情報処理学会論文誌　Vol. 51　No. 9　1859–1872 (Sep. 2010)

プログラム自動可視化ツール Avis を利用した結合テスト実施のための実行経路抽出手法の提案

喜　多　義　弘[†1]　片　山　徹　郎[†2]　冨　田　重　幸[†2]

ソフトウェアテストにおける結合テストの目的の 1 つは，モジュール間インタフェースの正当性を確認することである．そのためには，ホワイトボックステストによってプログラムの構造を考慮する必要があるが，テストを実施するための実行経路が爆発的に増大するという問題がある．そこで本研究では，結合テストにおいてモジュール間インタフェースの正当性を確認するために，実行経路数を削減し，全モジュールまたは全モジュール呼び出しを実行するための実行経路を，結合テスト実施前に提示するテスト支援手法について提案する．実行経路はプログラム自動可視化ツール Avis（**A**utomatic **Vis**ualization Tool for Programs）を用いて自動的に抽出する．本手法で得た実行経路を用いることによって，モジュール間における引数によるデータ授受または広域変数によるデータ授受の正当性確認の支援につながるため，モジュール間インタフェースの正当性を確認できる．さらに，実行経路数を削減することによって，効率良くテストを実施できるため，ソフトウェアテストの生産性の向上にもつながる．

Study on an Abstraction of Paths for Integration Testing by Using an Automatic Visualization Tool 'Avis'

Yoshihiro Kita,[†1] Tetsuro Katayama[†2]
and Shigeyuki Tomita[†2]

One of the goals of the integration testing is checking the justification of module interfaces. It is necessary to execute the integration testing with a white-box testing, but, there is a problem that the execution paths increase explosively. This research proposes the paths abstraction by using the existing automatic visualization tool Avis (**A**utomatic **Vis**ualization Tool for Programs) for the white-box testing before the integration testing. The proposed method can abstract the paths which express the execution for covering of the all modules or the all module-calls, and can reduce the number of the execution paths. It can check the justification of module interfaces, because it can support to check the justification of transfer of data by arguments and global variables between modules. Moreover, it can be executed the software testing efficiently, because it can reduce the number of execution paths.

1. はじめに

近年，企業の合併にともなうシステム統合や，ビジネス環境の激変に合わせたシステムの修正が増え，システムの肥大化ならびに複雑化が進んでいる．一方で，ソフトウェア開発では，マーケットの競争が激化し，非常に短期間で開発することを迫られており，要求や仕様の変更などにより，開発に遅れが出ることも多々ある．開発の短期化や遅れが原因で，システム稼動前のテストを十分にできず，そのためにシステム・トラブルを防げなかった事例が増えてきている．頻発するシステム・トラブルを事前に防ぐために，ソフトウェアテストを重要視する声が高まっている[1)]．

ソフトウェアテストにおいて，単体テストを対象とした研究は様々行われているが，その次段階で実施する結合テストを対象とした研究については，報告例が単体テストと比べて少ない．結合テストは，複数個のモジュールを組み合わせて，仕様書どおりに動作するか否かを確認するテストであり，結合テストにおいて確認すべき項目は以下の 3 つである[2)]．

- モジュール間インタフェースの正当性
- 機能の完全性（仕様書との整合性）
- 局所あるいは大域データに対する操作の正当性

モジュール間インタフェースとは，モジュール間で授受されるデータのことである．結合テストは，ブラックボックステストでの実施が中心となる[1),3),4)]．しかし，ブラックボックステストでは，プログラムの内部構造は考慮せず，与えられた入力に対して妥当な出力が得られるかどうかを確認することが目的であるため，機能の完全性およびデータの操作の正当性については確認できるが，モジュール間インタフェースの正当性については，プログラムの内部構造を考慮する必要があるため，十分に確認できない．そのため，結合テストにおいてプログラムの内部構造を考慮するホワイトボックステストを行う必要がある．

†1 宮崎大学大学院工学研究科
Interdisciplinary Graduate School of Engineering, University of Miyazaki
†2 宮崎大学工学部情報システム工学科
The Department of Computer Science and Systems Engineering, Faculty of Engineering, University of Miyazaki

© 2010 Information Processing Society of Japan

モジュール間インタフェースの正当性を確認するには，少なくとも全モジュール間のつながりの正当性を確認する必要があるため，すべてのモジュールおよびモジュール呼び出しを網羅する必要がある．しかし，それらを網羅する実行経路数は膨大になるため，そのすべての実行経路を実行することは不可能に近い．また，その膨大な実行経路から，すべてのモジュールおよびモジュール呼び出しを網羅する必要最低限の実行経路を人手によって選択することは，各モジュールの実行経路が複雑に絡み合うため困難である[2]．その困難さは，ソフトウェアテストを支援するツールにも表れている．結合テストのブラックボックステスト手法を支援するツール（たとえば，Conformiq[5] など）は多く存在するものの，結合テストのホワイトボックステスト手法を支援するツールは少ない．

そこで本研究では，統合テストにおいてモジュール間インタフェースの正当性を確認するために，実行経路数を削減し，全モジュールまたは全モジュール呼び出しを実行するための実行経路を，結合テスト実施前に提示するテスト支援手法について提案する．実行経路は独自に開発した既存のプログラム自動可視化ツール Avis（**A**utomatic **Vis**ualization Tool for Programs）[6)–9)] を用いて自動的に抽出し提示する．

2 章では，本提案手法の位置づけについて述べる．3 章では，実行経路を抽出するための指標について述べる．4 章では，Avis の既存機能について実行経路の導出を中心に述べる．5 章では，余分な実行経路を削減するための Avis の改良について述べる．6 章では，改良した Avis を用いた実行経路抽出手法について評価し，考察を行う．

2. 本提案手法の位置づけ

本提案手法を含む，ツールを用いたプログラムの可視化によるテスト支援手法には，可視化ツールの研究およびテスト技法に関するツールの研究の 2 つの研究分野が関わっている．

可視化ツールの研究目的は，プログラムを理解するためにプログラムを読みやすいように支援することである．そのため，可視化ツールの多くはデータフローや制御フローなどのプログラムの流れや振舞いを可視化した情報を提供する．一方，テスト技法に関するツールの研究目的は，プログラムに存在する誤りを発見することであり，テストやテストケース生成などを自動化し，人手では困難なテストの実施やプログラムの誤りの発見を行う．

本提案手法で扱う Avis は，可視化ツールの位置づけである．Avis は，プログラミング教育支援のための，プログラムを読みやすくすることを目的として，我々が独自に開発した，既存のプログラミング自動可視化ツールである[6)–9)]．Avis は，プログラムの流れを示すフローチャート，プログラムの振舞いを示す実行経路をそれぞれ提示する．

本提案手法における Avis の役割は，Avis が提示する実行経路によって，プログラムの全モジュールおよび全モジュール呼び出しを網羅するために必要なプログラムの振舞いを提示することである．Avis は実行経路を作成する際，実行経路数を抑えるためにテストカバレッジの概念を用いていた．そこで今回は，そのテストカバレッジを用いている Avis の特徴を活かし，Avis によるテスト支援を試みる．

現在の Avis は結合テスト支援のために使うには無駄が多いため，Avis の改良を行う．本論文では，Avis の既存機能について説明した後，Avis の結合テスト支援のための改良について述べ，Avis および本提案手法の結合テストへの有用性について評価を行う．

3. 実行経路抽出のための指標

膨大な実行経路数を実用的な数に抑えるための既存の指標として，テストカバレッジという概念がある．モジュール間の呼び出しに関するテストカバレッジとして，主に以下の 2 つがある[2]．

- モジュール網羅率（S0）
  すべてのモジュールのうち，何%のモジュールが呼び出されたかを示す．
- コールペア網羅率（S1）
  すべてのコールペアのうち，何%のコールペアが実行されたかを示す．コールペアとは，モジュール呼び出し文とそれによって呼び出されるモジュールの組のことを指す．

これらのテストカバレッジは，テストの終了基準を定めるための指針である．すべてのモジュールまたはコールペアを網羅するには，S0 または S1 を 100%満たせばよい．これを満たす実行経路を Avis によって提示することにより，S0 を 100%満たす実行経路によって実行されないモジュールを発見でき，S1 を 100%満たす実行経路によって実行されないモジュール呼び出しを発見することができる．

ただし，S0 および S1 の概念はプログラムのソースコード解析に関する静的な概念であるため，プログラムが正しく機能しているかなどを確認する動的なテストによって発見される不具合は，S0 または S1 を 100%満たしても発見することができない．

また，前述したように，S0 または S1 を 100%満たすための実行経路を選択することは，各モジュールの実行経路が複雑に絡み合うため，困難である．そのため，S0 または S1 というテストの完了基準はあるものの，それを 100%満たすための実行経路を選択する手法については報告が少ない．

© 2010 Information Processing Society of Japan

4. Avis の既存機能

図 1 に既存の Avis の出力の一例を示す．Avis は Java プログラムのソースコードを入力とし，静的解析手法を用いて，図 1 のようにプログラム全体のフローチャート，逐次型実行経路，および，モジュール遷移型実行経路を自動的に出力する．

以下に，Avis が実行経路導出のために用いる指標および基準，実行経路の導出手順，および，複数のモジュールまたはコールペアへの対応について述べる．

4.1 実行経路導出のために用いる指標および基準

Avis は，実行経路数を抑えるための指標として，既存のテストカバレッジの 1 つである分岐網羅率（C1）を用いている．分岐網羅率（C1）とは，プログラム中のすべての分岐方向のうち，何%の分岐が実行されたかを示す，単体テストで用いる指標である．Avis は，C1 を 100%満たすような実行経路を導出する．

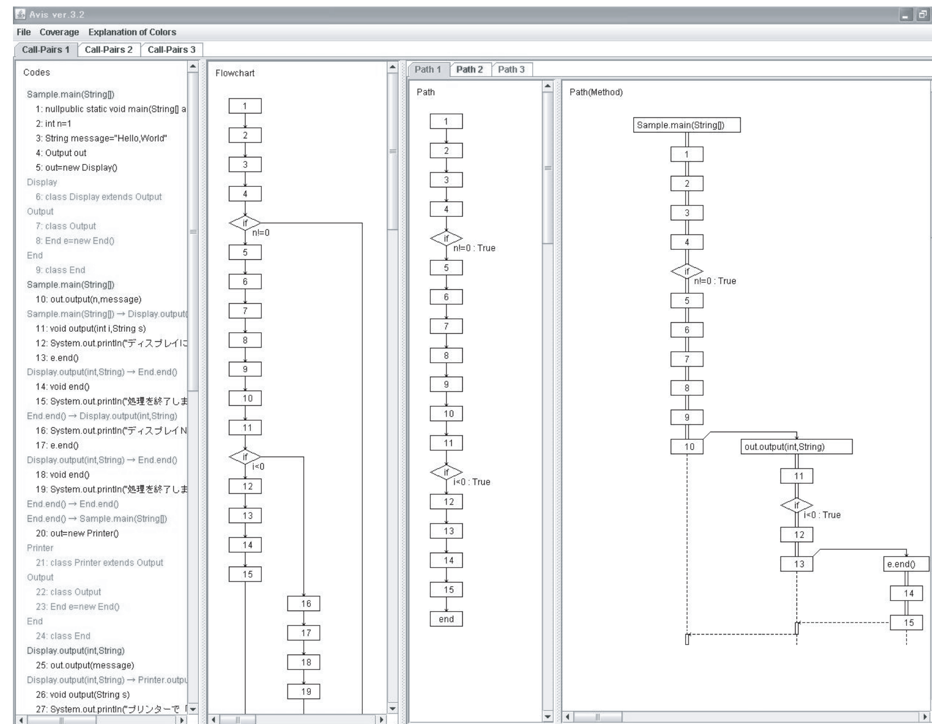


図 1　Avis の出力の一例
Fig. 1　An example of output of Avis.

また，プログラム内にループが存在する場合，実行経路数が無限になる可能性がある．無限の実行経路数を有限にとらえるために，以下の 2 つの基準を独自に設けている．

- 分岐網羅基準
  プログラム内の各分岐において，それぞれの分岐を少なくとも 1 回は実行する．
- ループ網羅基準
  プログラム内にループが存在する場合，ループの繰返し回数は 0 回（ループしない）と 1 回の場合のみを考える．

これらの基準は，C1 の概念に基づいている．

4.2 実行経路の導出手順

Avis はプログラムのソースコードからフローチャートを作成し，フローチャートから実行経路を導出する．ソースコードからフローチャートを作成できることは自明であり，文献 4) などによってフローチャートの作成法を述べているため，ソースコードからのフローチャートの作成手順については省略し，フローチャートからの実行経路の導出手順について述べる．

フローチャートから C1 を 100%満たす実行経路を導出する手順を，**図 2** を参照しながら以下に示す．

(1)　モジュール内のソースコードを基にフローチャートを作成する（図 2 の (1) 参照）．
(2)　フローチャートを分岐条件の真偽ごとに分割する（図 2 の (2) 参照）．
(3)　(2) で分割したコードおよび分岐をそれぞれ連結し，それぞれの連結を実行経路とする（図 2 の (3) 参照）．

手順 (2) によって，条件の真偽それぞれを 1 回は実行する実行経路が作成され，前述した分岐網羅基準に準じた実行経路を導出することができる．ループの場合はループ網羅基準に準ずるように，ループをしない場合と 1 回だけ行う場合とに分割する．C1 の概念に基づいた 2 つの基準を用いて実行経路を導出することにより，C1 を 100%満たす実行経路を導出することができる．

4.3 モジュール間のつながりへの対応

C1 は単体テストで用いるテストカバレッジであるため，複数のモジュールを含むプログラムの場合，モジュール呼び出しなどのモジュール間のつながりに対して C1 の概念を扱い

© 2010 Information Processing Society of Japan

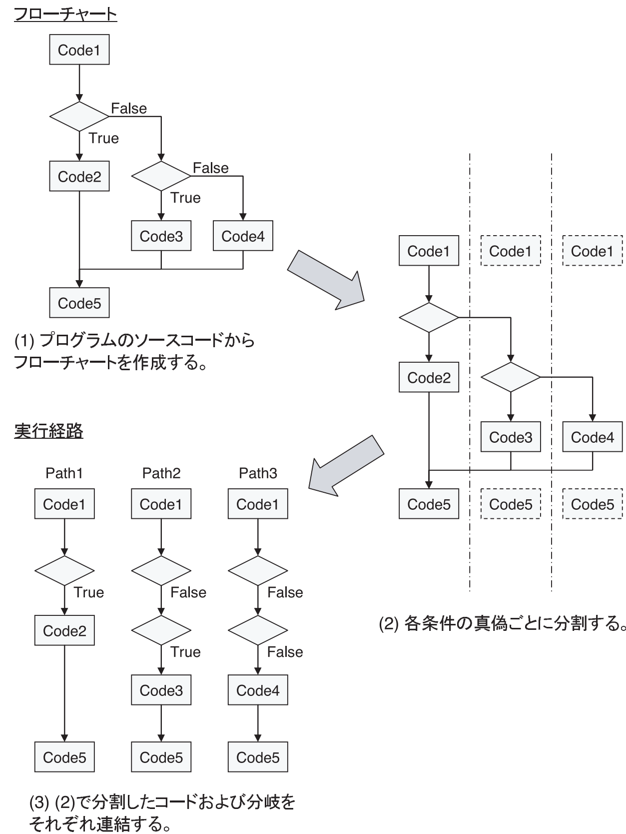


図 2　フローチャートからの実行経路の導出手順

Fig. 2　Processes of deriving of the paths from a flowchart.

にくく，C1 を 100%満たす実行経路を導き出すことが困難である．そこで，インライン展開を用いてモジュールの単一化を図る．インライン展開とは，コンパイラの研究分野における最適化手法の 1 つであり，呼び出し元のモジュールに呼び出し先のモジュールのコードを展開し，モジュール間の制御転送をしないようにする手法である[10]．モジュールの単一化を行うことにより，複数のモジュールを 1 つの大きなモジュールとして扱うことができるため，C1 を 100%満たす実行経路を導くことができる．

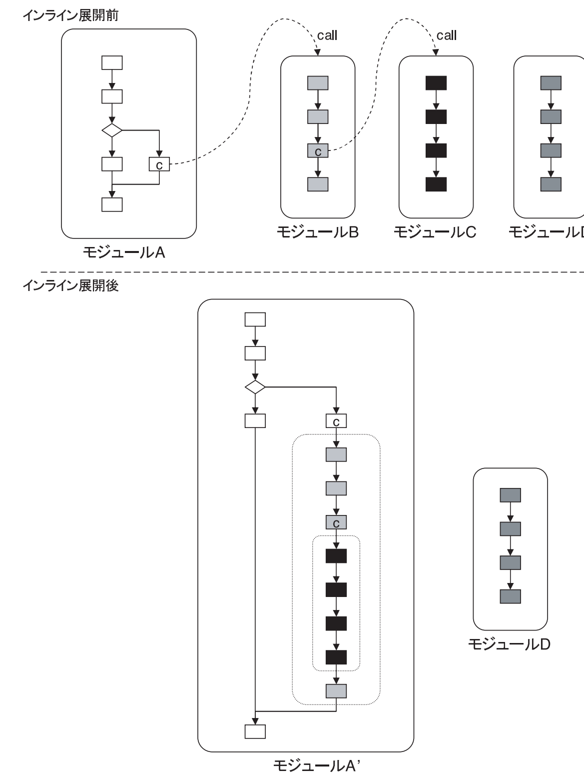


図 3　インライン展開の例

Fig. 3　An example of inline expansions.

図 3 に，インライン展開の例を示す．A〜D の 4 つのモジュールがあり，各モジュール内のノードはコードを，アークは処理の流れをそれぞれ示す．モジュール A および B 内のノード 'c' はモジュール呼び出し文である．そして，それぞれのノードをアークでつないだフローチャートによって各モジュールの流れを示している．モジュール A はモジュール B を，モジュール B はモジュール C を呼び出し，モジュール D はどのモジュールからも呼び出されない．

これらのモジュールをインライン展開すると，モジュール B はモジュール A 内に，モジュール C はモジュール B 内に取り込まれ，モジュール D 以外のモジュールが単一化さ

© 2010 Information Processing Society of Japan

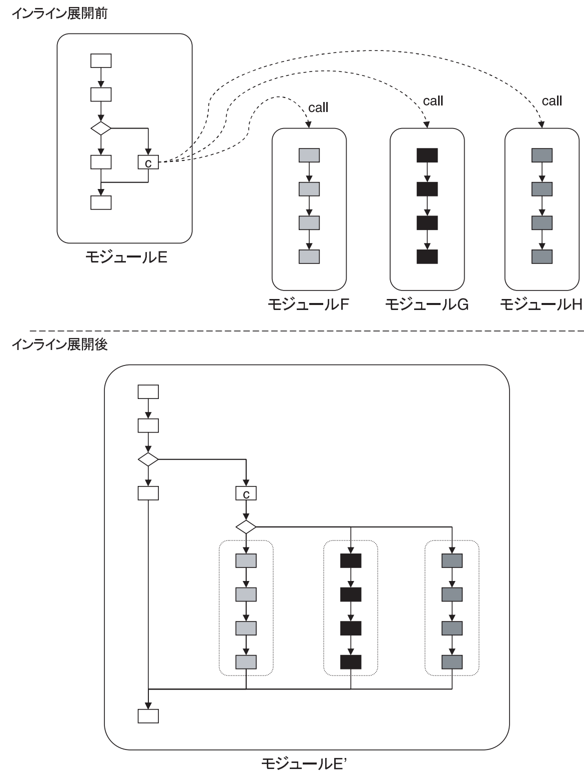


図 4　モジュール呼び出し文と呼び出し先のモジュールが 1 対多の関係である場合におけるインライン展開の例
Fig. 4　An example of inline expansions in the case of a module-call calls modules.

れる．単一化したモジュールをモジュール A' とする．モジュールを単一化することにより，モジュール呼び出しは互いのモジュールのフローチャートをつなぐアークとなり，1 つの大きなフローチャートにまとめられる．

モジュール A' のコードすべてを実行することによって，モジュール A，モジュール B，および，モジュール C のコードをすべて実行したことと同様になり，これらのモジュールまたはコールペアを実行したことになる．そのため，モジュール A' とモジュール D の C1 を満たす実行経路を導き出すことによって，すべてのモジュールまたはコールペアを網羅できる．ここで，モジュール A' やモジュール D のように，インライン展開後のモジュール単位を連結モジュールと呼ぶことにする．なお，図 3 のインライン展開は，モジュール呼び出し文と呼び出し先のモジュールが 1 対 1，または，多対 1 の関係である場合に適用する．

モジュール呼び出し文と呼び出し先のモジュールが 1 対多の関係である場合におけるインライン展開の例を，**図 4** に示す．E〜H の 4 つのモジュールがあり，モジュール E 内の 1 つの呼び出しによって複数のモジュールのうち 1 つを呼び出している．このような呼び出しの場合は，分岐を用いて図 4 のようにインライン展開をし，分岐処理として扱う．図 4 においてインライン展開後のモジュールをモジュール E' とする．モジュール A' と同様にモジュール E' の C1 を満たす実行経路を導き出すことによって，すべてのモジュールまたはコールペアを網羅できる．

また，再帰呼び出しなどがある場合，インライン展開によって，単一化したモジュールの規模が無限に増大する可能性がある．モジュールの規模を有限にするために，以下の 2 つの基準を独自に設ける．

- モジュール網羅基準
  すべてのモジュールを少なくとも 1 回は実行する．
- 再帰網羅基準
  モジュールの再帰呼び出しが存在する場合，再帰回数は 1 回の場合のみを考える．

これらの基準は，C1 の概念に基づいている．

以上により，Avis を用いることによって，C1 を 100%満たしつつすべてのモジュールまたはコールペアを網羅する，すなわち，S0 または S1 を 100%満たす実行経路を導出することができる．

5. 結合テストのための Avis の改良

Avis は，C1 を 100%満たすと同時に S0 または S1 を 100%満たす実行経路を導出することができる．しかし，提示する実行経路は C1 を 100%満たすための実行経路であるため，構成されるモジュールやモジュールの呼び出しが重複する実行経路が存在している．それらの実行経路は，S0 または S1 を 100%満たす実行経路を示すには余分である．そのため，それらの実行経路から，モジュールやコールペアが重複しない実行経路を選択する必要がある．また，Java 特有の機能であるインスタンス，継承，および，ポリモーフィズムにも対応する．

以下に，Avis への実行経路選択機能の追加，インスタンス，継承，ポリモーフィズム，ソースコードを用いることができないモジュールへの対応，および，実行経路の実行可能性

　© 2010 Information Processing Society of Japan

について述べる．

5.1　Avis への実行経路選択機能の追加

現在の Avis が提示する実行経路は，構成するモジュールやモジュール呼び出しが重複している実行経路があり，S0 または S1 を 100%満たすための実行経路として用いるには無駄が多い．そのため，提示した実行経路から S0 または S1 を 100%満たすために必要な実行経路を選択する必要がある．そこで我々は，数理計画法における組合せ最適化の既存の解法である，「けちけち法（stingy method）」[11] を用いて，提示した実行経路から必要な実行経路を選択する．

例として，あるプログラムのモジュール間のつながりを，**図 5** に示す．モジュール A および C は内部に分岐を含んでおり，他のモジュールは逐次処理のみの構成とする．モジュール A 内の分岐によって処理 a〜c に分かれており，各処理によって，モジュール B〜D がそれぞれ呼び出されている．モジュール C 内の分岐内の処理 d または処理 e は，モジュール呼び出しには関係のない処理である．また，モジュール呼び出し後の処理の戻りは省略する．

このプログラムにおいて，C1 を 100%満たすための実行経路は以下の 4 本ある．

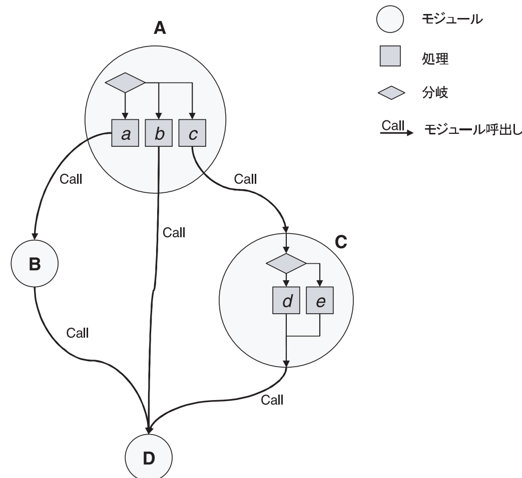


図 5　モジュール間のつながりの一例
Fig. 5　An example of connection with modules.

$Path1 \ldots A(a) \rightarrow B \rightarrow D$

$Path2 \ldots A(b) \rightarrow D$

$Path3 \ldots A(c) \rightarrow C(d) \rightarrow D$

$Path4 \ldots A(c) \rightarrow C(e) \rightarrow D$

これらの実行経路から，S1 を 100%満たすための実行経路を洗い出す．それぞれの実行経路におけるコールペアの集合を，以下に示す．

$Path1' = \{A \rightarrow B, B \rightarrow D\}$

$Path2' = \{A \rightarrow D\}$

$Path3' = \{A \rightarrow C, C \rightarrow D\}$

$Path4' = \{A \rightarrow C, C \rightarrow D\}$

$Path4' = Path3'$ より，$Path3$ および $Path4$ はモジュール C 内の分岐の処理が異なるだけで，モジュールのコールペアの集合は同じである．そのため，S1 を 100%満たすためには，$Path1$ および $Path2$ に加え，$Path3$ または $Path4$ のいずれかでよい．ここでは $Path3$ を選択することにし，S1 を 100%満たす実行経路は $Path1$，$Path2$，および，$Path3$ の 3 本となる．

次に，洗い出した 3 本の実行経路から，S0 を 100%満たすための実行経路を洗い出す．それぞれの実行経路において構成されているモジュールの集合を，以下に示す．

$Path1'' = \{A, B, D\}$

$Path2'' = \{A, D\}$

$Path3'' = \{A, C, D\}$

$Path2'' \subset Path1''$ より，$Path2$ を構成するモジュールは，$Path1$ を実行することにより網羅できることになる．そのため，すべてのモジュールを網羅するには，$Path1$ および $Path3$ を実行するだけでよい．これにより，S0 を 100%満たす実行経路は $Path1$ および $Path3$ の 2 本となる．

C1 を 100%満たす実行経路から，構成されるモジュールやコールペアが重複しない，S0 または S1 を 100%満たす実行経路を抽出するためのアルゴリズムを，以下に示す．

(1)　2 本の実行経路を選び，それぞれを cp1，cp2 とする．
(2)　cp1 と cp2 を比較し，それぞれ一致するコールペアの数をカウントする．
(3)　cp1 の全コールペア数=(2) でカウントしたコールペア数の場合，cp1 を除外し，(1) に戻る．
(4)　cp2 の全コールペア数=(2) でカウントしたコールペア数の場合，cp2 を除外，かつ，

© 2010 Information Processing Society of Japan

他の実行経路を新たに cp2 とし，(2) に戻る．
(5)　(3) または (4) にあてはまらない場合，cp1 と未比較の実行経路があればそれを cp2 に，なければ cp2 を cp1，他の実行経路を cp2 として，(2) に戻る．
(6)　すべての実行経路を比較し終えた時点で除外されずに残っている実行経路を，「S1 を満たす実行経路」とする．
(7)　(6) で得た実行経路から 2 本の実行経路を選び，それぞれを mp1，mp2 とする．
(8)　それぞれの実行経路はどのモジュールによって構成されているかを調べる．
(9)　mp1 を構成するモジュールが mp2 を構成するモジュールの部分集合である場合，mp1 を除外し，(7) に戻る．
(10)　mp2 を構成するモジュールが mp1 を構成するモジュールの部分集合である場合，mp2 を除外，かつ，他の実行経路を新たに mp2 とし，(8) に戻る．
(11)　(9) または (10) にあてはまらない場合，mp1 と未比較の実行経路があればそれを mp2 に，なければ mp2 を mp1，他の実行経路を mp2 として，(8) に戻る．
(12)　すべての実行経路を比較し終えた時点で除外されずに残っている実行経路を，「S0 を満たす実行経路」とする．

Avis が C1 を 100%満たす実行経路を作成した後に，上記のアルゴリズムが動作するように，Avis を改良する．

5.2　インスタンスへの対応

インスタンスとは，あるモジュールを雛形として作成されるオブジェクトのことである．

Java プログラムにおいて，インスタンス生成の際は，そのインスタンスの雛形となるクラスのメンバ変数やアブストラクトの実行を暗黙的に行う．そして，インスタンスによるメソッド呼び出しは，そのインスタンスの雛形となるクラスのメソッドを呼び出していることと同等である．これらにより，インスタンスは雛形となるクラスを呼び出すためのモジュール呼び出しととらえることができる．そのため，S0 の概念においては，インスタンスの雛形となるクラスを網羅の対象とし，S1 の概念においては，インスタンス生成およびインスタンスによるメソッド呼び出しを網羅の対象とする．つまり，インスタンス名は異なるが雛形となるクラスが同じであるインスタンスが複数存在している場合は，モジュール呼び出し文と呼び出し先のモジュールが 1 対 1 もしくは多対 1 の関係である場合ととらえることができる．

このことをふまえて，Avis はインスタンスの雛形となるクラスを呼び出し先のモジュール，インスタンス生成やインスタンスによるメソッド呼び出しをモジュール呼び出し文として，インライン展開を行う．インスタンス生成のコードがあった場合，モジュール呼び出しと同様に，そのコードの次に雛形となるクラスのメンバ変数の宣言やアブストラクトの処理を展開する．

5.3　継承への対応

継承とは，あるモジュールの機能をそのまま引き継いで別のモジュールを作成するという概念である．

Java プログラムにおいて，継承により継承先のクラスは継承元のクラスのメソッドを所持することになる．しかし，継承先のクラスのメソッドは継承元のクラスのメソッドの写しであるため，オーバライドがない限りは，それらのメソッドは同等であるといえる．オーバライドとは，継承先のクラスにおいてメソッドを再定義することである．本論文では，継承先のクラスのメソッドが呼び出されても，オーバライドがない場合は，継承元のクラスのメソッドを呼び出すこととし，S0 の概念においては継承元のクラスのメソッドを網羅した場合，そのメソッドと同等である継承先のクラスのメソッドも網羅したこととする．

5.4　ポリモーフィズムへの対応

ポリモーフィズムとは，プログラミング言語の各要素（定数，変数，式，オブジェクト，関数，メソッドなど）についてそれらが複数の型に属することを許すという概念である．オブジェクト指向言語には欠かせないものの 1 つとなっている．特に，継承によるポリモーフィズムは広く用いられている．

継承によるポリモーフィズムにおいては，オーバライドにより実行しないモジュールの存在を考慮する必要がある．たとえば，Java プログラムにおいて，親クラスの A メソッドが子クラスに継承され，さらに子クラスで A メソッドを再定義した A' メソッドが存在する場合，子クラスを呼び出せば A' メソッドが実行される．ここで，親クラスの A メソッドは実行しないモジュールとなる．

本研究では S0 または S1 を 100%満たすための実行経路を提示することが目的であるため，実行しないモジュールを含む実行経路も提示する．これにより，実行不可能なモジュールを発見することができる．そして，「親クラスを呼び出すつもりが子クラスを呼び出してしまった」などのモジュール呼び出しの誤りがあった場合，呼び出すはずのモジュールがその実行不可能な実行経路内に存在していれば，その誤りに気づくことができる．

ポリモーフィズムとして，Java プログラムのある 1 つのクラス内において，メソッドの型または引数の型や個数が異なる同名のメソッドが存在する場合もある．Avis は静的解析によって Java プログラムのソースコードを解析している．そのため，引数の個数違いによ

© 2010 Information Processing Society of Japan

るメソッドの区別はできるが，メソッドの型違いまたは引数の型違いによるメソッドの区別はできない．そこで，それらの型が異なるだけのメソッドが複数存在する場合は，引数の個数によって呼び出し候補であるメソッドをある程度絞り込み，絞り込んだメソッドをすべて提示する方法をとる．

呼び出し候補となるメソッドは，呼び出し文と呼び出し先のメソッドが 1 対多であるととらえ，すべてインライン展開し，実行経路を提示する際は，それらのメソッドを分岐処理として扱う（図 4 参照）．分岐処理として扱うことによって，実行経路が無限に増加する可能性があるため，分岐網羅基準を用いて実行経路を有限数に抑える．

これらにより，ポリモーフィズムに対応した実行経路を提示することができる．

5.5　ソースコードを用いることができないモジュールへの対応

モジュールの呼び出しは，プログラム内だけでなく，GUI やフレームワークからの呼び出しまたはライブラリへの呼び出しなど，ソースコードを参照できないモジュールも含んでいる．それらのモジュールへの対応を以下に述べる．

- 参照できないモジュールへの呼び出し
  ライブラリなどの参照できないモジュールへの呼び出しが存在する場合，呼び出し文は「呼び出し文」として認識するが，通常の逐次処理と同様に可視化を行う．Avis は C1 を 100%満たす実行経路を前もって作成するため，この呼び出し文は網羅され，「呼び出し文」として認識しているため，S0 または S1 を 100%満たす実行経路に抽出できる．
- 参照できないモジュールからの呼び出し
  GUI やフレームワークから呼び出される場合，呼び出されたそれぞれのメソッドを起点とした実行経路を作成する．そして，それらのメソッドは main メソッドにつながるメソッドからの呼び出しがないため，「main メソッドとつながらない実行経路」として扱い，main メソッドを含む実行経路とは別に提示する．main メソッドにつながっていないため，「実行されないメソッド」と同様に扱われる．そのため，なぜ main メソッドにつながっていないのかという原因を各実行経路で調べる必要がある．

ただし，本手法が適用できるのは，呼び出し側または呼び出される側のどちらかに必ずソースコードが参照できる場合のみであり，Avis に入力するプログラムのソースコードが参照できない場合は実行経路を得ることができない．

同様に，ファイル，データベース，ネットワーク，タイマなどの実際にプログラムを実行しないと正しく機能しているかどうかを確認できないテスト対象は，従来どおりブラックボックステストなどの動的なテスト手法を行う必要がある．そのため，Avis が提示する実行経路を使ったホワイトボックステストだけでなく，従来のブラックボックステストも併用しながら結合テストを行うことが必要である．

5.6　実行経路の実行可能性

Avis はプログラムを静的に構文解析し，分岐においては条件式の値に関係なく真偽別の実行経路を作成する．そのため，作成した実行経路どおりにプログラムを実行できない場合がある．このような実行経路における実行不可能な原因として，以下の 2 つが考えられる．

- 原因 1　分岐条件間などの依存関係を無視した実行経路を示している．
- 原因 2　絶対に実行できない部分（デッドコード）が含まれている．

原因 1 の分岐条件間の依存関係は，静的な構文解析だけでは判断することが難しい．そのため Avis では，依存関係は考慮せず，構文上実行しうる実行経路を作成する．これにより，実行不可能な実行経路が生成されうるが，構文どおりの実行経路を得ることができるため，多重分岐などの分岐の複雑化によって見落としやすい誤りや予期しない流れなどを発見しやすくなると考えられる．

原因 2 のデッドコードは，プログラムを実行するうえでは問題がなくても，その存在がプログラムにとって予期しない部分または本来実行すべき部分であり，プログラムの誤りとして通常扱われる．Avis が提示する実行経路にデッドコードを含む実行不可能な実行経路がある場合，実行不可能となった原因を調べることによって，実行不可能な実行経路内に含まれるデッドコードを発見することができる．

以上により，Avis が作成する実行経路が実行不可能であっても，デッドコード発見の可能性を考慮し，実行不可能な実行経路も提示する．

6.　Avis を用いた実行経路提示手法の評価および考察

今回提案した Avis を利用した実行経路提示手法の有用性を確認するための評価および考察を行う．

以下に，提示した実行経路の正当性の評価，Avis の実行時間測定によるスケーラビリティの評価，本手法の実行経路数削減の有用性の評価，実験による本手法の実行経路数削減の有用性の評価，モジュール間インタフェースの正当性確認の評価，他の結合テスト支援ツールとの比較，および，単体テストおよび回帰テストへの応用について述べる．

6.1　提示した実行経路の正当性の評価

Avis が提示した実行経路が S0 または S1 を 100%満たしているか否かについて評価を行う．例として**図 6** の Java プログラムを入力とした，Avis の出力結果を**図 7** に示す．図 6

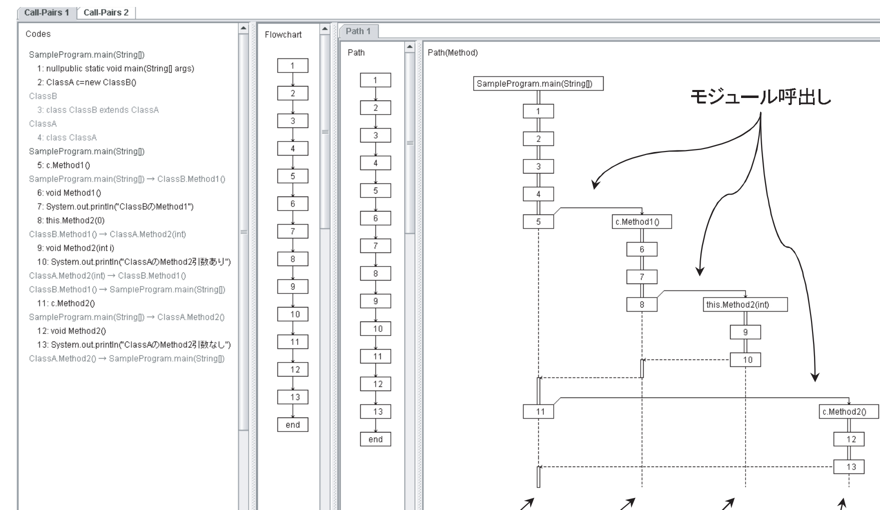
© 2010 Information Processing Society of Japan

```
public class SampleProgram {
    public static void main(String[] args) {
        ClassA c = new ClassB();
        c.Method1();
        c.Method2();
    }
}

class ClassA {
    void Method1() {
        System.out.println("ClassA の Method1");
    }

    void Method2() {
        System.out.println("ClassA の Method2 引数なし");
    }

    voif Method2(int i) {
        System.out.println("ClassA の Method2 引数あり");
    }
}

class ClassB extends ClassA {
    void Method1() {
        System.out.println("ClassB の Method1");
        this.Method2(0);
    }
}
```

図 6　Java プログラムの例
Fig. 6　An example of Java program.

の Java プログラムは，インスタンス，継承，および，ポリモーフィズムを含む簡単なプログラムである．main メソッドを含む SampleProgram クラスのほかに，ClassA クラス，ClassB クラスの 2 つのクラスがあり，ClassB クラスは ClassA クラスを継承したクラスである．ClassA クラス内には，Method1 メソッド，引数のない Method2 メソッド，および，int 型の引数がある Method2 メソッドの 3 つのメソッドを定義している．そして，モジュール呼び出しが main メソッド内に 2 カ所，ClassB クラスの Method1 メソッド内に 1 カ所の計 3 カ所ある．

図 7 より，Avis は 2 本の実行経路を示しており，それらの実行経路によってすべてのクラスやメソッドを網羅していることが確認できる．このことから，これらの実行経路によっ



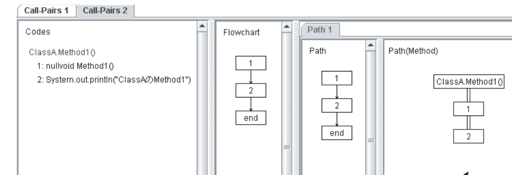


図 7　図 6 の Java プログラムを入力した Avis の出力結果
Fig. 7　Outputs of Avis that is adapted to the Java program in Fig. 6.

© 2010 Information Processing Society of Japan

て S0 を 100%満たすことがいえる．図 7 の実行経路 1 より，3 カ所のコールペアがあることが確認できる．このことから，実行経路 1 によって S1 を 100%満たすことがいえる．

以上により，Avis が提示する実行経路は S0 または S1 を 100%満たしていることがいえる．

図 7 の実行経路 2 は，ClassA クラスの Method1 メソッドの実行経路を提示している．このメソッドは，ポリモーフィズムによって実行しないモジュールであり，実行経路によりどこからも呼び出されていないことが確認できる．

また，これら 2 本の実行経路により，インスタンス生成時の雛形となるクラスの実行，および，継承によるモジュール呼び出しも確認できる．これらのことから，インスタンス，継承，および，ポリモーフィズムに正しく対応しているといえる．

6.2　Avis の実行時間測定によるスケーラビリティの評価

Avis のスケーラビリティを評価するために，規模の異なる 4 つの Java プログラムを入力として，Avis の実行時間について測定した．**表 1** にその測定結果を示す．測定は，32 bit OS Windows Vista，Intel(R) Core(TM)i7 CPU 2.93 GHz×2，メモリ 4.00 GB の環境で行い，計測には Java の System クラスの currentTimeMillis メソッドを用いた．currentTimeMillis メソッドは現在の時刻を取得するメソッドである．Avis を実行した時刻から実行経路が提示される時刻の差を実行時間として各プログラムで 10 回測定し，その平均の実行時間を示した．測定に用いた Java プログラムは以下の 4 つのプログラムである．

- プログラム A：カレンダ表示プログラム
- プログラム B：弾むボールのシミュレーションプログラム
- プログラム C：カードゲーム（ブラックジャック）のプログラム
- プログラム D：Avis 内の静的構文解析部のプログラム

表 1 の結果より，約 700 行のプログラムでは約 1 秒，約 8,000 行のプログラムでは約 10 秒の実行時間であり，ある程度の規模があるプログラムでも実用的な実行時間で Avis を実行することができるといえる．

表 1　各プログラムにおける Avis の実行時間
Table 1　Run time of Avis in each program.

Java プログラム	全コード数	実行時間（sec）
プログラム A	80	0.21
プログラム B	155	0.36
プログラム C	722	1.22
プログラム D	8,034	9.73

6.3　本手法の実行経路数削減の有用性の評価

結合テストにおいてホワイトボックステストの手法を用いる際，S0 または S1 を 100%満たすには，すべてのモジュールまたはコールペアを少なくとも 1 回は実行しなければならないため，最低でもモジュールまたはコールペアの数だけテストする必要がある．Avis は，モジュールをインライン展開によって単一化することにより，複数のモジュールをまたぐような実行経路を提示することができる．これにより，1 回の実行で複数のモジュールまたはコールペアを網羅することができるため，実行経路数が削減され，少ない実行回数ですべてのモジュールまたはコールペアを網羅できる．このことを確かめるために，Avis によってどれだけの実行経路を削減できるかを，6.2 節で用いた 4 つのプログラムを用いて評価した．各プログラムの全モジュール数と S0 を 100%満たす Avis が提示した実行経路数との比較を**表 2** に，各プログラムの全コールペア数と S1 を 100%満たす Avis が提示した実行経路数との比較を**表 3** に，それぞれ示す．

すべてのプログラムにおいて，表 2 より Avis が提示した S0 を 100%満たす実行経路数は全モジュール数の 4 割程度を，表 3 より S1 を 100%満たす実行経路数は全コールペア数の 8 割程度を削減していることが確認できる．このことから，本手法によって実行経路数を

表 2　各プログラムの全モジュール数と S0 を 100%満たす Avis が提示した実行経路数との比較
Table 2　Comparisons between the number of modules and the number of paths that cover all methods by Avis in each program.

Java プログラム	全モジュール数	S0 を 100%満たす Avis が提示した実行経路数	実行経路削減率
プログラム A	7	3	57.1%
プログラム B	23	14	39.1%
プログラム C	76	38	50.0%
プログラム D	573	352	38.6%

表 3　各プログラムの全コールペア数と S1 を 100%満たす Avis が提示した実行経路数との比較
Table 3　Comparisons between the number of modules and the number of paths that cover all call-pairs by Avis in each program.

Java プログラム	全コールペア数	S1 を 100%満たす Avis が提示した実行経路数	実行経路削減率
プログラム A	17	3	82.4%
プログラム B	66	16	75.8%
プログラム C	259	45	82.6%
プログラム D	2,277	372	83.7%

© 2010 Information Processing Society of Japan

削減することができるといえる．

また，提示した実行経路の実行可能性を考慮し，実行可能である実行経路数およびそれらによって網羅可能であるモジュール数について，S0 の場合を**表 4** に，S1 の場合を**表 5** に示し，実行不可能な実行経路に含まれるデッドコード数を**表 6** に示す．表 4 および表 5 より，プログラムの規模が大きくなるほど実行可能な実行経路の割合および網羅可能であるモジュールの割合が減少するが，表 6 より，実行不可能な実行経路内にデッドコードが含まれていることが確認できる．これらのことから，Avis が提示する実行経路は，プログラムの規模が大きくなるほど実行不可能な実行経路も多く含まれるが，すべての実行経路を実行し，実行不可能な場合はその原因を調べることにより，デッドコードを発見できる可能性があるといえる．

表 4　各プログラムにおいて S0 を 100%満たす Avis が提示した実行経路のうち実行可能な実行経路の数および割合
Table 4　The number and the rate of the executable paths in the paths that cover all modules by Avis in each program.

Java プログラム	S0 を 100%満たす Avis が提示した実行経路数…(1)	実行可能な実行経路数…(2)（(1) に対する割合）	(2) が網羅するモジュール数（全モジュール数に対する割合）
プログラム A	3	3（100.0%）	7（100.0%）
プログラム B	14	13（92.0%）	22（95.7%）
プログラム C	38	27（71.1%）	61（80.3%）
プログラム D	352	217（61.6%）	409（71.4%）

表 5　各プログラムにおいて S1 を 100%満たす Avis が提示した実行経路のうち実行可能な実行経路の数および割合
Table 5　The number and the rate of the executable paths in the paths that cover all call-pairs by Avis in each program.

Java プログラム	S1 を 100%満たす Avis が提示した実行経路数…(3)	実行可能な実行経路数…(4)（(3) に対する割合）	(4) が網羅するコールペア数（全コールペア数に対する割合）
プログラム A	3	3（100.0%）	17（100.0%）
プログラム B	16	14（87.5%）	63（95.5%）
プログラム C	45	29（64.4%）	217（83.8%）
プログラム D	372	223（59.9%）	1,558（68.4%）

表 6　各プログラムにおいて Avis が提示した実行不可能な実行経路内に存在するデッドコード数
Table 6　The number of dead-codes that be included among the unexecutable paths by Avis in each program.

Java プログラム	表 4 中の (1) のうち実行不可能な実行経路数…(5)	(5) に含まれるデッドコード数	表 5 中の (3) のうち実行不可能な実行経路数…(6)	(6) に含まれるデッドコード数
プログラム A	0	-	0	-
プログラム B	1	0	2	0
プログラム C	11	0	16	0
プログラム D	135	7	149	9

6.4　実験による本手法の実行経路数削減の有用性の評価

本手法によって削減した実行経路の有用性について，実験し評価する．実験は，まず 6.2 節で用いたプログラム A について，結合テストに必要な実行経路を被験者に求めさせる．次に，得られた実行経路を Avis が提示した実行経路と比較する．被験者は宮崎大学工学部情報システム工学科に所属する学生 5 人を対象とした．彼らは Java プログラミングの経験を持つが結合テストの経験がなかったため，実験を始める前に結合テストの方法などのレクチャを簡単に行った．

実験の結果について，S0 または S1 を 100%満たすために Avis および被験者 5 人それぞれによって提示された実行経路数を**表 7** および**表 8** に示す．表中の冗長な実行経路数とは，提示した実行経路数から S0 または S1 を 100%満たす必要最低限の実行経路数を差し

表 7　プログラム A において S0 を 100%満たすために Avis または被験者が提示した実行経路数
Table 7　The number of the paths that cover all modules by Avis or testees.

	S0 を 100%満たすために提示した実行経路数	冗長な実行経路数	不足している実行経路数
Avis	3	0	0
被験者 1	3	0	0
被験者 2	5	2	0
被験者 3	4	1	0
被験者 4	7	3	0
被験者 5	3	0	0

表 8　プログラム A において S1 を 100%満たすために Avis または被験者が提示した実行経路数
Table 8　The number of the paths that cover all call-pairs by Avis or testees.

	S1 を 100%満たすために提示した実行経路数	冗長な実行経路数	不足している実行経路数
Avis	3	0	0
被験者 1	5	2	0
被験者 2	10	7	0
被験者 3	7	4	1
被験者 4	5	2	2
被験者 5	3	0	0

© 2010 Information Processing Society of Japan

引いた値である．不足している実行経路数とは，提示した実行経路だけでは S0 または S1 を 100%満たせず，それを満たすために必要な実行経路，すなわち，見落としている実行経路の数を指す．表 7 より，被験者 5 人とも不足している実行経路はなかったが，5 人中 3 人に冗長な実行経路が見られた．また，表 8 より，被験者 5 人中 2 人に不足している実行経路が，5 人中 4 人に冗長な実行経路が見られた．冗長な実行経路が見られた原因は，複数のモジュール呼び出しを同時に実行できる実行経路を作成していなかった，または同時に実行するモジュール呼び出しの個数が少なかったことがあげられた．そして，不足している実行経路が見られた原因として，モジュール呼び出しを見逃していた，または，モジュール呼び出しを実行するための適切な実行経路を求めていなかったことから，人的要因によるものであると考えられる．

以上の結果によって，Avis が提示する実行経路は効率的に S0 または S1 を 100%満たしていると考えることができ，本手法の実行経路数削減の手法は有用性があるといえる．

6.5　モジュール間インタフェースの正当性確認の評価

モジュール間インタフェースは，広域変数によるデータ授受と引数によるデータ授受の 2 種類のデータ授受がある．

モジュール間インタフェースの正当性を確認するためには，これらのデータ授受の正当性をそれぞれ確認すればよい．すなわち Avis が提示する実行経路によって 2 種類のデータ授受の正当性を確認するための支援を行うことにより，モジュール間インタフェースの正当性を確認することができる．

広域変数によるデータ授受における誤りとして，変数の多重宣言や変数名誤りなどのプログラムの記述による誤りが考えられる．これらの誤りはプログラム内の誤りであるため，それらの誤りの発見にはブラックボックステスト手法よりもホワイトボックステスト手法が有効であると考えられる．Avis によって実行経路数を削減することによってホワイトボックステストを容易に行うことができるため，広域変数によるデータ授受の正当性を確認する際のテスト実施支援につながると考えられる．

引数によるデータ授受における誤りとして，引数の型違いや個数違いによるモジュール呼び出し誤りが考えられる．Avis は静的解析によってプログラムの解析を行っているため，引数の型違いについては判別できない（5.4 節参照）が，S1 を 100%満たす実行経路を用いてテストを行うことによって，すべてのモジュール呼び出しを確認でき，引数の型違いによるモジュール呼び出し誤りを発見する可能性がある．

引数の個数違いについては，Avis での静的解析によって判別することができる（5.4 節参照）．その際，実行されないモジュールの実行経路は別経路の実行経路として提示するため，呼び出し文と呼び出し先のモジュールの引数の個数が異なれば，呼び出し先のモジュールは呼び出されず実行されないモジュールとして提示される．そのため，引数の個数違いによるモジュール呼び出し誤りを，Avis が提示する実行経路を確認することにより，テスト実施前に発見することができる．これらにより，引数によるデータ授受の正当性を確認するために，Avis が提示する実行経路を用いてテストを行うことは有効であり，結合テストにおけるモジュール間インタフェースの正当性の確認に有効であるといえる．

6.6　他の結合テスト支援ツールとの比較

数少ない結合テストのホワイトボックステストを支援するツールの中で，カバレッジ計測ツール Emma[12] がある．このツールは，単体テストおよび結合テスト支援のための動的なカバレッジ計測ツールであり，Java プログラムのソースコードの実行経路を，ソースコード上に 3 色でハイライト表示する．Emma の機能を Eclipse[13] のプラグインとして利用できる EclEmma[14] というツールもある．

しかし，Emma は動的解析手法によってカバレッジ測定を行うため，プログラムのどこを実行し，全体の何%を網羅したかという情報は得ることができるが，実行していない残りの部分をどのように実行すればよいのか，また，S0 または S1 を 100%満たすためには，あとどれくらいの実行が必要なのかという情報は得ることができない．そのため，S0 または S1 を 100%満たすために，場合によっては膨大な数のテストを行う必要があることも考えられる．

Avis は，静的解析手法によりプログラム内の条件分岐の真偽にかかわらず，S0 または S1 を 100%満たす実行経路を提供するため，すべてのモジュールまたはコールペアを網羅するために最低限必要な実行経路やその数をテスト前に把握することができる．また，結合テストに必要な実行経路がテストを始める前に分かることにより，テストデータ作成のための指針を与えることができる．

定義使用に基づいた動的手法を用いる結合テストのためのツール[15),16] がある．これらのツールは動的手法によって実行可能なテストケースを生成するため，5.6 節の原因 1 について解決できる一手法であるといえる．しかし，実行可能なテストケースだけでは，実行可能な部分のみのテストを行うことになり，同節の原因 2 であるデッドコードを発見する可能性が低い．そのため，静的解析によって実行不可能な実行経路も提示する必要があり，この点において Avis は有用であるといえる．

© 2010 Information Processing Society of Japan

6.7 単体テストおよび回帰テストへの応用

Avis は S0 または S1 を 100%満たす実行経路を提示する前に，C1 を 100%満たす実行経路を導出するため，結合テストだけでなく C1 の概念を用いた単体テストにも応用することができる．

この応用をさらに発展させ，回帰テストへの応用も考える．回帰テストとは，プログラムをテストして不具合を発見した際，その不具合を修正した後に再び行うテストのことである．修正によってソースコードが変わってしまう場合，たとえ小さな修正であったとしても，実行経路が大きく変わる可能性があるため，これまで実行した実行経路およびテストケースが使用できないことが考えられる．この場合，現時点では実行経路やテストケースの再利用が難しく，Avis を再度実行し，新たに実行経路を得る必要がある．しかし，C1，S0，S1 の概念をあわせ持つ Avis が提示する実行経路により，単体テストおよび結合テストのホワイトボックステストを同時に実施できる．これにより，テストにかかる時間および費用を削減することができると考えられる．

また，本論文の表 1 や 6.2 節でも述べているように，ある程度規模のあるプログラムであっても Avis を実用的な実行時間で実行できることから，新たに実行経路を得ることが甚大なタスクにつながることはないと予想される．

7. おわりに

本論文では，結合テストにおいてモジュール間インタフェースの正当性を確認するために，実行経路数を削減し，全モジュールまたは全モジュール呼び出しを実行するための実行経路を，Avis を用いて自動的に抽出し，結合テスト実施前に提示する手法について提案した．提案した手法で得た実行経路によって，すべてのモジュールまたはコールペアを網羅できる．本手法で得た実行経路を用いることによって，モジュール間における引数によるデータ授受または広域変数によるデータ授受の正当性確認の支援につながるため，統合テストの目的の 1 つであるモジュール間インタフェースの正当性を確認できる．

特に，デッドコードについては，実行不可能な実行経路を調べることによって発見できる可能性が高く，引数の個数違いによるモジュール呼び出し誤りについては，提示した実行経路を確認することによって，テスト実施前に発見することができる．また，実行経路数を削減することにより，効率良くテストを実施でき，ソフトウェアの生産性向上につながる．

さらに，結合テストにおけるホワイトボックステストの実施の支援だけでなく，単体テストおよび回帰テストへの応用が見込まれることを示した．

以下に今後の課題を示す．

- 実行不可能な実行経路への対処
  5.6 節で述べたように，実行不可能な実行経路には，条件分岐間などの依存関係を無視している場合とデッドコードを含む場合との 2 つの原因が考えられる．前者は解析手法の問題であるが，後者はプログラムの誤りであるため，実行不可能である原因を調べる必要がある．そこで，これらの原因を区別するために，構文解析とあわせて意味解析を行うことが重要である．意味解析の手法としてプログラムを実行せずに解析できる静的スライシング[17]がある．この手法は，静的解析を行う Avis に最も適する意味解析の手法であるといえるが，起こりうるすべての入力を考慮した解析を行うため，不必要な情報を含みやすく，抽出した意味解析のデータが大きくなり，解析の精度が低くなるという問題がある．そのため，Avis への適用を考慮するとともに，解析の精度を向上させるための改善手法について考える必要がある．
- ソフトウェアテストの教育への Avis の応用
  従来の Avis は，プログラミング教育支援のためのプログラム自動可視化ツールである．今回，結合テストへの支援ができるように改良したことにより，ソフトウェアテストの教育支援ツールとして，Avis を利用することができると見込まれる．そのために，ソフトウェアテストに必要な要素を取り入れた，ソフトウェアテスト向けの実行経路の可視化手法を考察する必要がある．
- 結合テストのための Java 言語以外のプログラミング言語への Avis の応用
  現在の Avis は Java 言語を対象としているが，本支援手法は Java 言語だけでなく，モジュールの概念を持つ言語すべてに対応できると考えている．そこで，Avis を C 言語などの他の言語にも対応できるように改良することを考えている．

参考文献

1) 岡崎毅久：ソフトウェアテストと品質保証の実際，日本テクノセンター (1999).
2) 玉井哲雄，三嶋良武，松田茂広：ソフトウェアのテスト技法，共立出版株式会社 (1988).
3) Myers, J.G., Badgett, T., Thomas, M.T. and Sandler, C.: The Art of Software Testing, 2nd Edition, Word Association Inc. (2004). 長尾　真，松尾正信（訳）：ソフトウェア・テストの技法第 2 版，株式会社近代科学社 (2006).
4) Pressman, S.R.: Software Engineering, 6th edition, The McGraw-Hill Companies Inc. (2005). 西　康晴，榊原　彰，内藤裕史，古沢聡子，正木めぐみ，関口　梢（訳）：実践ソフトウェアエンジニアリング，株式会社日科技連出版社 (2005).

© 2010 Information Processing Society of Japan

5) 株式会社エー・アイ・コーポレーション：AIC / Conformiq Qtronic. 入手先 http://www.aicp.co.jp/products/conformiq.shtml（参照 2010-06-09）
6) Kita, Y., Kawasoe, T. and Katayama, T.: Prototype of an Automatic Visualization Tool for Java to Educate Novice Programmers, *Proc. International Conferences on Software Engineering* (*SE 2005*), *as part of the 23rd International Association of Science and Technique for Development* (*IASTED*) *International Multi-Conferences on Applied Informatics*, pp.307–312 (2005).
7) Kita, Y., Katayama, T. and Tomita, S.: Implementation and Evaluation of an Automatic Visualization Tool "PGT" for Programming Education, *Proc. 5th ACIS International Conferences on Software Engineering Research, Management and Applications* (*SERA 2007*), pp.213–220 (2007).
8) 喜多義弘，徳永友樹，片山徹郎，冨田重幸：プログラミング教育支援のためのプログラム自動可視化ツール Avis の試作，ソフトウェアエンジニアリング最前線 2007, pp.223–226 (2007).
9) Kita, Y., Tokunaga, T., Katayama, T. and Tomita, S.: Extension and Evaluation of an Automatic Visualization Tool "Avis" for Programming Education, *Proc. International Conferences on Software Engineering* (*SE 2009*), *as part of the 27th International Association of Science and Technique for Development* (*IASTED*) *International Multi-Conferences on Applied Informatics*, pp.31–36 (2009).
10) Aho, V.A., Sethi, R. and Ullman, D.J.: Compilers, Addison-Wesley Publishers Limited (1986). 原田賢一（訳）：コンパイラ I，株式会社サイエンス社 (1990).
11) 朝廣雄一，岩間一雄，玉木久夫，徳山　豪：密な部分グラフ問題の貪欲解法，電子情報通信学会技術報告，Vol.95, No.498, pp.49–58 (1996).
12) SourceForge Inc.: EMMA code coverage. available from http://sourceforge.net/projects/emma/ (accessed 2010-06-09)
13) Eclipse Foundation Inc.: Eclipse.org home. available from http://www.eclipse.org/ (accessed 2010-06-09)
14) SourceForge Inc.: EclEmma – Java Code Coverage for Eclipse. available from http://sourceforge.net/projects/eclemma/ (accessed 2010-06-09)
15) Horrold, J.M. and Rothermel, G.: Performing Data Flow Testing on Classes, *Proc. 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp.154–163 (1994).
16) Gallagher, L. and Offutt, J.: Test Sequence Generation for Integration Testing of Component Software, *The Computer Journal of Oxford University Press on behalf of the British Computer Society* (online), DOI:10.1093/comjnl/bxm093 (2007). available from http://comjnl.oxfordjournals.org/cgi/reprint/bxm093v1
17) 下村隆夫：プログラムスライシング技術と応用，共立出版株式会社 (1995).

(平成 21 年 9 月 10 日受付)
(平成 22 年 6 月 3 日採録)

喜多　義弘（正会員）

昭和 56 年生．平成 18 年宮崎大学大学院工学研究科情報工学専攻博士前期課程修了．現在，同大学院工学研究科システム工学博士後期課程在学．ソフトウェアテスト支援およびプログラミング教育支援のためのプログラム可視化手法に関する研究に従事．

片山　徹郎（正会員）

昭和 44 年生．平成 8 年九州大学大学院工学研究科情報工学専攻博士後期課程修了．同年より奈良先端科学技術大学院大学情報科学研究科助手．平成 12 年より宮崎大学工学部情報システム工学科助教授．現在，准教授．プログラムの可視化手法ならびに並行処理プログラムや組み込みシステムを対象としたテスト手法に関する研究に従事．工学博士．電子情報通信学会，日本ソフトウェア科学会各会員．

冨田　重幸（正会員）

昭和 23 年生．昭和 52 年京都大学大学院工学研究科化学工学専攻博士課程修了．同年より東京工業大学助手を経て，平成 5 年より宮崎大学工学部情報システム工学科教授．生産情報システム，離散事象システム，知識情報処理に関する研究に従事．工学博士．人工知能学会，計測自動制御学会，科学工学会各会員．

© 2010 Information Processing Society of Japan