ソフトウェアテストにおける静的解析ツールの段階的適用による不具合修正工数の更なる低減
―民生用音響・映像機器向け組み込みソフトウェア開発への QAC の段階的適用と その実証評価―

Further Reduction of Debugging Cost in Software Testing through Progressive Use of Static Code Analysis Tools
- Experimental Evaluation of Progressive Use of QAC in Developing Embedded Software for AV Appliances -

鶴田 雅明（Masaaki TSURUTA）[1]・大平 雅雄（Masao OHIRA）[2]
門田 暁人（Akito MONDEN）[3]・松本 健一（Kenichi MATSUMOTO）[4]
[1]奈良先端科学技術大学院大学情報科学研究科 博士後期課程・[2]和歌山大学システム工学部情報通信システム学科 准教授・[3]岡山大学工学部情報系学科 教授・[4]奈良先端科学技術大学院大学情報科学研究科 教授

［Abstract］
This paper proposes a new approach to reduce the cost of bug fixes by using static code analysis (SCA) tools progressively during software testing. The paper sets up four testing phases just after coding event in the whole software development process and the proposed approach defines evaluation criteria for SCA to be used to generate appropriate warnings for each testing phase, and provides a useful interface between a SCA tool and incomplete program code using a code wrapping technique. The approach enables SCA tools to be applied even at the early phase of testing, like unit testing and preliminary integration testing, which the tools have not been able to correspond to because of incomplete expression of code programs. The experimental results from 95 projects developing embedded software for AV appliances showed that progressive use of QAC, one of the most well-known SCA tools, reduced the cost of bug fixes during software testing to about one-third of conventional cost.

［キーワード］
ソフトウェアコードメトリクス，ソフトウェアテスト，組み込みソフトウェア，商用ソフトウェア開発での評価

１． はじめに

今日、社会のあらゆる場面で情報システムが利用され、高い付加価値を生みだす技術基盤の一つとして、社会経済の活性化に大きく寄与している。その機能の多くは、ソフトウェアによって実現されており、ネットワークやハードウェアの性能を最大限に引き出すと共に、初心者にもわかりやすい表示や操作体系を実現することで、情報システムの利用目的、利用環境、そして、利用者層の拡大をもたらしている。

情報システムとそれに基づく高度情報化社会を支えるソフトウェアであるが、ソフトウェアの不具合が原因とされる大規模情報システムの故障や障害は、現在でも数多く報告されている。ソフトウェアの品質が、個人のみならず社会に大きな影響を与えると強く認識されており[13]、その社会的損失は、国内総生産の 0.6%、日本国内だけで年間 3.6 兆円に達するとの試算もある[14]。ソフトウェアの不具合が作り込まれることを未然に防ぐ、あるいは、作り込まれた不具合をできるだけ早期に発見する技術の開発は、高度情報化社会の更なる発展に不可欠であり、急務である。

ソフトウェアの不具合を早期に発見する技術の一つとして「静的コード解析ツール（Static Code Analysis Tool）」が注目を集めている。同ツールは、主にソースコードを対象（ツールへの入力）として、コード全体としての構造や特性だけでなく、コードに含まれる各命令の特性や命令間の関係性なども計測、解析し、不具合につながる可能性の高い状態や不具合が潜んでいると推測されるコード断片（ソースコードの一部．Code Fragment）を指摘する警告（Warning）を出力する[2]。警告は、コーディング規約からの逸脱から安全性に関する違反まで多岐にわたる。ツールによる解析は、人手によるコードレビューなどに比べて見逃しや漏れが少なく、かつ、高速であることから、ソフトウェアのテストや不具合修正、更には、保守の工数低減に大きく貢献するとされている[3]。ソフトウェアセキュリティの分野においても、ソフトウェア脆弱性の早期検知に向けての活用が期待されている[1]。

ソフトウェアの不具合を早期に発見し、開発・保守工数の削減にも役立つとして、学術的には大いに注目されている静的コード解析ツールであるが、その一方で、具体的な適用方法や適用効果についての実証的研究が少な

く、産業界（ソフトウェア開発の現場）への導入は必ずしも進んでいない。また、その適用には多くの工数が必要になるとの報告もあり[6]、適用コストを低減する工夫も必要と考えられる。

本論文では、C／C++プログラムを対象とした代表的な静的コード解析ツールの一つである QAC[12]を、ある企業における民生用音響・映像機器向け組み込みソフトウェア開発に全社的に導入するにあたり明らかとなった技術的課題を整理すると共に、その対策を提案し、いくつかの適用実験を通じて不具合の早期発見と修正工数の更なる低減という観点からその効果を示す。提案手法では、解析に用いるコードメトリクスの種類や数を、テストプロセスが進むにつれて徐々に増やしていく。この独創的なアプローチにより、「偽陽性警告（False-positive Warning）の増大」と「テスト初期段階でのツール適用不能」という２つの技術的課題が解決され、不具合修正工数の更なる低減も可能となることを示す。なお、提案手法は、QAC を対象としたものであるが、「解析に用いるメトリクスを段階的に増やす」というアプローチには高い汎用性があり、QAC 以外の多くの静的コード解析ツールに転用，応用できる。また、プログラム言語（解析対象とするソースコードの記述言語）への依存性も低い。これは、同アプローチにおいて、どのメトリクスをテストプロセスのいつから適用するのかは、メトリクスの実装ではなく定義によって定まるためである。

以降、２章では、ソフトウェアの静的コード解析ツールの概要を紹介し、３章で、同ツールをソフトウェア開発現場に適用する際の技術的課題を、既存研究を交えて整理した上で、４章では、代表的な３つの関連研究を本研究と比較する。５章では、ツール適用における技術的課題を解決するための手法を提案し、６章で、提案手法の有効性を示す適用実験とその結果について述べる。最後に７章で全体をまとめる。

２． 静的コード解析ツール

静的コード解析ツールは、ソースコードを入力とし、その解析結果を出力するソフトウェアツールである。ソースコードからコンパイラによって生成されるオブジェクトコードやバイナリコードを解析対象とするツールも存在するが、いずれにおいても、解析において、ソースコードや対応する実行可能コードが実行されることはない。古くは、「コンパイラではチェックされないが不具合の原因となるような曖昧な記述」を警告するツール Lint が知られている[5]。最近では、より高度な解析を行うツールが、商用ソフトウェアとして開発され、また、オープンソースソフトウェアとしても数多く公開されている。いくつものテスト手法を組み合わせても見逃されてしまうような不具合も検知される場合があることから、極めて高い信頼性が求められるソフトウェアの開発においては、同ツールによる信頼性検証が有効であるとの指摘もある[2]。

QAC は、商用の静的コード解析ツールとして最も普及しているものの一つである[12]。C もしくは C++で記述されたソースコードを対象として、コードの複雑さをはじめとする特性を計測するための定量的尺度（コードメトリクス）が多数用意されており、コード実行順序（制御の流れ）についての詳細な解析と合わせて、ソフトウェア品質の向上に寄与するツールとされている[7]。特に、ソースコード上の文法誤りや不具合の検出能力が高く、ソースコードの構成管理ツールとの親和性も高いとされている。本論文のように、不具合修正工数の低減を目的として、企業における商用ソフトウェア開発での活用を目指す場合には、研究対象とすべき有力な候補の一つと言える。なお、一般に、ソフトウェアの不具合は、その発生原因や症状が多岐にわたるため、静的コード解析ツールの多くは、多種多様なメトリクスでソースコードを多面的に解析する、いわゆる「メトリクス集積型」に分類される。QAC はその代表格で、広く知られているメトリクス、他の解析ツールでも採用されているメトリクスの多くによる解析が可能である。

QAC が提供するコードメトリクスのうち、本論文で対象とするのは、19 種類の関数単位メトリクス（Function-based Metrics）と 33 種類のファイル単位メトリクス（File-based Metrics）の計 52 種類である。各メトリクスの名称や計測内容等は，表-1 と共に５章にて後述するが、関数単位メトリクスには、（変数宣言文などを除いた）実行行数、関数呼び出し数、制御構造における最大入れ子レベル、MaCabe が提案するサイクロマティック数[8]などが含まれる。また、ファイル単位メトリクスには、静的変数数、機能結合度の推定値、Halsted が提案する一連のソフトウェアサイエンス尺度[4]が含まれる。

３． ツール適用における技術的課題

民生用音響・映像機器向け組み込みソフトウェア開発のいくつかに静的コード解析ツールを適用する中で、同ツールの普及を妨げる最も重大な技術的課題として挙げられたのは、「偽陽性警告（False-positive Warning）の大量生成」である。多くの静的コード解析ツールでは、その利用目的に鑑み、不具合が潜んでいると推測されるコー

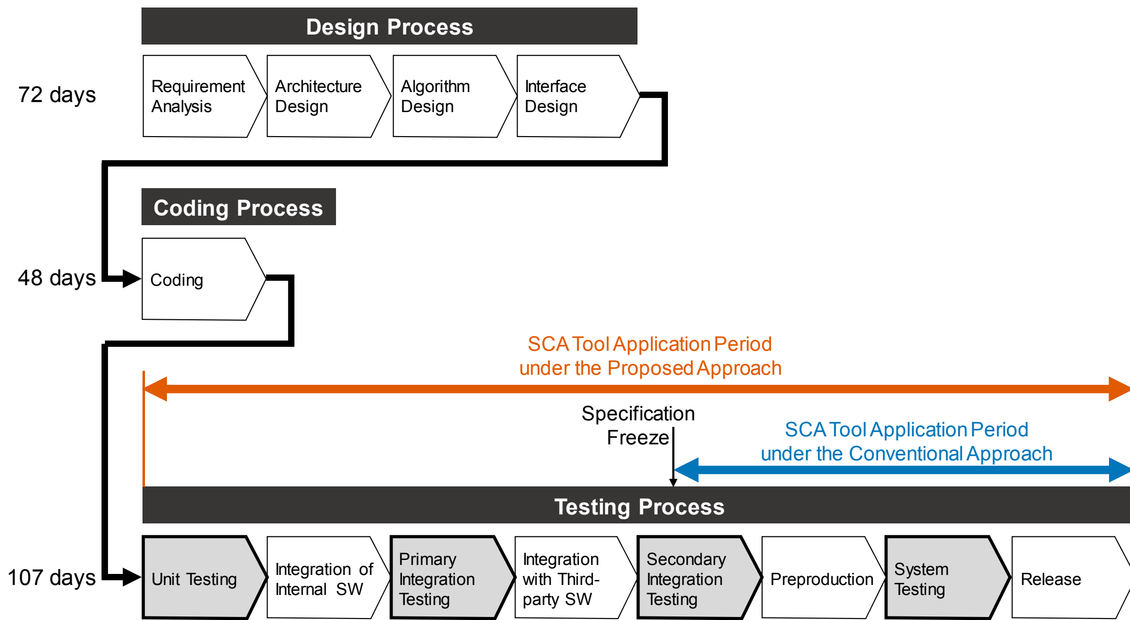


図-1 民生用音声・映像機器向け組み込みソフトウェアの開発工程例と静的コード解析ツール適用期間

ド箇所の指摘漏れ（False-negative）を極力抑えるため、不具合が存在するかどうかの判断基準が比較的低く設定されている。結果として多数の偽陽性警告が生成される傾向にあり、それぞれの警告が真に不具合につながるものなのかどうか、その評価や判断に多くの工数を要することになる。偽陽性警告率は、ツールとその適用対象の組み合わせによって異なるが、民生用デジタル機器向けソフトウェアの場合、偽陽性警告率は低いものでも 15～35%であったと報告されている[6]。

Google 社では、静的コード解析ツールの社内適用基準として、「偽陽性警告率が 10%以下であること」と定めている[10]。偽陽性警告率を 10%以下に抑えることが出来るかどうかが、静的コード解析ツールを産業界（ソフトウェア開発の現場）に普及される一つの目安と考えることができる。

ツールを適用する中で明らかになったもう一つの技術的課題は、「テスト初期段階でのツール適用不能」である。テストの初期段階では、ソースコードに多くの不具合が含まれており、また、不完全なコード断片（Incomplete Code Fragment）も数多く存在する。それらが相互干渉し、詳細な解析を阻むため、多数の偽陽性警告が出力される。あまりにも多くの警告が出力されるため、ツールが異常終了し、解析結果が得られない場合もある。そうした事態を避けるため、従来、テスト対象ソフトウェアに対する仕様変更の受入れの終了を意味する仕様凍結（Specification Freeze）以降、つまり、全てのソースコードが結合され，システムとして動作可能となるテスト工程の後半のみに同ツールは適用されてきた。

テストの初期段階（単体テスト）で発見される不具合は全体の 16%とされている[14]。残る 84%もの不具合は、テスト工程の後半で発見されることになるが、一般に、不具合は、作り込まれてから時間が経過するほど、その影響がコード全体に拡散するため、その除去にはより多くの工数が必要となる。テスト作業においても同様であり、不具合を出来るだけ初期に発見し除去することで、テスト工数が大幅に削減されると共に、リリース後に故障や障害を引き起こす不具合の見逃しも少なくなることが期待される。しかし、静的コード解析ツールの適用開始を仕様凍結以降としたままでは、いくら偽陽性警告率を低く抑えても、その効果は極めて限定的なものとなってしまう。

4. 関連研究

Wichmann らは、静的コード解析の重要性を企業におけるソフトウェア開発の観点から明らかにしている[11]。彼らは、静的コード解析は、プログラム実行を伴う動的解析を補完するものであり、対象ソフトウェアの品質をより強固に保証するためのツールの 1 つとして、確固たる地位を築きつつあると指摘している。本論文と比較す

ると、企業における静的コード解析ツールの適用例を紹介している点は共通している。ただし、解析対象としているのは、高い安全性が求められるクリティカルソフトウェアのみであり、解析ツールの新しい適用手法の提案などは行われていない。これに対し本論文では、より広範に、組み込みソフトウェアを解析対象とし、ソフトウェアテストにおいて不具合修正工数を更に低減する具体的な手法を提案している。

Baca は、多くの開発者が、ソフトウェアの脆弱性を早期に検知する道具として静的コード解析ツールを使い始めているとしている[1]。本論文と比較すると、静的コード解析ツールが非常に多くの警告を生成するため、その理解や評価に多くの工数を要するという問題も指摘している点、そして、その解決のためには、不具合修正につながる警告のみを選び出して開発者に提示すべきとしている点は共通している。ただし、修正対象として注目しているのは、脆弱性に関する不具合のみであり、それ以外の不具合やそれら修正に役立つ他の静的コード解析ツールへの言及はない。これに対して本論文では、不具合全般を修正対象としており、提案手法も他の静的コード解析ツールへの転用、応用が容易である。

Plocsh らは、Evaluation Method for Internal Software Quality (EMISQ)と呼ばれる方法論に基づき、開発中のソースコードの品質を系統的に査定し改善するためのアプローチのひとつとして Code Quality Monitoring Method (CQMM)を提案している[9]。EMISQ でも CQMM でも、コード品質の査定と改善は、静的コード解析ツールの出力に基づいて行われる。本論文と比較すると、静的コード解析ツールの継続的利用が CQMM の一つの特徴とされており、その点は共通している。ただし、大量に生成される警告やそうした警告への対処法についての言及はない。これに対して本論文では、不具合修正につながらない警告を「偽陽性警告」として取り除く具体的な手法を提案し、提案手法の適用を容易にするソフトウェアインタフェース SCA Wrapper を開発している。

５． 提案手法：静的コード解析ツールの段階的適用

５．１ 民生用音響・映像機器向け組み込みソフトウェアの開発工程

民生用の音響・映像機器み組み込まれるソフトウェアの、ある企業における典型的な開発工程例を図-1 に示す。同工程は 3 つのプロセス：「設計プロセス（Design Process）」、「実装プロセス（Coding Process）」、「テストプロセス（Testing Process）」で構成される。設計プロセスは、更に 4 つの設計副工程に分かれ、その完了には平均 72 日を要する。同じく、実装プロセスの完了には平均 48 日を要する。そして、テストプロセスは、実施の目的や観点が異なる 4 つのテスト副工程で構成されており、その完了には平均 107 日を要する。これら平均日数は、６章で後述する 95 プロジェクトにおける値である。4 つのテスト副工程でのそれぞれの目的や観点は次の通りである。

（T1）単体テスト（Unit Testing）：テスト対象ソフトウェアの構成要素のうち、自組織で開発されたモジュール（分割コンパイルや開発管理の単位となる一連のコード群）それぞれについて、ソースコードと要求仕様の間に差異がないか評価する。

（T2）一次結合テスト（Primary Integration Testing）：同じく自組織で開発されたモジュールを結合し、モジュール間でのデータのやり取りや制御の流れと要求仕様の間に差異がないか評価する。「全機能テスト（Full Function Testiing）」とも呼ばれる。

（T3）二次結合テスト（Secondary Integration Testing）：協力企業など組織外の第三者によって開発されたモジュールを自組織で開発されたモジュールに追加結合し、それら全体について、モジュール間でのデータのやり取りや制御の流れと要求仕様の間に差異がないか評価する。「製品候補版テスト（Release Candidate Testing）」とも呼ばれる。なお、本副工程の前半、例えば、実施予定期間の約 3 分の 1 が経過した時点で、テスト対象ソフトウェアに対する仕様変更の受入れは終了し、仕様凍結（Specification Freeze）が行われる。従来、静的コード解析ツールの適用期間は、この仕様凍結以降リリースまでとされている。本論文の提案手法は、この適用期間を、テストプロセス全体に拡大しようとするものである。

（T4）システムテスト（System Testing）：試作された機器に全モジュールを組み込んだ状態で、大量生産における生産性と適応性、および、市場におけるセキュリティ、安全性、保守性の観点から、ソフトウェアシステムとして評価する。更に、それら全モジュールが、開発規模、開発工数といった開発管理指標を満足するかどうかを明らかにする。「製品最終版テスト（Golden Master Testing）」とも呼ばれる。

５．２ テスト副工程における静的コード解析ツールの適用基準

静的コード解析ツールによる偽陽性警告率を低減し、かつ、ソフトウェアテスト工程の初期から適用可能とするため、ここでは、静的コード解析ツールをソフトウェアテスト工程において段階的に適用する方式を提案する。具体的には、C／C++プログラムを対象とした代表的な静的コード解析ツールの一つであるQACの適用において、テストの初期段階では、コード特性の計測に用いる定量的尺度（ソフトウェアコードメトリクス）の数を制限し、テスト作業の進展に伴って、その数を４段階に分けて増やしていく。

前述の通り、4 つのテスト副工程は、それぞれ異なる目的や観点を有しており、テスト対象となるソースコードの状態や置かれている環境も異なる。特に、単体テストや一次結合テストにおいては、ソースコードの状態は、その最終完成状態からはほど遠く、ある意味、不完全なままでテストが行われているとも言える。静的コード解析ツールに実装されているコードメトリクスの中には、不完全なコードに適用すると偽陽性警告を多数生成するものが多い。そこで、提案手法では、静的コード解析ツールに実装されているコードメトリクスを、テストプロセスの初期段階から全て適用するのではなく、テスト副工程毎にその適用基準を定め、適用するコードメトリクスの種類や数を、テストプロセスが進むにつれて増やしていくこととする。具体的な適用基準は次の通りである。

（C1）単体テストにおける適用基準：（自社で開発された）モジュール単位において、不具合の存在を直接的に示すメトリクス、もしくは、不具合の存在を強く示唆するメトリクスのみを適用する。

（C2）一次結合テストにおける適用基準：（C1）に該当するメトリクスに加え、次の 3 種類のメトリクスを適用する。
- （C2-1） 不具合の存在を直接的に示す指標であり、コーディング規約からの逸脱を計測するメトリクス。
- （C2-2） コードの安定性をモジュールレベルで示す指標であり、データもしくは制御の複雑さを計測するメトリクス。
- （C2-3） モジュールレベルでのプロダクト管理指標であり、コードの外的特徴を計測するメトリクス。

（C3）二次結合テストにおける適用基準：（C2）に該当するメトリクスに加え、テスト対象ソフトウェアを構成する全モジュールをソフトウェアシステムと捉えて計測する次の 3 種類のメトリクスを適用する。
- （C3-1） コードの安定性をシステムレベルで示す指標であり、データもしくは制御の複雑さを計測するメトリクス。
- （C3-2） システムレベルでのテストプロセス管理指標であり、不具合数を推定するメトリクス。
- （C3-3） システムレベルでの出荷後管理指標であり、コードの外的特徴を計測するメトリクス。

（C4）システムテストにおける適用基準：静的コード解析ツールに実装されている全てのコードメトリクスを適用する。特に、システムテストにおける評価観点である「開発規模、開発工数」、「大量生産における生産性と適応性」、および、「市場におけるセキュリティ、安全性、保守性」を、ソフトウェアシステムとして計測するメトリクス。

５．３ 静的コード解析ツールの段階的適用のためのソフトウェアインタフェース：SCA Wrapper

テストプロセスにおいて、静的コード解析ツールをソースコードに段階的に適用するために，著者らが開発したソフトウェアインタフェース SCA Wrapper の概要を図-2 に示す。SCA Wrapper は、５章２節で示した適用基準に基づくメトリクスの段階的適用、不具合修正に役立つ警告の提示など、提案手法で必要となる処理を人手に代わって行う。したがって、同図は、提案手法における処理とデータの流れを示す図でもある。SCA Wrapper は、2 つの機能「警告フィルタリング（Warning Filtering）」と「警告トラッキング（Warning Tracking）」とにより、テストプロセスの初期段階においても、静的コード解析ツールの適用を容易にし、ソフトウェアの設計・実装担当者（Design/Coding Team）とテスト・不具合修正担当者（Testing/Debugging Team）の双方に、対象ソースコードの品質向上に役立つ情報を提供する。

５．３．１ 警告フィルタリング

静的コード解析ツールが出力する警告中から、不具合修正につながらない偽陽性警告を削除する。５章２節で示した「静的コード解析ツール適用基準」を用いることで、偽陽性警告数は減少するが、それでもなお、不具合修正につながらない警告が多数残ってしまう場合がある。特に、テストプロセスの初期段階では、コード中に多

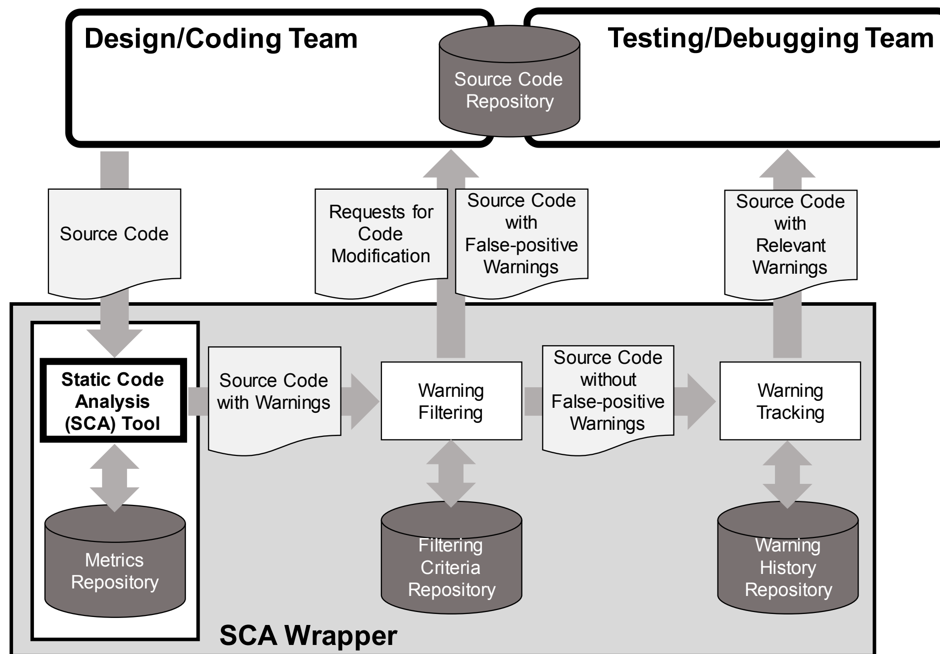


図-2 静的コード解析ツールの段階的適用ソフトウェアインタフェース「SCA Wapper」の概要

数の不具合が含まれている。また、局所的には正しいが、コード全体から見ると必ずしも正しくない、情報が欠落している、改良の余地のある、といった「不完全なコード断片」が存在することが多い。そうした不具合や不完全なコードは、ツールによる詳細な解析を阻み、多くの偽陽性警告につながる。テストプロセスの初期段階に見られる典型的な不完全なコードとしては、次のようなものがある。

① 関数宣言において仮引数の並びがない
② 関数宣言において Int 型指定子が省略されている
③ 引数を持たない関数が定義されている
④ C++言語において、関数 printf が事前に宣言されていない
⑤ 戻り値を持たない Printf 文
⑥ 文字列リテラル
⑦ 関数宣言がされていない
⑧ 関数 main の戻り値を割り当てる文がない

不完全なコードとそれに対し QAC が生成する警告の例を図-3 に示す.

本機能では、不完全なコードが原因で生成された警告も偽陽性警告と見なす。５章２節で示した「静的コード解析ツール適用基準」に加えて、不完全なコードの判定基準は、フィルタリング基準リポジトリ（Filtering Criteria Repository）で維持管理される。対象ソースコードとそれに対して静的コード解析ツールが生成した警告が入力として与えられると、次の 3 つが出力される。

(O1) ソースコードとそれに対する偽陽性ではない警告（Source Code without False-positive Warnings）
　厳密には、本機能によって偽陽性とは判定されなかった一連の警告と、対応するソースコードの組である。SCA Wrapper が提供するもう一つの機能「警告トラッキング」への入力となる。

(O2) ソースコード修正要求（Requests for Code Modification）
　不完全なコードは、それ自体に対する偽陽性警告の原因となるだけでなく、それ以外のコードに対する警告を隠して（生成されなくして）しまう可能性がある。警告が隠されてしまうと、テストや不具合修正において手戻りが発生することになる。そこで、静的コード解析ツールが解析に失敗し偽陽性警告を生成す

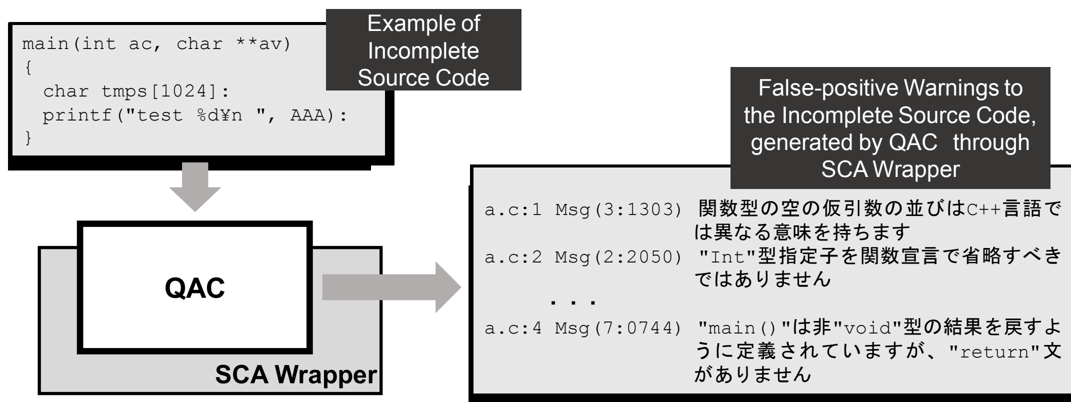


図-3 不完全なコードとそれに対し SCA Wrapper を介することで得られる偽陽性警告の例

る原因となった不完全なコードをリストアップし、設計・実装担当者に示し、その修正、除去を依頼する。

(O3) ソースコードとそれに対する偽陽性警告（Source Code with False-positive Warnings）
厳密には、本機能によって偽陽性と判定された一連の警告と、対応するソースコードの組である。本機能による出力「ソースコード修正要求」の補助資料として、設計・実装担当者に提供される。また、SCA Wrapperが提供するもう一つの機能「警告トラッキング」の出力「ソースコードと不具合修正に役立つ警告（Source Code with Relevant Warning)」の補助資料として、テスト・不具合修正担当者に提供される。

５．３．２ 警告トラッキング

不具合修正に役立つ警告が偽陰性（False-negative）として削除されてしまうことを防ぐ。本機能への入力は、前述の通り、SCA Wrapper のもう一つの機能「警告フィルタリング」の出力の一つ「ソースコードとそれに対する偽陽性ではない警告」である。また、対象ソースコードの旧バージョンに対して生成された一連の警告が、警告履歴リポジトリ（Warning History Repository）として維持管理されている。

まず、対象ソースコードについて、現バージョンと旧バージョンとを比較し、類似するコード断片を要素とする「類似コード集合」を作成する。そして、ある類似コード集合の要素である旧バージョンのコード断片に対して不具合修正につながる警告が生成されていたにもかかわらず、その集合の別の要素である現バージョンのコード断片に対して同様の警告が生成されていない、あるいは、SCA Wrapper のもう一つの機能「警告フィルタリング」によって削除されてしまっている場合には、その警告を、不具合修正に役立つ警告として追加、あるいは、復活させる。得られた結果は、「ソースコードと不具合修正に役立つ警告（Source Code with Relevant Warning)」として、テスト・不具合修正担当者に提供される。

６． 実プロジェクトデータによる適用実験

６．１ 概要

ある企業において 2004 年から 2007 年にかけて実施された民生用音響・映像機器向け組み込みソフトウェア開発プロジェクト 95 件に提案手法を適用し、その有効性を評価する実験（適用実験）を行った。評価のため、次に示す２つの仮説（Research Question）を設定した。

(RQ1) 提案手法によって静的コード解析ツールをテストプロセスに適用すると、同ツールが適用可能であった二次結合テストとシステムテストではもちろんのこと、同ツールが適用できなかった単体テストや一次結合テストにおいても、偽陽性警告率が 10%以下となる。（前述の通り、偽陽性警告率 10%以下は、静的コード解析ツールを社内適用するかどうかを判断するために、Google 社が定める基準[10]であり、同ツールを産業界（ソフトウェア開発の現場）に普及されるための一つの目安と考える。）

（RQ2）提案手法によって静的コード解析ツールをテストプロセスの初期段階にも適用可能となったことで、テストプロセスにおける不具合修正工数が、QAC を従来法で適用した場合に比べて更に減少する。（対象企業における民生用音響・映像機器向け組み込みソフトウェア開発のテストプロセスで、QAC を従来法で適用した場合の不具合修正工数の平均は、2003 年から 2004 年にかけて実施された 50 プロジェクトにおいて 4.50 人日であった。）

適用実験では、静的コード解析ツールとして QAC を用い、警告の生成には、QAC により計測可能なコードメトリクスのうち、関数単位メトリクス（Function-based Metrics）19 種類、ファイル単位メトリクス（File-based Metrics）33 種類の計 52 種類を用いることとし、５章２節で示した基準に照らし、適用するテスト副工程を決定した。メトリクスと適用するテスト副工程の対応を表-1 に示す。例えば、全ての関数単位メトリクスは、一次結合テスト以降の副工程で適用されるが、単体テストで適用されるのはそのうち 6 種類のみである。一方、ファイル単位メトリクスは、単体テストからシステムテストへの副工程が進行するにつれて、その適用数が 2、5、17、33 と段階的に増加していくことが分かる。

メトリクスの段階的適用や生成された警告のフィルタリング・トラッキングなど、提案手法で必要となる処理は、人手ではなく SCA Wrapper を用いて行った。開発者（設計・実装担当者、テスト・不具合修正担当者）からみると、提案手法によって QAC の適用範囲は広がるが、その適用工数は従来とほとんど変わらず、不具合修正工数の増減を，そのまま提案手法の効果と捉えることができる（仮説 RQ2 の検証に用いることができる）。

2 つの仮説（RQ1）（RQ2）を検証するために提案手法を適用した 95 プロジェクトのうち、公開可能な 11 プロジェクトの概要を表-2 に示す。いずれも、民生用の音響・映像機器に組み込まれるソフトウェアを開発するプロジェクトであり、プログラミング言語は C または C++、プログラムサイズ（ソースコードの行数）の平均は 398 KLOC　（Kilo Lines of Code）、不具合密度（1 KLOC あたりの不具合数）の平均は 2.82 であった。

６．２　適用結果

（RQ1）テストプロセスの初期段階での偽陽性警告率

適用対象 95 プロジェクトの一次結合テストにおいて QAC によって生成され、提案手法を通じて開発者に示された警告の内訳を図-4 に示す。同図より、警告のうち設計プロセスに起因する不具合の修正につながったものが 76%、同じく実装プロセスに起因する不具合の修正につながったものが 20%であった。そして、偽陽性警告（False-positive Warnings）は、基準とした 10%を大きく下回る 4%に過ぎなかった。

QAC が従来から適用可能であった二次結合テストにおいても、偽陽性警告率が低下したことを示す結果を図-5 に示す。同図は、表-2 に示した 11 プロジェクトのうち、二次結合テストに関する詳細なデータが入手可能であったプロジェクト#5 において、提案手法によって二次結合テストで生成され、開発者に示された警告の内訳である。ここでも、偽陽性警告率は 10.1%に留まり、基準とした 10%がほぼ達成されたことになる。なお、偽陽性ではなく不具合修正につながった警告のうち、すぐに修正すべき不具合に関する警告は全体の 1.7%だけで、全体の 88.2%は、市場出荷後、もしくは、次期開発で修正すればよい不具合（Deferred bug）に関する警告であった。

（RQ2）テストプロセスにおける不具合修正工数の減少

表-2 の最右列に、テストプロセスでの不具合修正工数（人日）の平均を示す。表-2 で示した 11 プロジェクトでは 0.93、提案手法を適用した全 95 プロジェクトでは 1.50 であった。対象企業において民生用音響・映像機器向け組み込みソフトウェアを開発し、かつ、テストにおいて QAC を従来法で適用した 50 プロジェクトでの平均 4.50 と比較すると、不具合修正工数はおよそ 3 分の 1 に減少したことになる。（提案手法による不具合修正工数の減少率は 0.67 となる。なお、提案手法の評価においては、平均だけでなく標準偏差も重要な指標の一つである。ただし、提案手法を適用した 95 プロジェクトについても，また，提案手法を適用していない 50 プロジェクトについても、標準偏差を算出するに必要なデータは、その機密性等を理由に開示されていない。）

図-4 に示したように、提案手法によって QAC を一次結合テストに適用した結果得られた警告の大半は、実装プロセスに起因する不具合ではなく、設計プロセスに起因する不具合の修正につながるものであった。一次結合テストで発見、修正されなかった場合、影響する範囲は拡大し、より多くの修正工数が必要となるのは後者である。不具合修正工数の大幅な減少には、こうした点も貢献していると考えられる。なお、提案手法を適用した 95 プロジェクトと適用しなかった 50 プロジェクト、それらの実施期間において、開発環境に大きな変化はなかった。

表-1 QACにより計測可能なコードメトリクスと適用するテスト副工程

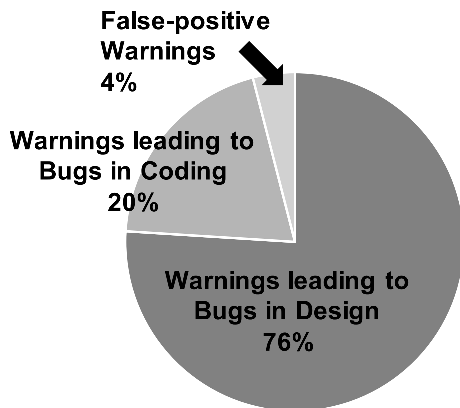
Abbr.	Metrics	Unit Testing	Preliminary Integration Testing	Secondary Integration Testing	System Testing
Function-based Metrics					
STFIL	Name of File	○	○	○	○
STNAM	Name of Function	○	○	○	○
STBAK	# of Backward Jumps	○	○	○	○
STELF	# of Dangling else-if's	○	○	○	○
STUNV	# of Unused Variables	○	○	○	○
STLCT	# of Local Variables Declared	○	○	○	○
STCYC	Cyclomatic Complexity	×	○	○	○
STMCC	Myer's Interval	×	○	○	○
STMIF	Maximum Nesting of Control Structures	×	○	○	○
STSUB	# of Function Calls	×	○	○	○
STAKI	Akiyama's Criterion	×	○	○	○
STLIN	# of Maintainable Code Lines	×	○	○	○
STXLN	# of Executable Lines	×	○	○	○
STGTO	# of goto's	×	○	○	○
STKNT	Knot Count	×	○	○	○
STKDN	Knot Density	×	○	○	○
STPTH	Estimated Static Path Count	×	○	○	○
STPDN	Path Density	×	○	○	○
STPBG	Path-based Residual Bug Estimate	×	○	○	○
File-based Metrics					
STFIL	Name of File	○	○	○	○
STNAM	Name of Function	○	○	○	○
STSCT	# of Static Variables	×	○	○	○
STECT	# of External Variables	×	○	○	○
STDIF	Halstead Program Dificulty	×	○	○	○
STNTB	# of Header Lines for Measurement (Non-token-based)	×	×	○	○
STFNC	# of Function Difinitions	×	×	○	○
STFCO	Estimated Function Coupling	×	×	○	○
STTKB	# of Header Lines for Measurement (Token-based)	×	×	○	○
STTOT	Total # of Tokens	×	×	○	○
STZIP	Estimated Program Length with Zipf 's Law (Token-based)	×	×	○	○
STHAL	Halstead Program Length	×	×	○	○
STSHN	Shannon Information Content	×	×	○	○
STMOB	Code Mobility	×	×	○	○
STBUG	Halstead Residual Bugs	×	×	○	○
STCOM	# of Header Lines for Measurement (Comment-based)	×	×	○	○
STCDN	Comment Density	×	×	○	○
STVAR	Halstead total # of Operands	×	×	×	○
STOPT	Halstead Distinct Operators	×	×	×	○
STOPN	Halstead Distinct Operands	×	×	×	○
STEFF	Halstead Program Effort	×	×	×	○
STVOL	Halstead Program Volume	×	×	×	○
STDEV	Halstead Development Time	×	×	×	○
STPRT	Estimated Porting Time	×	×	×	○
STTPP	Total Unpreprocessed Source Lines	×	×	×	○
STTLN	Toral Preprocessed Source Lines	×	×	×	○
STBCS	# of Header Lines for COCOMO	×	×	×	○
STBMO	COCOMO Organic Programmer Months	×	×	×	○
STBMS	COCOMO Semi-detached Programmer Months	×	×	×	○
STBME	COCOMO Embedded Programmer Months	×	×	×	○
STTDO	COCOMO Organic Total Months	×	×	×	○
STTDS	COCOMO Semi-detached Total Months	×	×	×	○
STTDE	COCOMO Embedded Total Months	×	×	×	○

Note: ○: Apply, ×: Not Apply,

表-2 提案手法を適用した 95 プロジェクトのうち 11 プロジェクトの概要

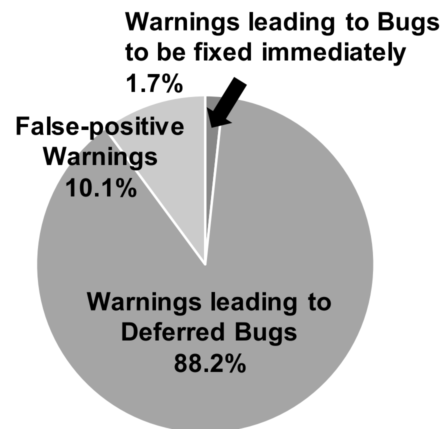
ID	Application Area	Language	Program Size (KLOC)	Bug Density (# of Bugs / KLOC)	Average Effort for Debugging (Person-Day / Bug)
1	FE	C	1199	0.35	0.24
2	BE	C	60	4.03	0.17
3	FE	C	153	0.43	2.80
4	FE, PC	C	540	1.73	0.09
5	FE	C / C++	307	8.11	0.08 / 0.03
6	BE	C	87	2.46	0.39
7	FE	C++	922	0.11	0.63
8	FE, BE, AVC	C	554	0.18	0.44
9	BE	C / C++	167	9.47	5.45 / 4.08
10	BE	C	127	1.24	0.13
11	BE	C	260	2.87	0.07
Average of 11 Projects			398	2.82	0.93
Average of 95 Projects					1.50
Average of 50 Projects with Conventional Use of QAC			13	9.00	4.50

Note: FE: Front End of Digital Tuning, Graphic OS, Media Processing on Digital TV
BE: Back End of Signal Processing, System Control, Video Streaming on Digital TV
PC: Peripheral Control on Digital TV
AVC: Advanced Video Coding Compression/Expansion Control



Subjects: Projects to develop embedded software for AV appliances
# of Subjects: 95
Process: Primary Integration Testing
SCA Tool: QAC under the proposed approach

図-4 一次結合テストで生成された警告の内訳



Subjects: Project #5 in Table 2
Process: Secondary Integration Testing
SCA Tool: QAC under the proposed approach

図-5 プロジェクト#5 の二次結合テストで生成された警告の内訳

７．まとめ

本論文では、ある企業における民生用音響・映像機器向け組み込みソフトウェア開発に、静的コード解析ツールを全社的に導入するにあたり明らかとなった技術的課題を整理すると共に、その対策を提案し、いくつかの適用実験を通じて不具合の早期発見という観点からその効果を示した。具体的には、C／C++プログラムを対象とした代表的な静的コード解析ツールの一つであるQACの適用において、QACに実装されているコードメトリクスを、テストプロセスの初期段階から全て適用するのではなく、テスト副工程毎にその適用基準を定め、適用するコードメトリクスの種類や数を、テストプロセスが進むにつれて増やしていく。提案手法に基づいてソースコードにQACを段階的に適用するためのソフトウェアインタフェースSCA Wrapperを実装すると共に、民生用の音響・映像機器に組み込まれるソフトウェアのテスト工程に提案手法を適用し、その有効性を評価した。その結果、提案手法によってQACの適用範囲をテストプロセス全体に広げることが可能であり、テストプロセス全体として不具合修正工数がこれまでよりも更に減少することも示された。静的コード解析ツールは多数存在するが、QACによって既に低減されている不具合修正工数を、更にその１／３にまで低減できるほどの高い性能を持ったツールに関する報告は、筆者らが知る限りない。

商用ソフトウェアの多くは、プログラムサイズで見た規模が数十万LOCかそれ以上と大規模化している。実装プロセスではもちろんのこと、設計プロセスにおいても、審査基準を設けて要求仕様等のレビューを行っている。しかし、設計プロセスや実装プロセスで発見される不具合は限られており、その多くはテストプロセスで発見される。特に、本論文で対象とした民生用音響・映像機器向け組み込みソフトウェア等では、自社内で開発したモジュールに協力企業など組織外の第三者によって開発されたモジュールを結合した製品候補版（Release Candidate）、そして、試作された機器（ハードウェア）に全モジュールが組み込まれた量産出荷直前の製品最終版（Golden Master）へと進むにつれて、アーキテクチャやアルゴリズムにかかわる設計上の不具合の修正には、非常に多くの工数が必要となる。レビュー・テスト方法の工夫や審査基準の厳格化も進められているが、規模が数十万LOCともなると、人手による不具合発見には限界があり、出荷遅延や出荷後に重大不具合が発見されるといった事態は後を絶えない。

本論文で注目した静的コード解析ツールは、人手によるコードレビューなどに比べて見逃しや漏れが少なく、かつ、高速である。解析対象はプログラムコードであるが、生成された警告の多くは、設計プロセスに起因する不具合につながるものであることが、適用実験で示された。静的コード解析ツールの段階的適用というアプローチは、ソフトウェア開発現場に端を発したものであり、提案手法は不具合修正に関する現場ノウハウの体系化でもある。また、本論文で用いたQACは、商用の静的コード解析ツールとして最も普及しているものの一つであり，特に、日本企業におけるソフトウェア開発において広く用いられている。QACでは、C/C++で記述されたソースコードが解析対象となるが，国内における組み込みソフトウェア開発の約60%においてCプログラムが，約21%においてC++/C#プログラムが用いられている[15]．同一組織内とはいえ、95のプロジェクトに適用され、不具合修正工数の削減に一定の効果が認められた点からも、提案手法の有用性や実用的は高く、日本の組み込みソフトウェア開発に大きく貢献するものである。

提案手法は、QACを対象としたものであるが、「解析に用いるメトリクスを段階的に増やす」というアプローチには高い汎用性があり、QAC以外の静的コード解析ツール、特に、多数のメトリクスで解析を行う「メトリクス集積型解析ツール」に転用、応用できる。すなわち、解析ツール毎に異なるのはメトリクスであり、同アプローチそのものはツールに依存しない。しかも、QACが解析に用いるメトリクスは、表-1で示したように多岐にわたり、その多くは他の解析ツールでも用いられている。それらメトリクスについては，本論文の成果をそのまま転用することができる。解析ツールが独自に定義するメトリクスについては、その適用時期を別途検討することになるが、その検討においても、本論文の５章２節で示した適用基準が参考となる。本論文の提案手法やその適用実験の結果は、その汎用性から、広く多くのソフトウェア開発者にとって有用である。

提案手法は、プログラム言語（解析対象とするソースコードの記述言語）への依存性も低い。解析ツールで用いられるメトリクスの定義自体は，その大半がプログラム言語に依存しない。例えば、表-1で示した「STUNV: Number of Unused Variables」は、変数（Variable）を用いる一般的な手続き型プログラム言語であればその数を数えることが可能である。提案手法において，どのメトリクスをどのテスト副工程で適用するのかは，各メトリクスの実装ではなく定義によって定まる。STUNVを適用すべきテスト副工程は、プログラム言語に関わらず、更に言えば、用いる解析ツールに関わらず同じとなる。提案手法は、QACの技術的課題を解決するために開発されたものであるが、QACやその解析対象であるC/C++プログラムのみに適用対象が限定されるものではない。本論文の提案手法やその摘要実験の結果は、プログラム言語への依存性が低いという点からも、広く多くのソフトウェ

ア開発者にとって有用である。

本論文では、静的コード解析ツールをテストプロセスにおいて段階的に適用するという、これまでにない独創的なアプローチにより、同ツールの技術的課題を解決した。同アプローチは多くの静的コード解析ツールへの転用、応用可能であり、プログラム言語への依存性も低い。今後は、その更なる改良に向け、静的コード解析ツールが生成する個々の警告とそれによって発見される、また、発見されない不具合との関係を、不具合修正工数の平均だけでなく、分布や標準偏差などの観点からもより詳細に明らかにするとともに、QAC 以外の静的コード解析ツールへの転用や応用の実践や評価、ソフトウェア開発組織や適用対象ソフトウェアの拡大が望まれる。

References

[1] D. Baca, "Identifying Security Relevant Warnings from Static Code Analysis Tools through Code Tainting," *Proc. of* 2010 *International Conference on Availability, Reliability and Security*, pp.386-390, 2010.

[2] B. Chelf and C. Ebert, "Ensuring the Integrity of Embedded Software with Static Code Analysis," *IEEE Software*, Vol. 26, No.3, pp96-99, 2009.

[3] C. Ebert and R. Dumke, *Software Measurement*, Springer, 2007.

[4] M. H. Halstead, *Elements of Software Science*, Elsevier Science, 1977.

[5] S. C. Johnson, "Lint: A C Program Checker," *Computer Science Technical Report*, 65, Bell Laboratories, 1977.

[6] 古賀国秀，山元和子，ソースコード静的解析技術，東芝レビュー，Vol.64，No.4，pp.44-47，2009.

[7] A. Krusko, "Complexity Analysis of Real Time Software, -Using Software Complexity Metrics to Improve the Quality of Real Time Software, *Master's Thesis in Computer Science*, TRITA-NA-E04032, Department of Numerical Analysis and Computer Science, Royal Institute of Technology, 2004.

[8] T. J. MaCabe, "A Complexity, Measure," IEEE *Transactions on Software Engineering*, Voi.SE-2, No.4, pp.308-320, 1976.

[9] R. Plosch, H. Gruber, C. Korner, and M. Saft, "A Method for Continuous Code Quality Management using Static Analysis," *Proc. of* 7th *International Conference on the Quality of Information and Communications Technology*, pp.370-375, 2010.

[10] C. Sadowski, J. Van Gogh, C. Jaspan, E. Soderberg, C. Winter, "Tricorder: Building a program analysis ecosystem," *Proc. of* 37th *International Conference om Software Engineering*, pp.598-608, 2015.

[11] B. A. Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsborrow, N. J. Ward and D. W. R. Marsh, "Industrial Perspective on Static Analysis," *Software Engineering Journal*, Vol.10, No.2, pp.69-75, 1995.

[12] *Getting Started with* QAC / QAC++, Programming Research, 2007. http://products.programmingresearch.com/Docs/Getting_Started_2.0.pdf.

[13] 情報システムの信頼性向上に関するガイドライン 第 2 版，経済産業省，2009.

[14] "The Economic Impact of Inadequate Infrastructure for Software Testing," Planning Report 02-3, National Institute of Standards and Technology, 2002.

[15] 2012 年度ソフトウェア産業の実態把握に関する調査，独立行政法人情報処理推進機構，2013.

（2016 年 7 月 1 日受理）