MAFP: ソースコードに対する静的検査における警告の管理ツール

渥美 紀寿　桑原 寛明

ソースコードの品質を向上させることを目的とし，様々な静的検査ツールが開発されている．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，静的検査ツールが報告する警告には false-positive が含まれており，それらはソースコードを改変後も繰り返し警告として報告される．この問題を解決するために，本稿では，ソースコードの静的検査において得られる警告に関して，開発者が修正の必要性の有無を記録し，その情報を管理するツールを提案する．本ツールでは，ソースコードの版管理ツールを利用して，記録された警告の対象コードの変更前後の対応箇所を追跡し，既に記録された修正の必要のない警告を非表示にする．これにより，何度も同じ違反を確認する必要がなくなる．

To improve the quality of the source code, a various static checkers have been developed. It is important to check and fix source code repeatedly when we use static checkers. However, the alerts reported by static checker include false-positive and those are reported repeatedly. In this paper, we propose a tool to manage the alerts. Our tool records the check rule, the alert message, the target line of source code, and developer's mark for ignoring such as false-positive. We use the blame function of version control system to trace the target code and suppress the alerts marked as the false-positive. Therefore, developer does not need to check the same alerts repeatedly.

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 [8] や GNU コーディングスタンダード †1 といったコーディングスタイルに関する規約や，MISRA-C [2] や CERT コーディングスタンダード †2 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 †3 や CX-Checker †4 [12]，Checkstyle †5 といったツール

MAFP: Alert Management Tool for Static Checking of Source Code.

Noritoshi Atsumi, 京都大学学術情報メディアセンター, Academic Center for Computing and Media Studies, Kyoto University.
Hiroaki Kuwabara, 南山大学情報センター, Center for Information and Communication Technology, Nanzan University.

コンピュータソフトウェア, Vol.33, No.4 (2016), pp.50–66.
[ソフトウェア論文] 2015 年 10 月 30 日受付.

†1　http://www.gnu.org/prep/standards/
†2　https://www.securecoding.cert.org/
†3　http://www.programmingresearch.com/products/qac/
†4　http://cxc.sapid.org/
†5　http://checkstyle.sourceforge.net/

```
  ...
  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

や，不具合を検出するための Splint[†6]，Cppcheck[†7]，Clang Source Analyzer[†8]，FindBugs[†9]，PMD[†10] 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 [4] 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．

†6 http://www.splint.org/
†7 http://cppcheck.sourceforge.net/
†8 http://clang-analyzer.llvm.org/
†9 http://findbugs.sourceforge.net/
†10 http://pmd.sourceforge.net/

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 [7] 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] [11] や，警告を重要な順にランキングする手法 [14] [6] 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 [3] 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，静的検査ツールによって報告される警告を管理し，過去に開発者が確認して false-positive

や不要な警告と判断した警告の除外する手法を提案する．また，提案手法を実現した Eclipse プラグイン MAFP について，個々の機能とその利用方法について紹介する．提案手法およびツールは [9] で発表したものである．本稿ではツールの機能，使用方法およびツールの評価について詳細を述べる．ツールの評価ではオープンソースソフトウェアを対象に，開発者が警告を確認するコストの削減効果と，MAFP を用いて検査を開始してから警告箇所をマーカーで表示するまでの実行時間に関する結果を示した．

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．たとえば，パターンマッチを用いた静的検査ツールとして，RATS[†11] や ITS4[†12]，Flawfinder[†13] 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，Cppcheck，Clang Static Analyzer，FindBugs，PMD などが存在する．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 [5]，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 [17] 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 [7] 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 1 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．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 警告の管理手法

3.1 概要

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

†11 https://code.google.com/p/rough-auditing-tool-for-security/
†12 http://www.cigital.com/its4/
†13 http://www.dwheeler.com/flawfinder/

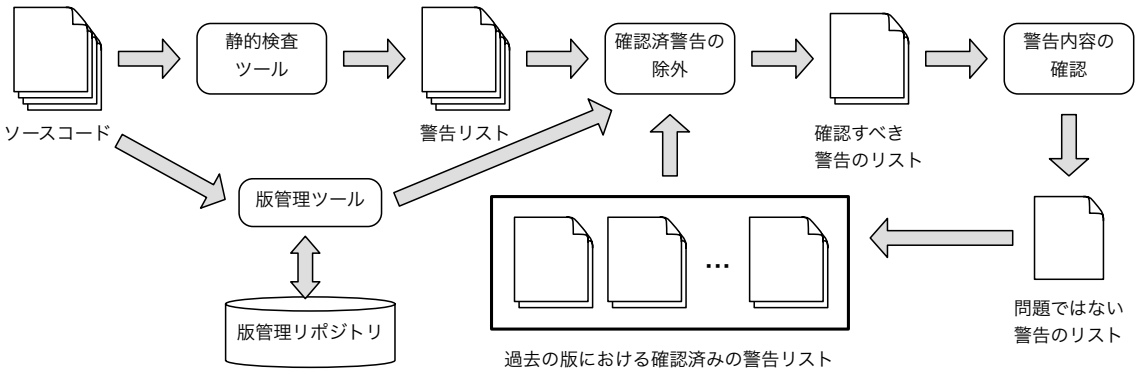

図 2 警告の版間追跡の全体像

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，修正の必要がないと判断された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，少なくとも修正の必要がないと判断された警告とその他の警告は区別される必要がある．

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．たとえば，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．

ソースコードを変更する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．提案手法では，警告の対応付けに変更前後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，開発者に提示する警告から除外する．全体像を図 2 に示す．

```
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 3 静的検査ツールの出力例

3.2 修正の必要のない警告

静的検査ツールが報告する警告のうち，修正の必要のない警告には，誤って報告される false-positive な警告と，開発者が不要と判断した検査ルールによる警告 (以下，不要な警告と呼ぶ) がある．

たとえば，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」と警告する．

```
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．また，以下に示すソースコードの 2 行目に対して,「文字列を `==` や `!=` を用いて比較しています」と警告する．

```
//String a, b
return a == b;
```

String 型の変数 `a` と `b` を `==` で比較しているが，変数 `a` と `b` に文字列定数が代入される場合には問題はないため，そのような場合には false-positive であると判断できる．

false-positive な警告は，コードを修正する必要がないため，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．また，Checkstyle などのコーディングスタイルに関するルールでは，開発者のスタイルと合わないルールがデフォルトの設定には含まれている．そのため，検査ツールで検出するルールを適切に設定しなければ，検査するたびに同様の警告が報告され続ける．提案手法では開発者が false-positive または不要と判断した警告を記録し，それらを除外して残った警告を開発者に提示する．

3.3 警告に対する開発者の判断の記録

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 3 に示す．図 3 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 や不要な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，静的検査ツール実行時にその時点のソースコードを版管理ツールに

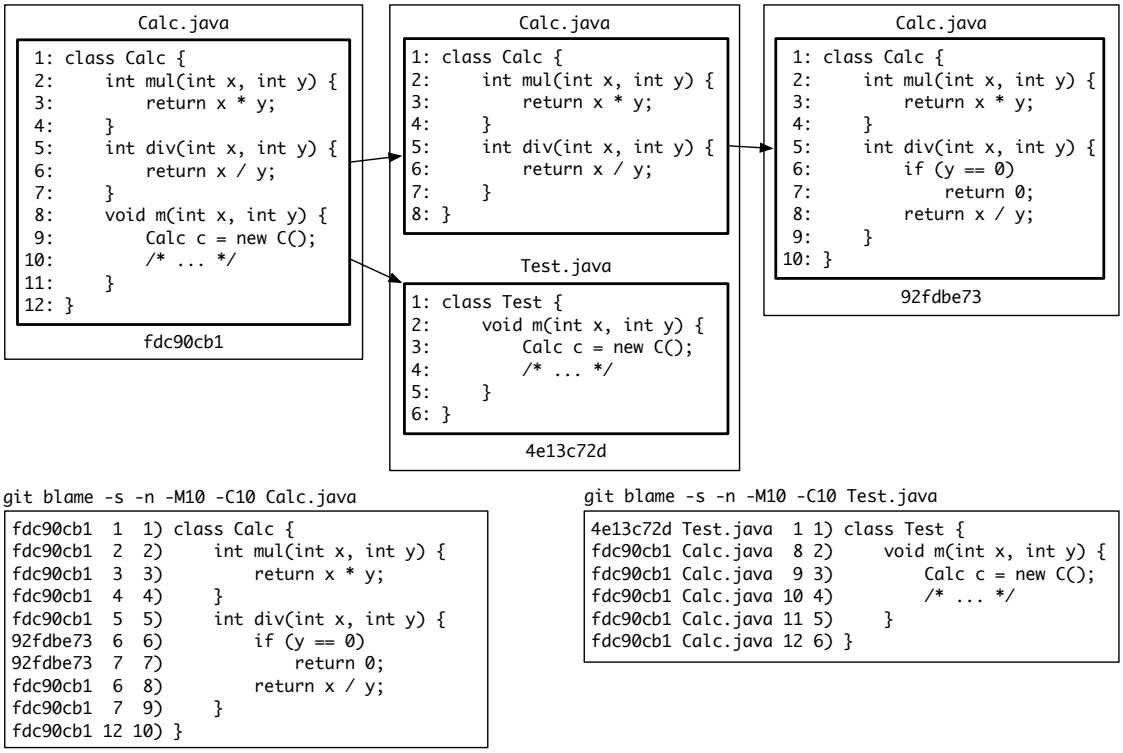

図 4 git-blame による変更の版間追跡

コミットする．同時に下記警告情報と警告対象の位置情報を記録する．

- 警告情報
  - 警告 ID
  - 警告を報告したツール名
  - 検査ルール ID
  - 警告メッセージ
  - 警告対象の位置 ID
- 警告対象の位置情報
  - 位置 ID
  - コミット ID
  - ファイル名
  - 開始行番号
  - 開始列番号
  - 終了行番号
  - 終了列番号
  - 警告の発生リビジョンの位置 ID

警告情報を記録する際，警告対象の行を過去のリビジョンを遡って追跡し，追跡された行に同じ検査ルール ID の警告情報が存在した場合には初出のリビジョンにおける位置 ID を，存在しない場合には現在のリビジョンにおける位置 ID を発生リビジョンの位置 ID として記録する．ソースコードの行の追跡方法については 3.4 節で詳しく説明する．

開発者が警告の内容を確認した警告については，false-positive な警告と不要な警告をそれぞれ分けて，警告 ID を記録する．この情報を利用することによって過去に確認し，修正の必要がないと判断した警告を除外することが可能となる．

3.4 警告の版間追跡

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 4 に示す．

図 4 では，コミット fdc90cb1 の Calc.java がコミット 4e13c72d で Calc.java と Test.java に分割され，コミット 92fdbe73 で Calc.java 内のメソッド div が変

更された場合を示している．コミット 92fdbe73 で Test.java 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．たとえば，コミット 92fdbe73 において `git blame -s -n -M10 -C10 Calc.java` を実行した結果から，1〜5 行目はコミット fdc90cb1 から変更がなく，6, 7 行目はコミット 92fdbe73 で変更されたこと，8〜10 行目はコミット fdc90cb1 から変更されていないが，順にコミット fdc90cb1 の 6, 7, 12 行目であったことがわかる．同様に `git blame -s -n -M10 -C10 Test.java` を実行した結果から，1 行目がコミット 4e13c72d の 1 行目，2〜6 行目はコミット fdc90cb1 において Calc.java の 8〜12 行目であったことがわかる．

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 4 のコミット fdc90cb1 からコミット 4e13c72d 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．

3.5 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．

本手法では，警告対象の位置情報として，開始行と開始列，終了行と終了列を記録する．一方，警告対象の追跡は行を追跡することによって，同じコードに対する警告を判別するため，同一行で異なる位置に出現する同一内容の警告を区別することができない，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，本手法では行単位での対応関係を利用する．

4 MAFP: 静的検査における警告の管理ツール

4.1 ツールの概要

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，それを実現したツール MAFP †14 は Java 言語を対象とした Eclipse プラグインである．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 を用いている．静的検査ツールとして Checkstyle と FindBugs を利用している．開発者が確認した警告の記録には RDBMS である SQLite を利用している．

本ツールの主要な機能は，(1) FindBugs と Checkstyle による静的検査実行機能，(2) FindBugs と Checkstyle によって報告された警告の提示機能，(3) 修正の必要のない警告の登録機能から構成される．ツールの内部構成を図 5 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している．

†14 Eclipse の Software Site に http://sst.se.nanzan-u.ac.jp/sapid-beta/eclipse/ を指定することでインストールできる．Eclipse 4.5 以上で動作確認している．

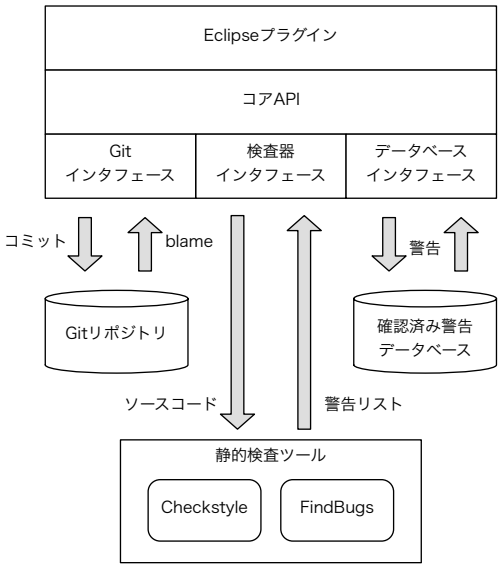

図 5 内部構成

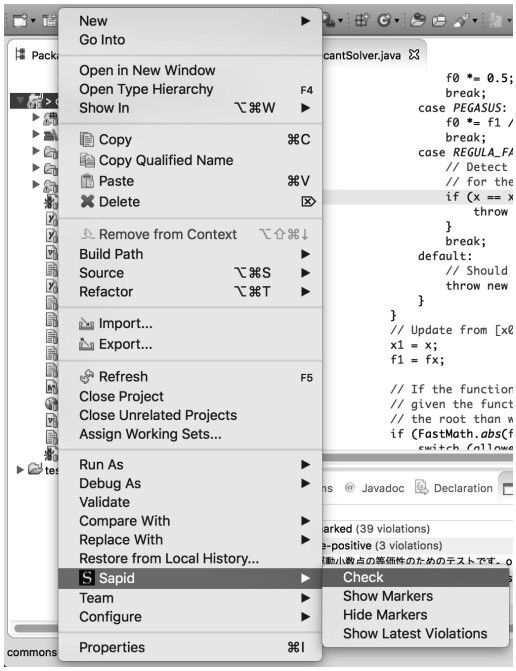

図 6 静的検査実行

検査の実行から修正の必要のない警告を登録するまでの処理の流れは以下の通りである．データベースには 3.3 節で述べた警告情報と位置情報それぞれに対応するテーブルと，修正の必要のない警告 ID を記録する false-positive と ignore に対応するテーブルの計 4 つのテーブルがある．ignore とは 3.2 節で述べた不要な警告のことである．以下の処理においてこれらのテーブルにそれぞれの情報が格納される．

1. プロジェクトを対象に静的検査を実行する
   (a) 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
   (b) Checkstyle と FindBugs を実行し，警告リストを取得する
   (c) 各警告に対し，警告を報告したツール名，検査ルール ID，警告メッセージ，警告対象の位置情報，過去のリビジョンにおける同じコードに対する同じ警告の位置情報 (発生リビジョンの位置情報) を記録する
2. 静的検査の結果を提示する
   (a) 警告リスト中の各警告に対し，発生リビジョンの位置情報が，修正の必要のない警告リスト中の警告における発生リビジョンの位置情報と一致する場合，リストから除外する
   (b) 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
3. 修正の必要のない警告を登録する
   (a) マーカーが表示された箇所のコードを開発者が確認し，修正の必要がない場合，false-positive か不要な警告かを指定する
   (b) 指定された警告の警告 ID を false-positive と不要な警告を区別してデータベースに登録する

ソースコードを変更するたびに繰り返し静的検査を実施する場合において，本ツールを利用することにより，開発者は警告を無視して良いかどうかを判断するコストを削減することが可能である．

4.2 静的検査実行機能

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．Package Explorer 中の対象プロジェクトを右クリックし，そのメニューの「Sapid」のサブメニューの「Check」を選択することによって，検査が実行される．その様子を図 6 に示し

表 1 警告情報と位置情報

警告情報			位置情報				
警告 ID	検査ルール ID	警告対象の位置 ID	位置 ID	コミット ID	ファイル名	開始行番号	警告発生リビジョンの位置 ID
1	rule1	1	1	fdc90cb1	Calc.java	9	1
2	rule1	2	2	4e13c72d	Test.java	3	1
3	rule1	3	3	92fdbe73	Test.java	3	1

た．Checkstyle と FindBugs を用いて検査を行う前に，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．

Checkstyle と FindBugs の実行後，3.3 節で述べた警告情報をデータベースの警告情報テーブルに，警告対象の位置情報をデータベースの位置情報テーブルに登録する．警告対象のファイル名と開始行，開始列，終了行，終了列は Checkstyle と FindBugs の出力から抽出する．検査ルール 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．開始行，開始列，終了行，終了列が検査ツールから得られない場合は，負の値を記録する．

過去のリビジョンにおける同じコードに対する同じ警告を追跡するために，警告が報告された各ファイルに対して git-blame を実行する．データベースには過去に検査ツールを実行した際の警告について検査ルール ID と警告対象の位置情報が登録されているので，警告対象行における git-blame による直近の修正リビジョンにおける行番号と検査ルール ID が一致する位置 ID を検出し，その位置 ID を警告が発生したリビジョンの位置 ID として登録する．

図 4 に示したコードを例にデータベースに記録される情報について説明する．それぞれのコミットは静的検査を実行した時を示しており，コミット fdc90cb1 の時 Calc.java の 9 行目に検査ルール ID rule 1 の警告が，コミット 4e13c72d の時 Test.java の 3 行目に検査ルール ID rule 1 の警告が出たとする．コミット 92fdbe73 では Test.java は変更していないため，コミット 4e13c72d の時と同様に Test.java の 3 行目に rule1 の警告が出る．この時点での警告情報と位置情報を表 1 に示す．ただし，説明に必要なデータのみを示している．また，MAFP で利用した JGit では，コミット 4e13c72d の Test.java の 2 行目から 6 行目は コミット fdc90cb1 の Calc.java の 8 行目から 12 行目が移動されたコードであることは追跡できないが，ここでは追跡できた場合の結果を示している．

警告 ID と位置 ID はデータが追加されるたびに自動でインクリメントされた値が記録される．位置情報のうちの警告発生リビジョンの位置 ID 以外は，検出された警告とその対象コードの位置がそれぞれ記録される．警告 ID 1 は新規に検出された警告であるため，警告発生リビジョンの位置 ID は 1 となり，警告 ID 2 は警告 1 の警告対象コードが Test.java の 3 行目に移動して同じ警告が検出されているため，警告発生リビジョンの位置 ID は 1 となっている．警告 ID 3 における警告発生リビジョンにおける位置 ID も同様である．

4.3 警告の提示

検査を実行してそれぞれのツールが報告した各警告について，発生リビジョンの位置 ID が修正不要として登録された警告の発生リビジョンの位置 ID のいずれとも異なればマーカーとして表示する．

検査結果を表示したツールの画面例を図 7 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 1 つの警告について警告の内容を表示し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．

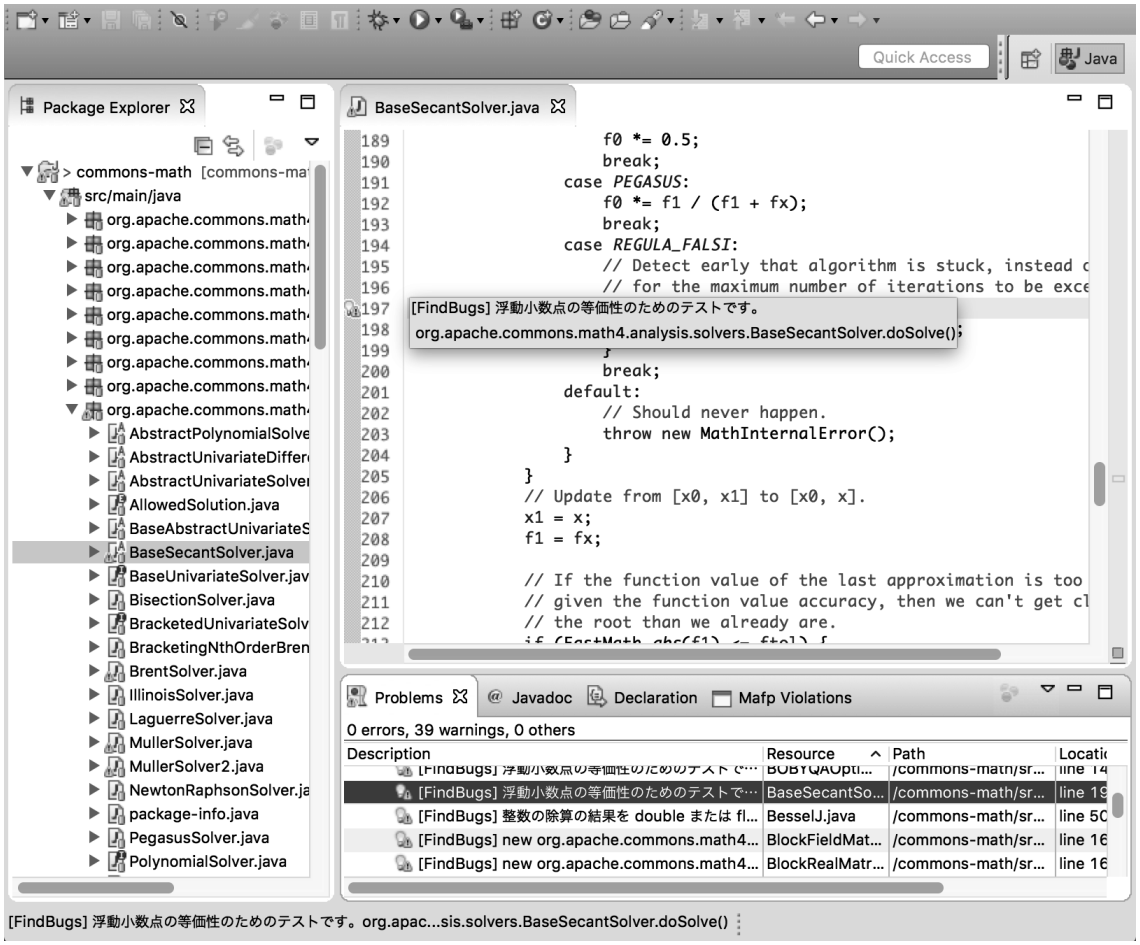

図 7 警告の提示

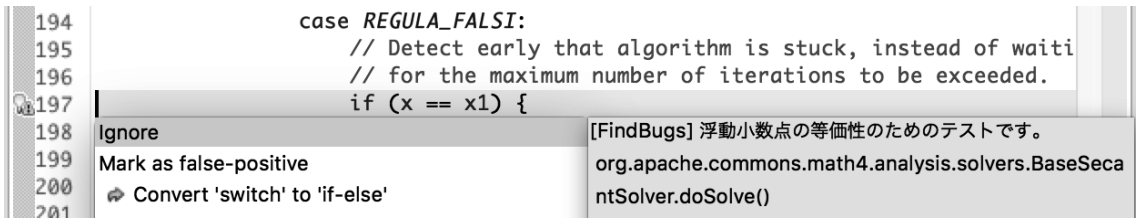

図 8 修正の不要な警告の登録

4.4 修正の必要のない警告の登録

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．開発者はマーカーで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，修正が不要な場合は，そのことを Quick Fix を呼び出してその情報を記録する．

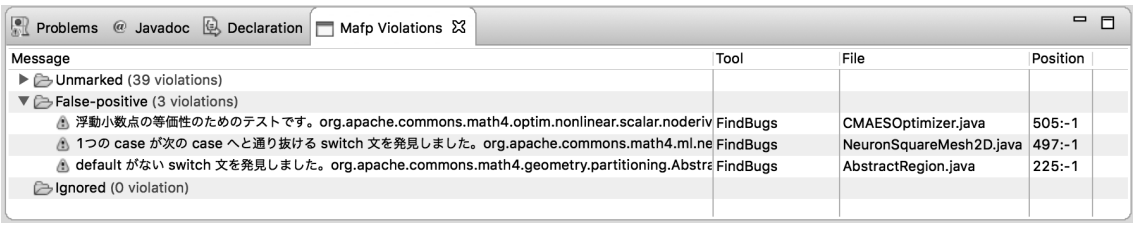

図 9　登録された警告

図 8 は，警告対象行の左側のアイコンをクリックして Quick Fix を呼び出した状態を示している．false-positive であると判断した場合には Mark as false-positive を，不要な警告であると判断した場合には Ignore を選択することによって，データベースに警告 ID が登録される．警告対象の開始列が静的検査ツールから得られている場合にはその開始列で，得られない場合には行頭で Quick Fix を呼び出すことが可能である．また，Problems ビューで警告を右クリックすることによって Mark as false-positive と Ignore を選択可能である．Mark as false-positive が選択された場合はデータベースの false-positive テーブルに，Ignore が選択された場合はデータベースの ignore テーブルに対象の警告 ID が登録される．

図 6 に示した「Sapid」メニューの「Show Latest Violations」を選択すると，図 9 に示す情報が Mafp Violations ビューに表示される．直近で検査した結果において，開発者によって false-positive や ignore として登録された警告と，マークを付けていない警告がそれぞれ，False-positive，Ignored，Unmarked のところに表示される．

たとえば，4.4 節の例で説明したように図 4 のコミット 4e13c72d 後に警告 ID 2 を false-positive として登録した場合，その後，静的検査を実行すると，コミット 92fdbe73 としてコミットされ，ツールは Test.java の 3 行目を警告として報告する．この情報は警告情報として記録され，その位置情報の発生リビジョンにおける位置情報には 1 が記録される．false-positive として警告 2 が登録されており，この検査ルール ID と発生リビジョンの位置 ID が同じであるため，この警告は表示されない．

4.5　Checkstyle のルールの指定

Checkstyle はコーディングスタイルを検査するためのツールであり，すべてのルールを検査すると一般には膨大な警告が報告される．プロジェクトによってはそれぞれコーディングスタイルが決められており，Checkstyle で検査すべきルールを設定している．MAFP では Eclipse の Preference 中の Sapid Mafp という設定項目において，Checkstyle のルールを指定することが可能である．

FindBugs では，検査するルールの指定や警告を除外するためのフィルターを指定することが可能であるが，MAFP にはそれらを設定する機能は現在はない．

5　ツールの評価

本章では，開発者が警告箇所を確認する際のコストとツールの実行時間について評価した結果および MAFP の問題点について述べる．

5.1　開発者の確認コスト削減

apache が github で公開しているプロジェクトのうち，commons-jxpath, tomcat を対象に，各リビジョンにおける警告の総数と，前リビジョンにおいて全ての警告箇所を確認して false-positive か ignore のいずれかとして登録したとし，新たに出現する警告がどの程度あるかを表 2 に示した．静的検査ツールによって提示される警告には修正が必要な警告が含まれている可能性があるが，著者は対象ソフトウェアの開発メンバーでないため，修正することは困難である．そのため，本実験ではすべて修正の必要がない警告として登録した．

表 2 では，調査対象の期間とその期間の最初のリビジョンを 1 としたリビジョン番号，ソースコー

表 2 版の更新に伴なう警告数の変化

(a) commons-jxpath

対象期間	2008/4/23 - 2012/1/21		
リビジョン	行数	総警告数	新たな警告数
1	39,815	106	-
8	37,536	105	0
20	37,557	97	0
33	37,586	95	3
49	37,204	95	1
54	37,705	96	1
77	37,941	98	2
81	38,079	98	2
85	38,189	97	0
87	38,191	96	0
100	38,251	91	0

(b) tomcat

対象期間	2015/10/26 - 2015/11/8		
リビジョン	行数	総警告数	新たな警告数
1	414,088	762	-
4	414,116	757	0
6	414,113	753	0
7	414,113	751	0
9	414,113	710	6
12	414,134	710	1
15	414,139	704	0
17	414,136	703	0
19	414,087	702	0
32	414,110	703	1
100	411,708	692	0

ドの行数 (コメントや空行を含む)，総警告数，新たな警告数を示している．総警告数と新たな警告数に変化がないリビジョンは省略している．たとえば，commons-jxpath のリビジョン 2 では，総警告数が 106 で，新たな警告数は 0 である．commons-jxpath のリビジョン 33 では，新たな警告数が 3 つあり，除外された警告が 92 であることを示している．

tomcat のリビジョン 9 における 6 個の新たな警告は，リビジョン 1 において出ていた警告と同一の警告であるが，警告とは関係のない修正を警告行に行ったことによって新たな警告として提示された．また，tomcat のリビジョン 12 における新たな警告も，リビジョン 1 において出ていた警告と同一の警告である．これは リビジョン 12 における修正において，git-blame が正しく追跡できなかったために新たな警告として提示された．

本実験では，リビジョン 1 において，すべての警告を確認し，ignore として登録したが，実際には警告箇所を確認した際に false-positive か ignore として登録するため，一度に提示される警告数が 0 となるとは限らない．一度 false-positive や ignore として登録された警告のうち多くは， git-blame 機能によって未変更行が追跡され，その警告の提示を抑止することができた．前述のように，修正内容によって再度新たな警告として提示される場合があるが，その数はそれほど多くなく，開発者が警告箇所を確認するコストを削減することができた．

5.2 実行時間

ツールの評価をするため，apache が github で公開しているプロジェクト commons-jxpath，commons-math，jmeter を対象に静的検査ツールの実行時間を計測した．Checkstyle と FindBugs の設定は，それぞれのプロジェクトで使用している設定ファイルを使用した．実行環境は CPU が Intel Core i7 4GHz，メモリが 32GB の計算機である．また，Eclipse の最大ヒープサイズは 1024MB である．対象ソフトウェアのコミット ID とファイル数，総行数，Checkstyle の警告数，FindBugs の警告数を表 3 に示す．総行数は空白とコメントを含んだ行数である．MAFP の処理は下記の通りであり，それぞれの処理にかかった時間を表 4 に示した．単位はすべて秒である．

1. ソースコードを Git リポジトリに登録 (Git 登録)
2. Checkstyle の実行 (Checkstyle)

表 3 対象ソフトウェア

ソフトウェア名	コミット ID	ファイル数	総行数	Checkstyle	FindBugs
commons-math	8ed2209	1,333	313,080	9,027	378
commons-jxpath	f1dde17	226	39,479	3,575	91
jmeter	b522cbb	1,026	194,196	1	210

表 4 実行時間

(a) commons-math

	Git 登録	Checkstyle	FindBugs	警告登録	除外	マーカー	合計
1 回目	2.7	6.7	40.2	20.3	1.7	19.2	90.8
2 回目	0.9	4.7	27.1	15.7	139.9	0.1	188.4

(b) commons-jxpath

	Git 登録	Checkstyle	FindBugs	警告登録	除外	マーカー	合計
1 回目	0.5	1.3	6.3	8.0	0.7	8.1	24.9
2 回目	0.1	0.7	3.3	5.6	15.0	0.1	24.8

(c) jmeter

	Git 登録	Checkstyle	FindBugs	警告登録	除外	マーカー	合計
1 回目	8.7	3.0	28.9	0.6	0.1	1.0	42.3
2 回目	3.6	1.4	19.5	0.4	0.1	0.1	25.1

3. FindBugs の実行 (FindBugs)
4. 警告の登録 (警告登録)
5. 過去に修正が不要と判断した警告の除外 (除外)
6. Eclipse のマーカーとして登録 (マーカー)

1 回目は Git リポジトリがない状態でかつ，修正が不要として登録された警告がないため，2 回目以降と実行時間に大きな差が出るため，1 回目と 2 回目の実行時間を示した．本実験では，1 回目のチェックを実行した後，すべての警告を ignore として登録し，再度同じソースコードに対してチェックした時間を計測した．すべての対象ソフトウェアにおいて，Git 登録処理とマーカー処理の時間が 2 回目の方が短かく，除外処理の時間が 2 回目の方が長くなっている．Checkstyle と FindBugs の処理時間が 2 回目の方が短くなっているが，その理由は不明である．

commons-math では合計実行時間が 2 回目のチェックで 3 分以上と長くなっているが，総警告数が非常に多いことが理由である．このように警告数が多いと実行時間が長くなるが，commons-jxpath や jmeter 程度であれば 1 分以内に全処理が完了しており，実用に堪え得る時間で処理できていると我々は考えている．

5.3 問題点

本節では MAFP における問題点として，ソースコードの版間の追跡に関する問題点，警告の除外，修正が不要だと登録された情報に関する問題点について述べる．

版間追跡

MAFP では各警告対象の行が最初に修正されたリビジョンにおけるファイル名と行番号を取得するために JGit の blame 機能を利用している．JGit の

blame 機能がどのように追跡しているか不明であるが，特に変更量が多い場合には，正しく追跡することができないことがある．正しく追跡ができないと，その行で出た警告は新たな警告と判断され，過去に修正が不要と登録した場合でも除外されることなく警告が出てしまう．しかし，変更量が少なければ正しく追跡することができるため，頻繁に検査を実行することによって，この問題を回避することが可能である．

警告の除外

MAFP では警告行に変更がなければ，その行を追跡し，過去に修正の必要がないとして登録されていれば，その警告を除外する．この手法の問題点として，(1) 過去に修正の必要がないとして登録しても新たな警告として提示される問題や，(2) 過去に修正の必要がないとして登録されたために，修正の必要がある警告が除外される問題がある．

(1) の問題は，たとえば変数名を変更した場合などに発生する．図 10 のコードにおいて，最後の行で「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」と警告が出たとし，この警告を false-positive として登録したとする．その後，このコード中の変数 `a` を `x` に変更した場合，警告行が変更されるため，新たな警告として同じ警告が出る．

このように過去に修正が不要だと判断して登録しても，同じ警告が出ることは，開発者の確認コストを削減する目的において問題である，しかし，この問題を解決するためには，警告対象のコード断片に関して，行だけでなく，列の情報や構文要素の種類 (変数やメソッドなど) に関する情報が必要で，行単位ではなく，構文要素単位で版間の追跡が必要となる．また，警告メッセージを解析し，警告対象における問題を識別する必要がある．これらのことから対応が困難である．

(2) の問題は図 10 を図 11 のように修正した場合に発生する．図 10 の最後の行で出た警告を false-positive として登録したとする．図 11 では `a` が `null` になる可能性があり，最後の行の警告は true-positive となる．最後の行は未修正であり，blame 機能によって，修正前のコードの最後の行が得られ，その行の警告が false-positive として登録されているため，この警告は除外される．

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

図 10 false-positive の例

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

図 11 図 10 の修正後コード

このように警告が隠蔽されてしまう可能性があり，このようなケースに対応するには修正が不要と登録した警告についても再度確認を行う必要がある．このことは開発者が警告を確認するコストを削減する目的において問題である．jmeter において明らかに false-positive だと判断できた警告 17 件に対して，true-positive となる警告があるか調査したところ，そのような警告はなかった．このようなケースがどの程度発生するかの調査は今後の課題である．

修正不要と登録された情報

MAFP では直近で検査を実行した際の検査結果しか提示することができない．そのため，過去に修正が不要と判断して登録した情報を閲覧したり，削除することができない．過去に登録した警告を確認したい場合や誤って登録した警告を削除したい場合などがあるため，登録した警告情報を閲覧・削除する機能は必要である．この機能の実装は今後の課題である．

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 [11]．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 [10]．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 [14]．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 [16]．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 [1]．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 [13]．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管理し，報告された警告について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除外する手法を提案した．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．

提案手法を Eclipse プラグインとして実装した MAFP について紹介した．提案ツールでは，静的検査ツールとして Checkstyle と FindBugs を用い，版間追跡には Git の Java API である JGit を用いて実装した．ツールの評価のため，apache プロジェクトのソフトウェアを対象に開発者が警告を確認するコストを削減可能か，MAFP の利用における実行時間が開発の妨げにならないかを確認するための実験を行った．その結果，新たな警告の数は少なく，開発者の確認コストを削減することが可能であること，検査ルールを適切に設定し，警告の数が数千程度であれば実用的な実行時間で処理可能であることを確認した．ただし，確認済みの警告を新たな警告として提示する可能性や，問題のある警告を隠蔽してしまう可能性があるという問題がある．これらの問題への対応は今後の課題である．

処理時間を短縮するためにファイル単位での検査を可能にしたり，検査ルールの設定を容易にするために，過去に ignore として登録された検査ルールを除外するなど，ツールをより良くするための機能が考えられるが，これらは今後の課題である．また，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 本研究の一部は科研費 15K15973, 24300006, 15H02683 の助成による．

参 考 文 献

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

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

[3] 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.

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

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

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

[7] Johnson, B., Song, Y., Murphy-Hill, E. and Bowdidge, R.: Why Don't Software Developers Use Static Analysis Tools to Find Bugs?, in *Proceedings of the 2013 International Conference on Software*

Engineering, 2013, pp. 672–681.
[8] Kernighan, B. W. and Ritchie, D. M.: *C Programming Language*, Prentice Hall, 1988.
[9] 桑原寛明, 渥美紀寿: 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, ソフトウェアエンジニアリングシンポジウム 2015 論文集, 2015, pp. 38–47.
[10] Muske, T. B., Baid, A. and Sanas, T.: Review Efforts Reduction by Partitioning of Static Analysis Warnings, in *Source Code Analysis and Manipulation* (*SCAM*), *2013 IEEE 13th International Working Conference on*, 2013, pp. 106–115.
[11] Nanda, M. G. and Sinha, S.: Accurate Interprocedural Null-Dereference Analysis for Java, in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 133–143.
[12] 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(2012), pp. 590–600.
[13] Sadowski, C., van Gogh, J., Jaspan, C., Söderberg, E. and Winter, C.: Tricorder: Building a Program Analysis Ecosystem, in *Proceedings of the 37th International Conference on Software Engineering*, 2015, pp. 598–608.
[14] Shen, H., Fang, J. and Zhao, J.: EFindBugs: Effective Error Ranking for FindBugs, in *Software Testing, Verification and Validation* (*ICST*), *2011 IEEE Fourth International Conference on*, 2011, pp. 299–308.
[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, in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 50–59.
[16] Tripp, O., Guarnieri, S., Pistoia, M. and Aravkin, A.: ALETHEIA: Improving the Usability of Static Security Analysis, in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 762–774.
[17] Zitser, M., Lippmann, R. and Leek, T.: Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code, in *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, 2004, pp. 97–106.

渥 美 紀 寿

2000 年名古屋大学工学部卒業，2002 年同大学大学院工学研究科博士前期課程修了，2005 年同研究科博士後期課程単位取得満期退学．2005 年南山大学数理情報学部講師，2010 年名古屋大学大学院情報科学研究科附属組込みシステム研究センター研究員，2011 年同センター特任助教，2012 年同大学情報連携統括本部情報戦略室特任助教，2016 年 6 月より京都大学学術情報メディアセンター助教．博士 (工学)．プログラム解析，ソフトウェア保守，ソフトウェア再利用などの研究に従事．日本ソフトウェア科学会，電子情報通信学会，IEEE CS，ACM 各会員．

桑 原 寛 明

2001 年名古屋大学工学部卒業．2003 年同大学大学院工学研究科博士前期課程修了，同大学院情報科学研究科博士後期課程へ進学．2007 年立命館大学情報理工学部助教．2016 年より南山大学情報センター講師．博士 (情報科学)．ソフトウェア工学，形式手法，プロセス代数，型理論，プログラム解析などの研究に従事．情報処理学会，IEEE CS 各会員．