンに一致するソースコード断片を検出する．false-positive は非常に多くなるが，高速に検査を行うことが可能である．構文解析に基づく静的検査ツールには，Splint[8]，Cppcheck[9]，Clang Static Analyzer[10]，PMD[12] 等が存在する．これらのツールは，バッファオーバーフロー，メモリリーク，不正な NULL ポインタ参照，インタフェースの不整合など様々な欠陥を検出できる．

Chatzieleftheriou らは，オープンソースの静的検査ツール 4 種類（Splint, UNO, Cppcheck, Frama-C）と商用のツール 2 種類（Parasoft C++ Test, Com. B）について欠陥検出能力を調査しており [23]，それぞれのツールで検出可能な欠陥と検出精度が異なることを示している．同様に，Zitser らの調査 [24] でもオープンソースのツール 4 種類（ARCHER, BOON, Split, UNO）と商用のツール 1 種

類（PolySpace C Verifier）を対象に評価が行われており，検出可能な欠陥と検出精度について議論されている．

その他のツールについても検出可能な欠陥の種類や検出精度はそれぞれ異なっている．そのため，複数の静的検査ツールを組み合わせて活用することで，より多くの欠陥を検出しソースコードの品質を向上させることができる．しかし，その場合は Johnson らの調査 [14] で挙げられている

- false-positive が非常に多い
- 警告の数が多過ぎて処理しきれない

といった問題がさらに悪化することになる．false-positive の多いことが警告の数が多過ぎることの原因の一つであるため，false-positive の削減が対策となり得る．しかし，解析精度の向上による false-positive の削減は，多くの研究が行われているが容易ではない．一般に，静的解析に基づく検査では false-positive を減らしていくと false-negative が増えていくことになる．他の対策として，false-positive の判断は開発者が行い，その結果を以降の検査に静的検査ツールが自動的に反映させる方法が考えられるが，個々の警告を区別して扱うことが可能であり判断結果をソースコード上に記録する必要がないツールは著者らが知る限り存在していない．

## 3. 警告の版間追跡

### 3.1 追跡したい警告

多くの静的検査ツールは，検出した欠陥を警告として出力する．警告には，欠陥が検出されたソースコードのファイル名，行番号，検出された欠陥の説明などが含まれる．開発者は警告の内容とソースコードの該当箇所を参照し，不具合であると判断すれば修正を行い，そうでないと判断すれば警告を無視する．ソースコードの変更中に静的検査ツールによる検査を随時実施し，検出された欠陥を修正することでソースコードの品質を確保できる．しかし，既存の静的検査ツールは警告に対して版を跨いだ支援をしない．ソースコードを変更する前後で行った検査で得られた結果の間に関連は一切存在しない．そのため，ある検査における警告がそれ以前の検査で無視した警告であることも多いが，開発者は検査を行うたびに出力された警告とソースコードの変更箇所を比較しながら問題の有無を確認しなければならない．

静的検査ツールが報告する警告は，以下のいずれかに分類することができる．

(1) 以前の検査で報告され，修正の必要がないと判断された警告
(2) 以前の検査で報告され，修正の必要があると判断されたが未修正の警告
(3) 以前の検査でも報告されていたが，まだ確認されていない警告
(4) 以前の検査では報告されていない新しい警告

しかし，既存の静的検査ツールはこれらを区別せず，(1) の警告も検査を行うたびに報告される．そのため，何らかの対処を行うべきその他の警告なのか，無視してもよい警告なのか，開発者は何度も判断する必要があり，問題の有無を確認するコストが高くなる．多くの場合，検査の直前に変更した箇所を中心に警告を確認すれば確認のコストは下げられるが，変更箇所が関連する警告が未変更の関連する箇所に報告されるような場合には対処しきれない．静的検査ツールが報告する警告のうち，少なくとも (1) に該当する警告とその他の警告は区別される必要がある．

例えば，Java プログラム向けの静的検査ツールである FindBugs は，以下に示すソースコードの最終行に対して，「実行不可能かもしれない分岐で `null` 値の `a` を利用する可能性がある」ことを警告する．

```
if (a == null && b == null) {
  return true;
}
if ((a != null && b == null) ||
    (a == null && b != null)) {
  return false;
}
return a.equals(b);
```

しかし，実際には if 文の条件から `a` の値が `null` の場合に最終行が実行されることはないため，この警告は false-positive であると判断できる．その結果，該当のソースコードは変更されずに残り，以降の検査においても警告として報告され続けることになる．ソースコードの変更前後で警告を追跡できれば，修正する必要がない警告の報告を抑制できる．

### 3.2 警告の版間追跡の概要

著者らが [19] で提案する手法では，ソースコードの変更前後で行われた検査における警告のうち，同一箇所で検出された同一内容の警告同士を対応付ける．これにより，変更前の警告のうち変更により消滅した警告，変更後も残っている警告，および変更により新たに出現した警告をそれぞれ区別できる．さらに，変更前の各警告のうち開発者が問題のないことを確認した警告について記録しておき，変更後も残っている警告の中から確認済みの警告を除外すれば，変更後の警告の中から開発者が問題の有無を本当に確認しなければならない警告のみを抽出することができる．全体像を図 2 に示す．

我々の手法では，警告の対応付けに版間のソースコードの行単位での対応関係を利用する．ソースコード変更後の各警告について，警告が報告された行に対応する変更前の行に同一内容の警告が報告されており，かつ変更前に報告された警告が確認済みであると記録されていれば，開発者

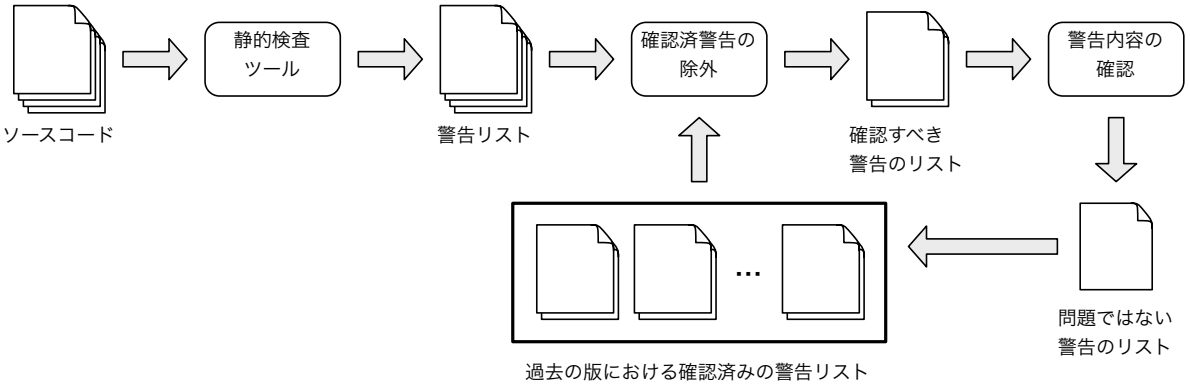


**図 2** 警告の版間追跡の全体像

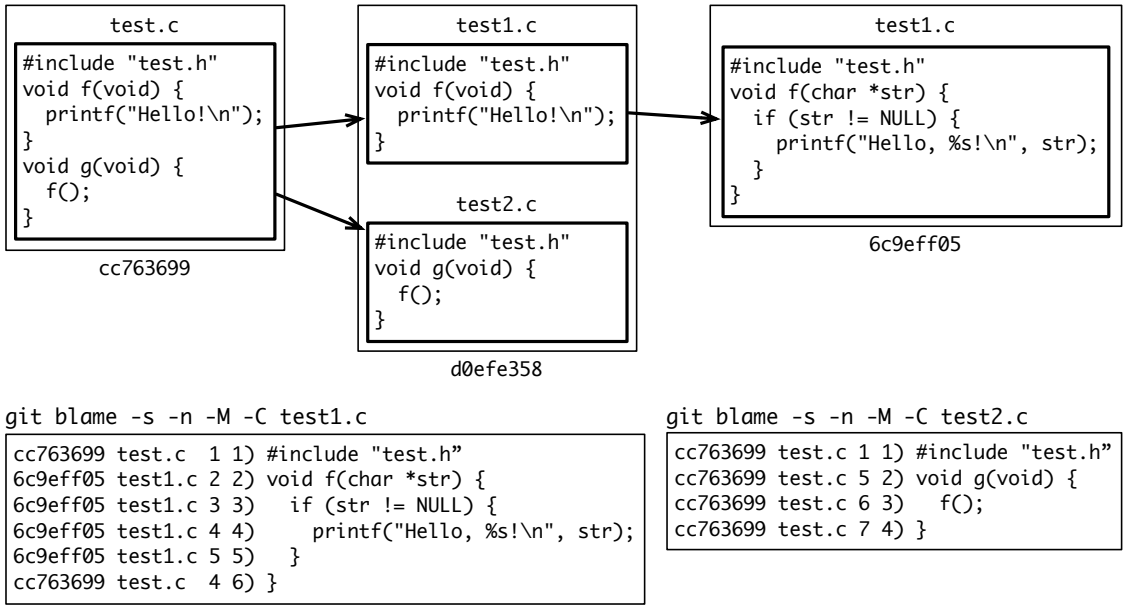


**図 3** git-blame による変更の版間追跡

に提示する警告から除外する．確認済みの警告を除外する方法として，除外したい警告が検出された箇所をコメントやアノテーションを用いてマークする方法が考えられる．しかし，この方法では警告が検出されないように修正した際にマークを忘れずに外す必要があり，マークを外し忘れると確認すべき警告が隠蔽される可能性がある．例えば，FindBugs では

```
@SuppressFBWarnings("NP_BOOLEAN_RETURN_NULL")
public Boolean getBoolean(...) {
  ...
}
```

のように`@SuppressFBWarnings`アノテーションを用いることで特定の構文要素に対して特定の検査項目に関する警告を抑制できる．しかし，アノテーションが記述されている限り警告が抑制され続けるため，アノテーションの対象を変更した場合は変更箇所の検査を行うためにアノテーションを削除しなければならない．各警告について個別に確認済みの記録と版間追跡を行うことで，警告を抑制するコメントやアノテーションの手作業による追加と削除は不要となる．

本手法では，警告の検出位置をソースコードの先頭からのオフセットや構文要素ではなく，ソースコードの行を単位として区別する．そのため，同じ行の異なる位置に出現する同一内容の警告が区別できない，同一行で警告とは無関係な箇所が変更された場合に既存の警告の追跡に失敗する，ある行の変更によって別の変更していない行に警告が出現した場合に警告行において同種の警告が確認済みであってもわからない，といった問題が残る．しかし，警告の位置情報として行番号より詳細な情報を出力しない静的検査ツールも多い．加えて，行よりも詳細な単位で編集前後の差分を追跡することも容易ではなく，手軽に利用できるツールも存在しない．これらの理由から，本手法では行

```
ViolationsView.java:173:32: Parameter viewer should be final.
ViolationsView.java:173:39: 'viewer' hides a field.
```

(a) Checkstyle

```
H D ST: Write to static field Activator.plugin from instance method Activator.start(BundleContext)
  At Activator.java:[line 96]
```

(b) FindBugs （実際は一行）

**図 4** 静的検査ツールの出力例

単位での対応関係を利用する．

### 3.3 版管理ツールを用いた版間追跡

Git や Subversion などの版管理ツールは，ファイル中のそれぞれの行について直近の変更がどの版で誰によって行われたのかを示す blame 機能を持っている．Git の blame 機能の例を図 3 に示す．

図 3 では，コミット cc763699 の test.c がコミット d0efe358 で test1.c と test2.c に分割され，コミット 6c9eff05 で test1.c 内の関数 f が変更された場合を示している．コミット 6c9eff05 で test2.c は変更されていない．オプションにもよるが git-blame の結果として，現在のファイルの各行について，変更が行われた直近のコミット，その時のファイル名と行番号，現在の行番号，現在の内容が得られる．例えば，コミット 6c9eff05 において `git blame -s -n -M -C test1.c` を実行した結果から，1 行目は最初のコミット cc763699 から変更されていないが当初のファイル名は test.c であったこと，2 行目から 5 行目はコミット 6c9eff05 で変更されたこと，6 行目はコミット cc763699 から変更されていないが，その時は test.c の 4 行目であったことがわかる．同様に `git blame -s -n -M -C test2.c` を実行した結果から，すべての行の内容はコミット cc763699 から変更されていないが，2 行目以降がコミット cc763699 では test.c の 5 行目以降に存在していたことがわかる．

git-blame のオプション `-M` によりファイル内における行の移動やコピーを，`-C` によりファイルを跨がった行の移動やコピーを検出できる．これにより，図 3 のコミット cc763699 からコミット d0efe358 への変更のようにファイルを分割した場合やファイル名を変更した場合にも行を追跡することができる．

版管理ツールのこの機能を利用することで，ソースコードのそれぞれの行において静的検査ツールが報告した警告を版を跨いで追跡することができる．異なる版において報告された警告について，警告内容が同じであり，かつそれらの警告が報告された行が変更された直近のコミット，およびその時のファイル名と行番号が同じであれば，それらの警告は版を跨いで同一の警告であると判断する．

### 3.4 確認済みの警告の記録

静的検査ツールは検査結果として警告のリストを出力する．それぞれの警告はソースコード中に検出された何らかの欠陥に対応している．出力形式はツールごとにそれぞれ異なるが，多くのツールでは，欠陥を検出したファイルの名前と行番号および欠陥の内容を説明する警告メッセージが出力される．Checkstyle と FindBugs の出力例を図 4 に示す．図 4 は開発者向けの出力の例であるが，各警告を生成した検査項目の識別子といった詳細な情報を含めて XML など機械可読な形式で出力するオプションを備えたツールも多い．

開発者は静的検査ツールが報告するそれぞれの警告について内容を確認し，ソースコードを修正すべき警告であるか，false-positive であるため無視できる警告であるか判断する．将来の検査において静的検査ツールが報告する警告の中から確認済みの警告を除外するために，開発者が内容を確認した警告を記録する．

警告の版間追跡と連動させるため，内容を確認した警告について，その警告が報告されたコミット，ソースコードのファイル名，行番号，警告内容を記録する．一般に，静的検査ツールは複数の検査項目について同時に検査を行うため，異なる検査項目に対する警告を区別するために警告内容を記録する．記録する警告内容として，静的検査ツールが生成する警告メッセージそのものではなく，検査項目を識別可能な ID やメッセージ中の定型文を利用する．同時に，版管理ツールの blame 機能を用いて，警告が報告された行が変更された直近のコミットと，その時のファイル名と行番号もあわせて記録する．

### 3.5 確認済みの警告の除外

あるコミットに対して静的検査ツールが報告する警告について，警告内容および報告された行が変更された直近のコミット，ファイル名，行番号のすべてが等しい警告が確認済みであると記録されていれば，その警告は過去の版において確認済みであるとみなすことができるため，開発者に提示する警告から除外する．

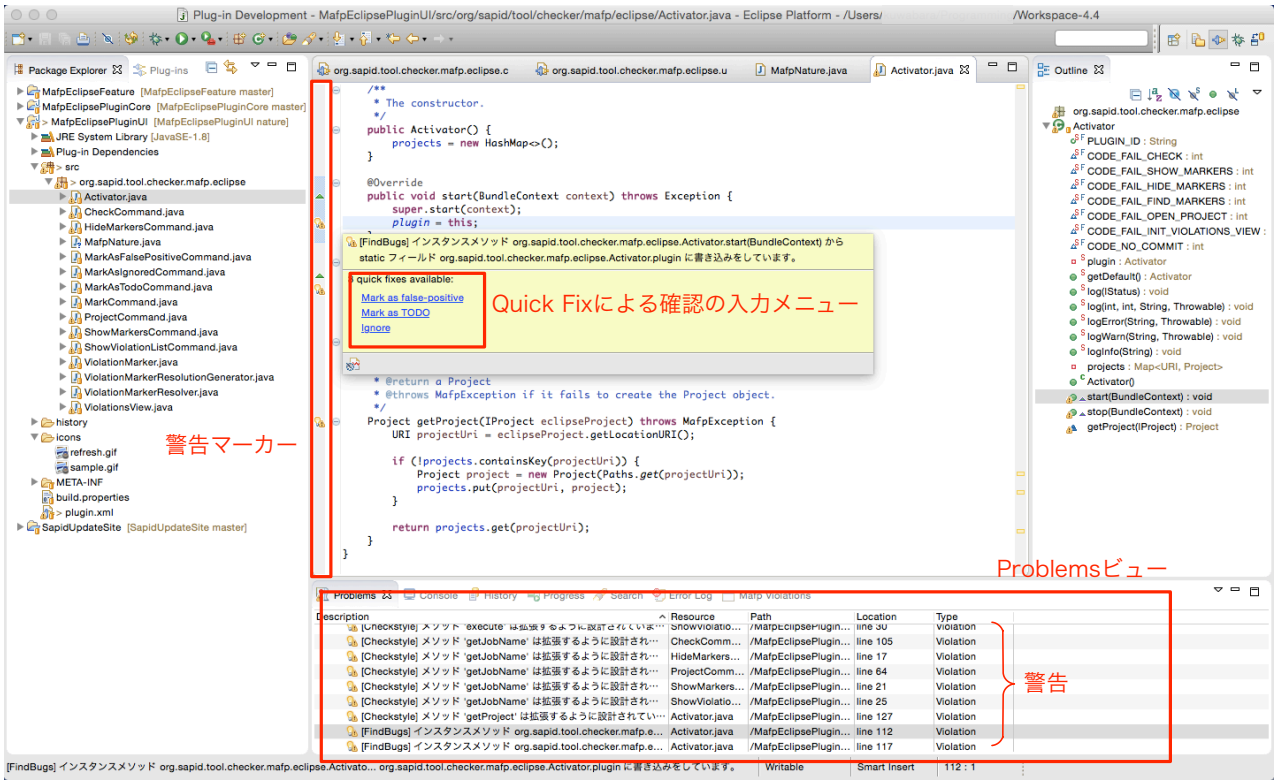


**図 5** ツールの画面例

## 4. ツールの実装

提案手法を実際のコーディング作業に適用するためにツールとして実現する．開発者が提示されたそれぞれの警告に対して確認したことを入力するユーザインタフェースが必要であるため，今回は Eclipse プラグインとして実装した．blame 機能を備えた版管理ツールとして Git を採用し，Java 言語向けの API である JGit を用いる．提案手法は検査対象のプログラミング言語を問わないが，今回は Java 言語を対象とする．静的検査ツールとして Checkstyle と FindBugs を利用し，記録する警告内容として検査項目に固有の ID を用いる．これらの静的検査ツールに限らず，報告する各警告について対象のファイル名と行番号および検査項目を識別可能な情報を生成するツールであれば利用できる．開発者が確認した警告の記録には RDBMS である SQLite を利用する．

ツールの画面例を図 5 に示す．図の画面は，検査結果として確認すべき警告が表示されており，さらに一つの警告について確認したことを入力するためのメニューが表示されている様子を示している．警告はマーカーとして対象のファイルに付加されるため，エディタの該当行の左端のアイコンや Problems ビューの一覧に表示される．それぞれの警告に対して開発者が確認したことを入力するために，マーカーを解消する手段を提供するための Eclipse 標準の仕組みを利用する．これにより，Eclipse の Quick Fix を用いて確認したことを入力できる．所定の方法で Quick Fix を呼び出し，確認したことを入力するための項目を選択すればよい．警告を確認した開発者が下す判断はソースコードを修正するか否かのいずれかであるが，修正しないとの判断は，警告が false-positive であるためか，静的検査ツールの設定が十分ではなく本来報告されないはずの警告であるためか，いずれかの場合に区別できる．本ツールでは，これらの判断を区別して入力する．静的検査ツールの設定が不十分であるとの判断を残すことで，静的検査ツールの望ましい設定の追求に活用できる．

検査の実行から確認すべき警告の表示までの処理の流れは以下の通りである．

(1) プロジェクトを対象に検査が起動される
(2) プロジェクト内のファイルを Git リポジトリにコミットする
(3) Checkstyle と FindBugs を用いて検査を行い警告リストを取得する
(4) 各警告に対し現在のコミット ID，ファイル名，行番号，検査項目の ID，該当行を変更した直近のコミット ID，その時のファイル名と行番号を特定する
(5) 各警告に対し，検査項目の ID，該当行を変更した直近のコミット，その時のファイル名と行番号がすべて同じ警告が確認済みの警告としてデータベースに登録されていればリストから除外する

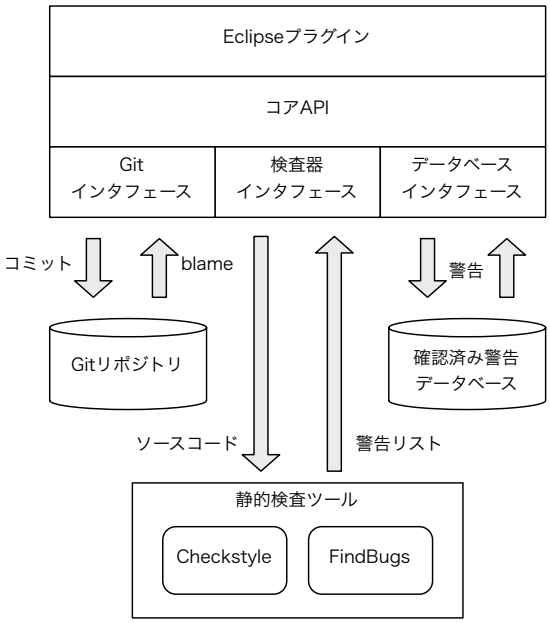


**図 6** 内部構成

(6) 除外されずリストに残った警告をマーカーとして表示する

Eclipse ではプロジェクトを単位としてソースコードなどのリソースを管理するため，プロジェクトを対象として検査を実行する．Checkstyle と FindBugs を用いて検査を行う前に，前回の検査後に行われた各ファイルの変更を Git リポジトリにコミットして版を改める．警告の版間追跡のために，毎回の検査が異なる版に対して行われるようにする．各警告が報告された版を識別するために，検査を行った版のコミット ID を付加する．警告対象のファイル名と行番号は Checkstyle と FindBugs の出力から抽出できる．検査項目の ID として，Checkstyle では検査を実装したクラスの完全修飾名，FindBugs では固有の識別子が出力されるのでそれらを利用する．該当行を変更した直近のコミットおよびその時のファイル名と行番号を得るために，警告が報告された各ファイルに対して git-blame を実行する．データベースには，確認済みの警告について検査項目の ID，該当行を変更した直近のコミット，その時のファイル名と行番号が登録されているので，これらが一致するか否か報告された各警告に対して比較を行う．

開発者は表示されたそれぞれの警告の内容を確認し対処を決定する．確認の結果，修正すべき警告であると判断した場合はソースコードを適切に変更する．適切な変更が行われれば，次の検査では該当箇所に警告は報告されない．一方，false-positive である，あるいは設定の不備であると判断した場合は，そのことを Quick Fix を呼び出して入力する．開発者が判断を入力した警告は，確認済みの警告として検査項目の ID，該当行を変更した直近のコミット，その時のファイル名と行番号がデータベースに登録される．

ツールの内部構成を図 6 に示す．Eclipse プラグインはユーザーインタフェースを提供し，Quick Fix や検査の起動メニュー，マーカーによる確認すべき警告の表示を行う．主要な処理はコア API として提供されており，Git リポジトリの操作，静的検査ツールの呼び出し，データベースへのアクセスを統合して必要な処理を実現する．

## 5. 適用例

本ツールを実際のコーディング作業に用いた例として，小規模なプログラム開発に適用して得られた結果を示す．適用対象は本ツールの Eclipse プラグインの開発である．コア API の開発は Eclipse プラグインとは別のプロジェクトとなっており，適用対象には含まれない．対象の Eclipse プラグインは，1200 行弱の Java ファイルと 250 行程度の XML ファイルからなる小規模なプログラムであり，1 名で開発されている．Checkstyle の設定にはパッケージに標準で含まれている `sun_checks.xml` を利用し，FindBugs の設定は何も指定せずデフォルトのままである．

連続する 18 回分の検査結果について表 1 に示す．表の上から新しい順に並んでいる．表 1 の列は左から順に，検査を行った版のコミット ID，検査時の Java ファイルの総行数，直前の版から変更された行数，静的検査ツールが報告した警告の数，その版で新しく報告された警告の数，確認済みの警告を除去して得られた確認すべき警告の数，表示された確認すべき警告のうち開発者が確認した警告の数を表している．例えば，最新の検査では，静的検査ツールが報告した警告は全部で 215 個であるが，そのうちの 171 個は以前の検査において開発者が確認した警告と同じであり，実際に確認すべき警告は 1 個の新しい警告を含む 44 個であったことがわかる．Checkstyle と FindBugs は Eclipse プラグインが存在するが，これらのプラグインを用いた場合は 170 個から 230 個を超える警告が検査のたびに報告される．その中から 20 個程度の確認すべき警告を見つけ出すには時間と手間を要するが，本ツールを利用すればその作業を開発者が行う必要はない．

開発者が一度確認した警告は，その警告が報告された行が変更されない間は表示されない．そのため，前回の検査から変更された行数が少なければ確認すべき警告の数は一般に減少すると考えられる．実際に，表 1 においてコミット 4770cfa から f899d94 へかけて変更行数は減少する傾向にあり，確認すべき警告の数も総警告数の 2 分の 1 から 4 分の 1 へと減少する傾向であったことがわかる．コミット e90861a と 937700e の間で新しい機能を実装したため，コミット 937700e では変更行数が多く確認すべき警告の数も増加している．その後，変更の行数は少なく，かつ警告の確認を少しずつ行っているため，確認すべき警告の数は少なくなっている．

**表 1** 適用結果

| コミット ID | Java ファイル総行数 | 変更行数 | 総警告数 | 新しい警告数 | 確認すべき警告数 | 確認した警告数 |
|---|---|---|---|---|---|---|
| eb03eeb | 1171 | 7 | 215 | 1 | 44 | 1 |
| 0e05cae | 1164 | 10 | 214 | 3 | 47 | 4 |
| 55c727a | 1157 | 16 | 214 | 5 | 50 | 6 |
| 070fef0 | 1152 | 31 | 216 | 3 | 46 | 1 |
| 8aed3b1 | 1130 | 1 | 215 | 0 | 43 | 0 |
| 700f1c5 | 1198 | 13 | 231 | 1 | 46 | 3 |
| 832fa7f | 1191 | 82 | 229 | 19 | 62 | 17 |
| 937700e | 1112 | 205 | 215 | 52 | 83 | 40 |
| e90861a | 942 | 31 | 176 | 9 | 43 | 12 |
| f899d94 | 936 | 11 | 173 | 0 | 34 | 0 |
| d8d971f | 925 | 19 | 176 | 3 | 41 | 7 |
| ec4f2ce | 920 | 2 | 176 | 0 | 62 | 24 |
| 9a1abea | 920 | 25 | 178 | 1 | 62 | 0 |
| 8b045e2 | 920 | 7 | 203 | 4 | 67 | 6 |
| f476614 | 920 | 1 | 210 | 0 | 63 | 0 |
| e0af957 | 926 | 74 | 213 | 20 | 76 | 13 |
| 56ac823 | 860 | 98 | 195 | 19 | 106 | 50 |
| 4770cfa | 799 | 111 | 181 | 31 | 113 | 26 |

利用している静的検査ツールは Checkstyle と FindBugs であるが，すべての版の検査において FindBugs による警告は 2 つだけであり，その他の警告はすべて Checkstyle による．FindBugs が報告した 2 つの警告はすべての版で同じであり，いずれかの版で確認済みと記録されれば以降の版では表示されない警告である．ただし，今回の例では未確認のままとなっている．これら 2 つの警告は，いずれもインスタンスメソッド内で static フィールドを変更していることの指摘である．これらの処理は Eclipse プラグインにおける定型処理であり変更することができないため，false-positive であると判断できる．このように，警告が false-positive であるか否かは一般にコンテキストによるが，FindBugs に限らず既存の静的検査ツールにとって警告が false-positive か否かの判断をコンテキストを利用して行うことは困難である．本ツールでも判断についてはやはり開発者に委ねるが，警告の版間追跡によって過去の検査結果に対する判断を将来の検査結果に対して自動的に適用できる．過去に false-positive であると判断された警告は，将来の検査においても false-positive であると自動的に判断される．

## 6. 関連研究

ソースコードの品質向上のための静的検査ツールに関する研究として，false-positive や false-negative の削減といった検査精度の向上を目的とした研究，重要な欠陥を優先して修正するための警告の提示方法に関する研究，静的検査ツールの利用促進を目的とする研究などが行われている．

### 6.1 検査の精度向上

Thung らは，FindBugs, PMD, Jlint の 3 種類の静的検査ツールを対象に false-negative について評価している [15]．false-negative とは，本来欠陥であるはずだが検出されないものを指す．彼らは，検査対象として 3 種類のオープンソースソフトウェアを用いて実験を行い，FindBugs と PMD は false-negative の抑制に比較的有用であるとの結論を下している．しかし，いずれの静的検査ツールにおいても誤検出を防止することはできていない．

Nanda らは，NULL ポインタの間接参照の検査において，path-sensitive かつ context-sensitive かつメソッドを跨いだ解析を行うことで false-positive を削減する手法を提案している [16]．3 種類の商用製品を対象とした検査結果について，FindBugs および Jlint による検査結果と比較してより多くの欠陥を検出可能であることと，false-positive が少ないことを示した．

これらの研究では，false-positive あるいは false-negative の削減が目的であり，開発者が確認すべき警告を減らす点で本研究と類似している．しかし，false-positive を完全になくすことは困難であり，同じ false-positive が繰り返し報告されることは避けられない．我々の手法では，開発者が false-positive と判断した警告については，警告が報告された行を追跡できる限り再度出力されることがないため，開発者が同じ false-positive を何度も確認する必要はない．

### 6.2 警告の提示法

Muske らは，静的検査ツールが提示する警告を複数のクラスに分類して冗長な警告を特定する手法を提案して

いる [25]．初めに，警告を類似度に基づいて複数のクラスに分類し，false-positive が含まれるクラスに属する警告は false-positive であると判断する．次に，警告箇所のソースコード中で参照される変数の変更箇所の類似性に基づいて再分類する．合計で 14MLOC の C プログラムを対象とする実験から，およそ 6 割の警告が冗長な警告であることが示されている．

Shen らは，FindBugs の警告に対し true-positive な警告を上位に提示するための 2 段階のランキング手法を提案している [17]．これにより，false-positive な警告を開発者の目に触れにくくすることができる．ランキングの第一段階では，あらかじめバグパターンごとに定められた欠陥の可能性によって警告をランキングする．第二段階では，ユーザからのフィードバックによりランキングを最適化する．AspectJ，Tomcat，Axis の 3 種類の Java プロジェクトに対して適用実験を行い，上位に提示される警告の適合率，再現率，F1 値が FindBugs のオリジナルの提示と比較して向上したことが示されている．

これらの研究は，false-positive な警告を開発者の目に触れにくくするという点で本研究と類似しているが，false-positive か否かの判断を自動化している点が異なる．これらの手法を我々の手法に適用することによって，開発者が確認すべき警告のさらなる削減を実現できる可能性がある．

### 6.3 利用促進

Tripp らは，静的検査ツールが提示する警告の一部について利用者にフィードバックさせ，そのフィードバックに基づく統計的学習により警告を分析する手法を提案し，手法を実現するツール ALETHEIA を開発している [26]．実験において，200 の警告をユーザのフィードバックに基づいて分類することで適合率を向上させられることを示している．

新井らは，静的検査ツールを利用する開発者の欠陥修正に対するモチベーションを向上させ，静的検査ツールが検出した欠陥がより多く修正されるように，ゲーミフィケーションに基づく仕組みを提案している [27]．提案手法を導入したツールを開発して被験者実験を行ったところ，修正される欠陥の数がおよそ 1.5 倍になったことが示されている．

Sadowski らは，コードレビュープロセスに統合された静的検査プラットフォームを提案している [28]．コードレビューの際に，レビュー対象のコードを静的検査した結果が表示され，レビューアは各警告に対し false-positive であるとして無視するか，修正するように開発者に依頼できる．彼らのシステムでは，報告した警告が無視される割合が高い検査器を改善されるまで停止させることで，レビューアや開発者に対して提示される false-positive を少なくすることに成功している．

これらの研究では，本研究と同様に静的検査ツールがあまり活用されていないという問題に取り組んでおり，我々の手法とは異なるアプローチで開発者が静的検査の結果に対応しやすくする方法が提案されている．

## 7. おわりに

本稿では，静的検査ツールが報告する警告を版間追跡することで，過去の版における検査結果に対して開発者が確認した警告と同じ警告を将来の版に対する検査において報告しないツールを提案した．警告の版間追跡には版管理ツールの blame 機能を利用する．異なる版における 2 つの警告について，警告内容が同じであり，かつ警告が報告された行に対して blame 機能によって得られる直近の変更が行われたコミットとその時のファイル名および行番号が同じであれば，その 2 つの警告は同じ警告であると判断する．静的検査ツールが報告する警告のうち開発者が内容を確認した警告を記録しておくことで，以降の検査において確認済みの警告と同じ警告を開発者に提示しないようにできる．

ツールを Eclipse プラグインとして実装して[*1]小規模なプログラム開発に適用した．検査と警告の確認を繰り返し行うことで，検査と検査の間で行われるファイルの変更が少なければ，総警告数に対して確認すべき警告の割合が減少することを確認した．false-positive であると判断できる警告についても確認済みであると記録されていれば，その警告が報告された行を変更しない限り，以降の検査において警告として報告されることはない．これにより，既存の静的検査ツールをそのまま利用するよりも効率的に検査結果を確認することが可能となる．

今後の課題として，利用する静的検査ツールを増やし，規模の大きなプログラムに対して適用することが挙げられる．一般に，規模が大きくなるほど報告される警告の数も多くなるため，少数の確認すべき警告のみを提示することは，開発者に警告を確認し必要な修正を行うように促す点から重要である．

本ツールでは警告の版間追跡に git blame を用いており，追跡の精度は git blame の性能に依存している．一方 Avgustinov らは，警告の行番号，警告行のコード，警告行周辺のコードを用いた警告の版間追跡アルゴリズムを提案している [29]．警告の版間追跡の精度は，警告が確認済みか否か判断するために重要であり，版間追跡のアルゴリズムの性能を比較することは今後の課題として挙げられる．

本ツールを活用し，静的検査ツールが報告する警告の中で false-positive な警告をコンテキストを含めて収集し分

[*1] Eclipse の Software Site に `http://www.hpcss.is.ritsumei.ac.jp/Member/kuwabara/sapid-beta/eclipse/` を指定することでインストールできる．Eclipse 4.4 以上で動作確認している．

析することも今後の課題である．報告された警告が false-positive である確率の推定，誤って false-positive であると判断された警告の検出，一貫していない開発者の判断の検出などをコンテキストに基づいて行うことで，静的検査ツールを活用しやすくなることが期待できる．

**謝辞** 本研究の一部は，JSPS 科研費 15K15973，24300006，15H02683 の助成による．

## 参考文献


[1] Kernighan, B. W. and Ritchie, D. M.: *C Programming Language*, Prentice Hall (1988).

[2] GNU coding standard, `http://www.gnu.org/prep/standards/`.

[3] Association, M. I. S. R.: *Guidelines for the Use of the C Language in Critical Systems*, Motor Industry Research Association (2014).

[4] CERT Coding Standards, `https://www.securecoding.cert.org/`.

[5] 静的検査ツール QAC, `http://www.programmingresearch.com/products/qac/`.

[6] 大須賀俊憲，小林隆志，渥美紀寿，間瀬順一，山本晋一郎，鈴村延保，阿草清滋：CX-Checker：柔軟にカスタマイズ可能な C 言語プログラムのコーディングチェッカ，情報処理学会論文誌，Vol. 53, No. 2, pp. 590–600 (2012).

[7] Checkstyle, `http://checkstyle.sourceforge.net/`.

[8] Splint - Secure Programming Lint, `http://www.splint.org/`.

[9] Cppcheck A tool for static C/C++ code analysis, `http://cppcheck.sourceforge.net/`.

[10] Clang Static Analyzer, `http://clang-analyzer.llvm.org/`.

[11] FindBugs - Find Bugs in Java Programs, `http://findbugs.sourceforge.net/`.

[12] PMD, `http://pmd.sourceforge.net/`.

[13] Bland, M.: Finding More Than One Worm in the Apple, *Communications fo the ACM*, Vol. 57, No. 7, pp. 58–64 (2014).

[14] Johnson, B., Song, Y., Murphy-Hill, E. and Bowdidge, R.: Why Don't Software Developers Use Static Analysis Tools to Find Bugs?, *Proceedings of the 2013 International Conference on Software Engineering*, pp. 672–681 (2013).

[15] Thung, F., Lucia, Lo, D., Jiang, L., Rahman, F. and Devanbu, P. T.: 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*, pp. 50–59 (2012).

[16] Nanda, M. G. and Sinha, S.: Accurate Interprocedural Null-Dereference Analysis for Java, *Proceedings of the 31st International Conference on Software Engineering*, pp. 133–143 (2009).

[17] Shen, H., Fang, J. and Zhao, J.: EFindBugs: Effective Error Ranking for FindBugs, *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pp. 299–308 (2011).

[18] Heckman, S. and Williams, L.: On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques, *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pp. 41–50 (2008).

[19] 渥美紀寿，桑原寛明：静的検査ツールにおける警告箇所の版間追跡による確認コスト削減手法，情報処理学会研究報告ソフトウェア工学，Vol. 2015-SE-187 (2015).

[20] RATS - Rough Auditing Tool for Security, `https://code.google.com/p/rough-auditing-tool-for-security/`.

[21] ITS4: Software Security Tool, `http://www.cigital.com/its4/`.

[22] Flawfinder, `http://www.dwheeler.com/flawfinder/`.

[23] Chatzieleftheriou, G. and Katsaros, P.: Test-Driving Static Analysis Tools in Search of C Code Vulnerabilities, *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*, pp. 96–103 (2011).

[24] Zitser, M., Lippmann, R. and Leek, T.: Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code, *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, pp. 97–106 (2004).

[25] Muske, T. B., Baid, A. and Sanas, T.: Review Efforts Reduction by Partitioning of Static Analysis Warnings, *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pp. 106–115 (2013).

[26] Tripp, O., Guarnieri, S., Pistoia, M. and Aravkin, A.: ALETHEIA: Improving the Usability of Static Security Analysis, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 762–774 (2014).

[27] Arai, S., Sakamoto, K., Washizaki, H. and Fukazawa, Y.: A Gamified Tool for Motivating Developers to Remove Warnings of Bug Pattern Tools, *Empirical Software Engineering in Practice (IWESEP), 2014 6th International Workshop on*, pp. 37–42 (2014).

[28] Sadowski, C., van Gogh, J., Jaspan, C., Söderberg, E. and Winter, C.: Tricorder: Building a Program Analysis Ecosystem, *Proceedings of the 37th International Conference on Software Engineering*, pp. 598–608 (2015).

[29] Avgustinov, P., Baars, A. I., Henriksen, A. S., Lavender, G., Menzel, G., de Moor, O., Schäfer, M. and Tibble, J.: Tracking Static Analysis Violations over Time to Capture Developer Characteristics, *Proceedings of the 37th International Conference on Software Engineering*, pp. 437–447 (2015).